A Vector Implementation Based On RRB-Tree for Rust

A confluently persistent vector implementation for Rust based on RRB-Tree

Araz Abishov

Thesis submitted for the degree of Master in Informatics: Programming and Networks
60 credits

Institute of Informatics
Faculty of Mathematics and Natural Sciences

UNIVERSITY OF OSLO

Spring 2020

A Vector Implementation Based On RRB-Tree for Rust

A confluently persistent vector implementation for Rust based on RRB-Tree

Araz Abishov

A Vector Implementation
Based On RRB-Tree
for Rust

http://www.duo.uio.no/

Printed: Reprosentralen, University of Oslo

Contents

Abstract

Rust is a multi-paradigm system programming language focused on performance and reliability. Its rich type system offers memory and thread-safety guarantees at compile-time.

Therefore, Rust forbids simultaneous sharing and mutation that sometimes is a necessary and useful pattern. A way to mitigate this limitation in Rust is to clone a value before sharing it. Naive cloning by copying, however, is an expensive operation both in terms of memory and performance.

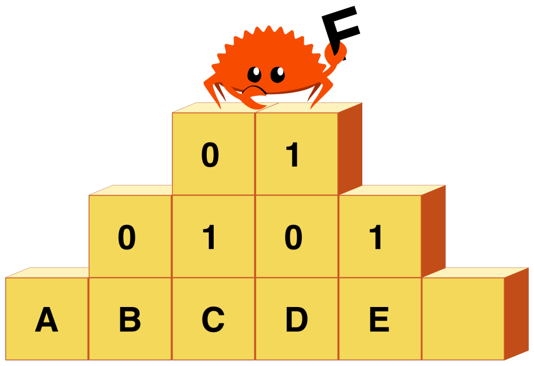
This thesis presents ***pvec-rs***, a project that contributes a vector implementation with efficient clone operation that borrows ideas from persistent data structures. The project explores novel approaches to optimize the vector's performance by leveraging Rust's ownership and borrowing rules, as well as aiming to provide a convenient, idiomatic interface familiar to developers. The proposed optimizations are evaluated and discussed based on the results of sequential and parallel test suites.

Acknowledgements

I would like to thank my supervisors, Martin Steffen and Volker Stolz, for their guidance and feedback. Without their patience and support, as well as trust in my own idea, this venture would not have been possible.

My deepest appreciation goes to my family and my friends for their moral support, their continuous encouragement, and their help whenever it was needed.

Finally, I would like to thank the Rust community for being so welcoming and supportive. In particular, I am grateful to Nicholas Matsakis for inspiring me to work on this project.



Reading notes

The links to the LaTeX source code and the latest version of this document can be found at https://abishov.com/thesis/. The implementation, documentation, and visualization demo can be found at https://abishov.com/pvec-rs.

If you notice any typos while reading the document, or have any feedback in general, feel free to open an issue at https://github.com/arazabishov/thesis/issues or send me an email at araz@abishov.com.

Colophon The illustration above with Ferris[1] sitting on top of an RRB-tree was kindly prepared by Vanessa Tesorone. The design of the reading notes and the idea to use Rust's mascot for the document decoration was inspired by the master's thesis of Erik Vesteraas[2].

Typographic conventions

Clickable link	Rust Programming Language
Inline code and types	`Vec::new()`
Project or library name	***pvec-rs***

[1]Unofficial mascot of Rust: https://rustacean.net/
[2]http://erik.vestera.as/thesis/

Chapter 1

Introduction

Rust is a modern, open-source programming language with a focus on memory safety and performance. Its rich type system eliminates several classes of bugs and makes the language powerful and expressive for building high-level programs such as web servers and command-line interface applications. With direct access to computer memory and hardware, Rust is an excellent language for embedded and low-level programming as well.

However, due to the emphasis on memory safety, it is common to get errors when building a Rust program, such as forbidden simultaneous sharing and mutation. Often, such compile-time errors of that kind can be avoided by better design, but sometimes, the best resolution is to clone the value before sharing it. Naive cloning by copying, however, is an expensive operation both in terms of time and space. Thus, resorting to it, especially for large-sized collections might be inefficient.

Persistent data structures are data structures that provide access to all their previous versions. Often, persistence is achieved through copying, and thus, various data structure designs have been developed throughout the years to optimize for this operation. For example, the standard library of the Scala programming language[1] provides a persistent vector implementation that demonstrates good performance for all operations, including copying.

[1] https://www.scala-lang.org/

This thesis presents ***pvec-rs***, a project that contributes a vector implementation with efficient clone operation that borrows ideas from persistent data structures. The project explores novel approaches to optimize the vector's performance by relying on ownership and borrowing rules enforced by Rust, as well as aiming to achieve a convenient, familiar interface to developers. The proposed optimizations are evaluated and discussed based on the results of the sequential and parallel tests.

In this chapter, we will look at the background for the ***pvec-rs*** project starting with Section 1.1 for the Rust programming language, followed by Section 1.2 dedicated to persistent data structures and their classification. Finally, Section 1.3 gives an overview of the contributions made in this project.

1.1 The Rust programming language

Rust is a relatively new programming language developed at Mozilla that favors reliability and performance. It is a statically typed language with type inference, with high-level constructs such as closures, pattern matching, and algebraic data types. At the same time, Rust gives the option to control low-level details, such as memory management, without all the unsafety traditionally associated with it.

Rust does not have a garbage collector and can be configured to exclude the standard library, allowing it to be used for programming microcontrollers, operating systems, and drivers, a domain that has been occupied mainly by C/C++. It avoids the tradeoff between control and safety by statically checking memory correctness at compile-time without introducing any runtime overhead [14].

Rust's advanced type system guarantees memory safety by enforcing ownership and borrowing rules, making Rust a unique programming language that helps to write fast, safe, and reliable software.

The project presented in the thesis, ***pvec-rs***, contributes persistent vectors to the Rust programming language. The following sections are a basic introduction to the relevant parts of Rust touched in this thesis. For a more in-depth overview of the more advanced language features, see the Rust book [10].

1.1.1 Ownership and borrowing

Ownership Ownership is Rust's most unique feature, and it enables Rust to make memory safety guarantees without needing a garbage collector. Every object allocated in Rust always has exactly one *owner*. Ownership can be transferred from one function to another by *moving* the object. Rust's compiler is capable of tracking these movements and identify the location where the object is no longer used, or in Rust's terminology, where it goes out of scope, and generate code for destroying that object. To summarize:

- Each value in Rust has a variable that is called its owner.
- There can only be one owner at a time.
- When the owner goes out of scope, the value is dropped.

Borrowing Borrowing in Rust is the act of creating references to an object. The compiler checks that references always go out of scope before the object they are pointing to, returning an error if it does not, thus guaranteeing that references will never be dangling pointers [14]. The borrowing rules are:

- At any given time, you can have either one mutable reference or any number of immutable references.
- References must always be valid: they cannot outlive an object they are pointing to, and they have to point to an object of a correct type.

Ownership and borrowing rules are demonstrated in Listing 1 taken from [7]. On line 2, a new vector is created and assigned to the `vec` variable. Since `vec` owns the object, it is allowed to mutate it by pushing a value on line 3.

On lines 6 and 7, two references are created with the `&` operator. One of them is borrowed by the `borrow` function, with the `vec` still being the owner. When `borrow` returns, the reference goes out of the scope first, and then the same happens to `vecref` a line later. The two references are a demonstration of the possibility to simultaneously create more than one immutable reference to the same object.

On line 10 a new mutable reference is created with the `&mut` operator. While data is borrowed mutably, no other references can exist. When `borrow_mut`

```
 1 fn main() {
 2     let mut vec = Vec::new();
 3     vec.push(42);
 4
 5     { // Two references are created.
 6         let vecref = &vec;
 7         borrow(&vec);
 8         // ^ Ref goes out of scope when borrow returns
 9     } // vecref goes out of scope here
10
11     borrow_mut(&mut vec); // <- Lend ‘vec‘ mutably to ‘borrow_mut‘
12     take(vec); // <- Ownership transferred to ‘take‘
13     vec.push(37); // <- Error: use of moved value: ‘vec‘
14 }
15
16 fn take(mut my_data: Vec<i32>) {
17     my_data.push(99); // ‘my_data‘ is owner, can perform mutation
18 } // ‘my_data‘ goes out of scope and will be freed here
19
20 fn borrow(vec: &Vec<i32>) {
21     vec.push(10); // error: cannot borrow immutable borrowed ...
22     let element = vec.get(0); // Read is possible.
23 } // Borrowing ends, but ‘vec‘ continues to live in ‘foo‘
24
25 fn borrow_mut(vec: &mut Vec<i32>) {
26     vec.push(0); // Mutable borrow of ‘vec‘, can mutate.
27 }
```

Listing 1: Demonstrating ownership and borrowing rules

returns the reference goes of the scope.

When the function `take` is invoked the ownership to the `vec` object is transferred to the variable `my_data`. The *move* happens because `take` accepts the object by value rather than by a reference. Compiler will error on attempt to use `vec` after it has been moved on line 13.

Unsafe Rust Rust has another language hidden inside it that does not enforce memory safety guarantees: it is called *unsafe* Rust. Rust forces developers to find a safer solution to problematic code or mark it as unsafe and thus highlight the code as a potential source of problems during future debugging.

The ***pvec-rs*** project is designed and implemented without using unsafe Rust features to take advantage of all compile-time checks to minimize the risk of introducing bugs.

1.1.2 Memory management: the stack and the heap

Programming languages such as Java, do not give direct control over the stack and the heap memory, by abstracting them away from a developer. However, systems programming languages like Rust, allow explicit control over memory allocation on the stack or the heap.

By default, Rust allocates objects on the stack. The allocation is local to a function call and is limited in size. The heap allocations, on the other hand, have to be explicitly requested by the developer, and they are virtually unlimited in size and globally accessible [10].

The following section gives an overview of how Rust supports heap allocation.

Smart pointers

Smart pointers are data structures that not only act as a pointer but also have additional metadata and capabilities.

```rust
fn main() {
    let b = Box::new(5); // <- 5 will be stored on the heap.
    println!("b = {}", b);
} // <- Goes out of the scope.
```

Listing 2: Example of using the box pointer

Box The most straightforward smart pointer is `Box<T>` that allows storing data on the heap rather than the stack.

When the box pointer in Listing 2 goes out of the scope on line 4, the corresponding heap allocation gets freed as well.

Rc or reference counting There are cases when a single object might have multiple owners. For example, in graph data structures, multiple edges might point to the same node, and that node is conceptually owned by all of the edges that point to it. A node should not be cleaned up unless it does not have any edges pointing to it [10].

```rust
fn main() {
    let mut rc = Rc::new(Vec::new());
    // ^ Wrapping the vec instance into rc pointer

    Rc::make_mut(&mut rc).push(42);
    // ^ Ref count is 1, push succeeds without cloning.

    take(rc.clone()); // <- Ref count is incremented to 2.
    Rc::make_mut(&mut rc).push(37); // <- Updating the original 'vec'.
} // <- 'rc' is decremented to 0 and dropped

fn take(mut my_rc: Rc<Vec<i32>>) {
    println!("val={:?}", my_rc.get(0));
    // ^ Reading values does not cause clone

    Rc::make_mut(&mut my_rc).push(99);
    // ^ Mutating the object with ref count of 2
    // will clone the wrapped value and then update it.
} // ‘my_rc' is decremented to 0 and dropped.
```

Listing 3: Example of using the reference counting pointer

To enable multiple ownership, Rust has a type called `Rc<T>`, which is an abbreviation for reference counting. The `Rc<T>` type keeps track of the number of references to a value that determines whether or not a value is still in use. If there are zero references to a value, the value can be cleaned up without any references becoming invalid.

Listing 3 demonstrates how multiple ownership works using the `Rc<T>` pointer. On line 2, a vector is created and wrapped into reference counting pointer immediately. On the first call to `Rc::make_mut` method, the number 42 is pushed onto the vector.

The `rc` variable is cloned before it is passed to `take` by value, causing the reference count to be incremented to 2. It is important to note that the underlying vector object is not cloned until line 16, where `Rc::make_mut` function clones the inner value to ensure unique ownership. When `take` is finished, both `my_rc` and the newly copied vector go out of the scope and destroyed.

The execution continues on line 9, where `main` proceeds updating the vector. In this case, `Rc::make_mut` is not causing a clone because the underlying vector has only a single reference to it.

Note that `Rc<T>` is only for use in single-threaded scenarios. `Arc<T>` that stands for atomic reference counting should be used in a multi-threaded environment instead.

1.1.3 Enums, structs and traits

Structs

```rust
struct Book {
    author: String,
    name: String
}
```

Listing 4: A basic Rust struct

Structs are the most common way to declare complex custom data types in Rust. A *struct*, or *structure*, is a custom data type that lets you name and package together multiple related values[2].

Enums

[2]https://doc.rust-lang.org/book/ch05-00-structs.html

```rust
struct Point(isize, isize);

enum Shape {
    Rectangle {
        p_1: Point,
        p_2: Point
    },
    Triangle {
        p_1: Point,
        p_2: Point,
        p_3: Point
    }
};
```

Listing 5: A basic Rust enum

Enums or *enumerations*, are used to define a custom data type by enumerating its possible variants, where each variant might have different data associated with it. The size of an enum instance is equal to the maximum size of its variants. For example, `Shape::Rectangle` will be allocated the same amount of memory as `Shape::Triangle`, even though it contains only two points.

Traits

A *trait* is a language feature used to define a set of methods that implement behavior. Moreover, they can be used to reuse behavior without reusing state [16]. A distinct ability of traits is that they can be implemented outside of the type declaration, including types that belong to other modules. Traits support default function implementations, as well as inheritance. Listing 6 shows how traits can be used to define the common contract between several data types.

Drop *Dropping* is the process of cleaning-up a value that goes out of the scope in a Rust program. By default, the compiler generates code that automatically drops the value, but there is also a way to override the default behavior by implementing the `Drop` trait. The `Drop` trait requires to implement one method named drop that takes a mutable reference to self.

```rust
struct Email(String);
struct Tweet(String);

trait Message {
    fn send();
}

impl Message for Email {
    fn send() { /* sending email */ }
}

impl Message for Tweet {
    fn send() { /* sending tweet */ }
}
```

Listing 6: A basic Rust trait

1.1.4 Cargo and crates

Cargo is the package manager for Rust. It manages Rust dependencies by distributing, downloading, and compiling packages.

Rust packages are called *crates*. The package registry of the Rust community is located at https://crates.io/.

Every *crate*, including ***pvec-rs***, has a *Cargo.toml* configuration file that contains feature flags, dependencies and their versions, and the crate metadata such as name, version, and description.

1.2 Persistent data structures

Ephemeral data structures are standard data structures that do not keep the history of their versions. Once an ephemeral data structure is modified, there is no mechanism to go back to previous states. This behavior is typical for collections provided by the standard library of modern general-purpose programming languages.

Persistent data structures, on the other hand, are set up in a different way to allow access to any version, old or new, at any time [5]. Persistent data structures were adopted in functional programming[3] languages such as Scala[4] and Clojure[5].

1.2.1 Persistence categories

Persistent data structures are categorized based on the operations which they offer over their versions:

- *Partial persistence* — Read-only access to any previous version of the data structure with the ability to update only the newest one. The versions are ordered linearly.
- *Full persistence* — Read and write operations are available for all versions. The versions are structured as a tree.
- *Confluent persistence* — Full persistence with the ability to merge several previous versions into a single new one. The versions form a directed acyclic graph [6].
- *Functional persistence* — This model takes its name from functional programming, where objects are immutable. In comparison to the previous models, it prohibits change of the internal representation of the data structure [13].

1.2.2 Achieving persistence

While persistence can be achieved by copying, the cost of write operations quickly becomes unacceptable. Functional programming languages that have immutable data structures by design were at the forefront of the research for a more efficient alternative.

Multiple data structure designs were proposed, often offering good performance for particular operations with the focus on particular use cases. For example, a

[3]Functional programming is a programming paradigm where programs are constructed by applying and composing functions.
[4]https://www.scala-lang.org/
[5]https://clojure.org/

singly linked list guarantees $\mathcal{O}(1)$ performance for adding or removing elements at the head, but suffers from the worst case $\mathcal{O}(n)$ random access operations.

1.2.3 Tries

Since most write operations modify only some parts of a data structure, a complete copy is often unnecessary. A better approach is to use the similarity between the new and old versions by *sharing* structure between them. For example, instead of using a single memory block to represent a data structure, it can be split into smaller pieces or *nodes* linked together as a *tree*. Since modifications only apply to some nodes, the rest of them remain unchanged and can be shared without copying.

Inspired by Bagwell's paper Ideal Hash Trees [1], Rich Hickey pioneered the first persistent vector to offer uniformly good performance across different operations comparable to mutable vectors for the Clojure programming language. Clojure's vectors are *fully* persistent, as they allow us to read and write to all versions.

A persistent vector is a *bit-array mapped trie*, or simply a "wide" tree with a high branching factor. Thus, tries[6] are very shallow, being at most 7 levels deep when the branching factor is equal to 32, offering effectively constant time for almost all operations. Wide nodes and shallow height help to improve cache locality and to reduce cache misses.

Later, Bagwell and Rompf introduced *relaxed radix balanced trees* based on the design of Clojure's persistent vectors [2]. RRB-trees offer efficient concatenation and splitting, enabling vectors to be suitable data structures for parallel processing. Later it became a foundation for parallel vectors in Scala's standard library [17].

The RRB-tree data structure is used to implement persistent vectors in ***pvec-rs*** and is explored further in Section 2.2.

[6]Generally, a trie is a kind of a tree data structure used to store key-value pairs where keys are usually strings.

1.3 Goals and contributions

The purpose of this thesis is to explore and evaluate novel ideas proposed by Nicholas Matsakis for optimizations that emerge at the intersection of the unique Rust features and persistent data structures to contribute a persistent vector implementation with excellent performance and idiomatic Rust interface.

The thesis makes the following contributions:

- A persistent vector implementation based on RRB-tree with efficient concatenation and splitting to make it a reasonable data structure for parallel programming.
- An extension for the data parallelism library Rayon, combined with persistent vectors for creating an efficient implementation of general-purpose parallel vectors.
- Describing and implementing novel optimizations specific to Rust, specifically *dynamic representation* and *unique access* optimizations.
- An extensive performance evaluation of the contributed persistent and parallel vectors, as well as the effectiveness of the proposed optimizations.

Section 2.1 gives an overview of the RRB-tree data structure that serves as the foundation for the persistent vector. In Chapter 3, the ***pvec-rs*** project, its interface, and optimizations specific to Rust are presented in detail. Chapter 4 focuses on the performance evaluation of the persistent vector followed by a chapter with results.

Chapter 2

Background

This chapter gives an introduction to the *radix balanced tree* and *relaxed radix balanced tree* data structures that are used as the foundation for persistent vectors:

- First, we will take a look at *radix balanced trees*, their organization, and algorithms such as *radix search* and *path copying* that are used at the core of all vector operations.
- Then we will proceed to an extension — *relaxed radix balanced tree*, which enables efficient concatenation and splitting.
- Finally, we discuss the concept of *transience* and how it can be used to improve the performance of persistent data structures.

2.1 Radix balanced tree

Radix balanced trees or RB-trees are m-ary trees that use integers as keys to find values. The data structure was pioneered by Rich Hickey as the foundation for the persistent vector implementation in Clojure [9].

RB-trees consist of nodes that reference subtrees or values. We will be referring to the former and latter types of nodes as *branch* nodes and *leaves* correspondingly. The number of subtrees and values in the node is configurable and is defined as m or the *branching factor*. The branching factor can be any number that is a power of 2, allowing an efficient radix search implementation.

The higher the value of m, the wider and shallower the tree will be for the given number of elements. From now on the height of a tree will be referred to as h, where h_{max} is the upper boundary:

$$h_{max} = log_m(n) \tag{2.1}$$

When m is a large value, for example 32, the tree becomes shallow, and the complexity of accessing values by traversing the tree from the root to a leaf becomes *effectively* constant. For example, if the maximum value of 32 bit signed integer is substituted for n in Equation (2.1) then h_{max} will never exceed 7 levels. Thus, due to its strong performance guarantees, RB-tree serves as a solid foundation for a general-purpose persistent vector implementation.

In the following sections, we will take a look at RB-tree algorithms used to implement vector operations[1].

2.1.1 Radix search

The algorithm used to lookup values in RB-tree is called *radix search*. It serves as a foundation for other operations that involve the tree traversal, such as push, pop, and update.

Bit partitioning

Radix search accepts an integer key as an argument. It can be thought of as a composite key, where each subkey is a sequence of bits. Conceptually, the idea is to divide the key into bit blocks, where each block is an index, specific to the tree level.

As demonstrated in Equation (2.2), the bit block size can be calculated from the branching factor. It will be referred to as x or *bits per level*. When the branching factor m is 16, x is 4, meaning that the subkey size is equal to 4 bits.

[1]For a formal definition of RB-tree, refer to [11]

$$x = log_2(m) \tag{2.2}$$

Extracting subkeys

The number of subkeys within the search key depends on the number of tree levels or h. For example, a search key addressing an element of the tree of $h = 3$ and $x = 2$ will consist of 3 subkeys taking up 6 bits of space in total.

Subkeys are arranged in the order from the most to the least significant bits, where the most significant block of bits is a key used to access the child node of the root. Each following key is used to find a child node on the corresponding tree level.

Let us consider an example of processing a key that addresses a value in the tree of $h = 3$, $m = 4$ and $x = 2$.

$$54_{10} = 00110110_2 \tag{2.3}$$

Since the tree height is three, we have three corresponding subkeys: 11_2, 01_2, and 10_2. Let us assume that we are interested in extracting the subkey for the child node on the second level that corresponds to the 01_2 bits.

The first step is to get rid of the bits following the subkey of our interest. The logical right shift operation[2] denoted as $\ggg$, will push the specified number of 0s into a key k, where l is the *level* at which current node is located:

$$k \ggg ((l - 1) \cdot x). \tag{2.4}$$

Since the node from the example is located at $l = 2$ in the tree of $x = 2$, the key k is shifted by 2 bits. The result of the operation is 00001101_2. As a result, the "tail" of the key is truncated. The next step is to remove bits preceding the subkey by masking them to 0 using the bitwise "and" operator.

[2]A logical right shift is a bitwise operation that shifts all the bits of its first operand to the right by number of bits specified in the second operand.

A bitwise "and" takes two equal-length binary representations and executes the "and" operation for each pair of the corresponding bits. If both bits in the compared position are 1, the bit in the resulting binary representation is 1; otherwise, the result is 0. Here is an example, where the first and the second operands are the key and mask respectively:

$$00001101 \,\&\, 00000011 = 00000001. \tag{2.5}$$

Only the last two bits of the mask are set to 1, meaning that all bits of the key except the last two will be masked to 0. The result of the "and"-ing operation will be the value of the subkey.

A mask is of the integer type and is calculated from the branching factor m:

$$mask = m - 1. \tag{2.6}$$

If m is equal to 4, the maximum subkey value will be 3, which equals to 00000011_2.

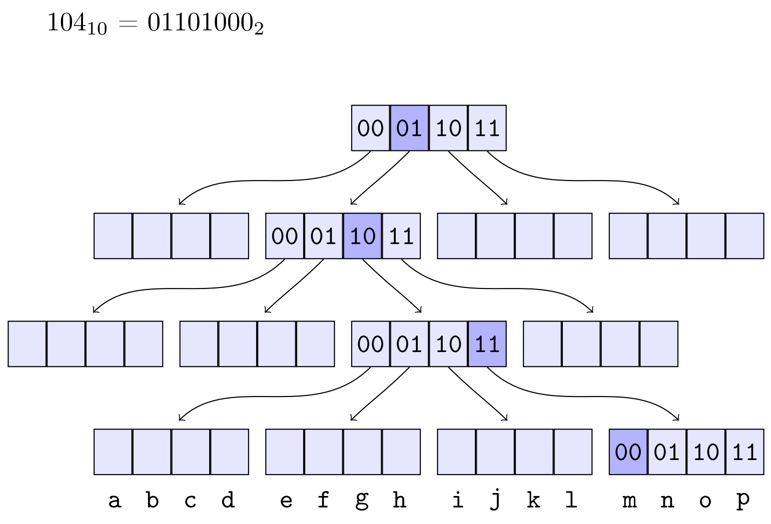


Figure 2.1: Visualization of the radix search algorithm

Figure 2.1 is an illustration of how radix search works. There is only a fraction of the tree visualized in the example rendering values in the [92, 107] index range.

The branching factor of the tree is 4, resulting in the subkey size of 2 bits. The mask is equal to 3 or 00000011 in binary representation. For simplicity, the index type is selected to be *unsigned byte* with a maximum capacity of 256.

The tree height in Figure 2.1 is 4. The goal is to lookup the value at the index 104, which is equal to 01101000_2. Listing 7 outlines the radix search algorithm.

The search starts by initializing the node and level variables to root and $height - 1$ correspondingly. The for loop runs until the leaf node level is reached, where each step towards it involves selecting a child node by using bit shifting and masking. After exiting the loop, the function returns the value from the leaf node. The runtime complexity of radix search is $\mathcal{O}(log_m(n))$.

```
function RadixSearch(root, key)
    node ← root
    for level ← $root_{height}$ - 1, 1 do
        index ← (key ⋙ (level · x)) & mask
        node ← node[index]
    index ← key & mask
    return node[index]
```

Listing 7: Radix search algorithm

2.1.2 Update

The persistent *update* operation returns a new tree version that includes an updated value, instead of mutating the original instance in-place as with *ephemeral* data structures.

Path copying

When updating a RB-tree, the radix search algorithm is used to build a path to the value. In order to avoid mutating the original tree instance, each node on the path is copied. This process is called as *path copying* [15]. It is important to emphasize that nodes that are not a part of the path are reused.

The update operation performs the h copies of m sized nodes, where h is the maximum height of the tree. As described in Equation (2.1), h is bound by $\mathcal{O}(log_m(n))$, which results in $\mathcal{O}(m \cdot log_m(n))$ complexity of the update operation.

For large branching factors m such as 32, the performance becomes effectively $\mathcal{O}(1)$. For example, if the tree is full, where the number of all elements n is bound by the maximum value of 32 bit integer[3], h will be at most 7.

```
function UPDATE(root, key, value)
    newRoot ← CLONE(root)
    node ← newRoot
    for level ← $\mathrm{root_h}$ - 1, 1 do
        index ← (key ⋙ (level · x)) & mask
        newChildNode ← CLONE(node[index])
        node[index] ← newChildNode
        node ← newChildNode
    index ← key & mask
    node[index] ← value
    return newRoot
```

Listing 8: Path copying algorithm for RB-tree

From an implementation perspective, the difference between update and radix search is that every visited node, including root, must be copied. The return value is the root node for the new version of the tree.

2.1.3 Push

The push operation is used to add new values to the end of an RB-tree. It employs *path copying* to preserve the original tree without changes. It accepts the root node and new value as arguments and returns a new version of the tree.

The RB-tree size is used as the search key. For example, if the tree size is 9, a key for the new value will be 9. The size is incremented after the operation completes.

[3] 2147483647

Since the push operation adds new values, it has to manage the tree capacity. Generally, there are two cases that require special handling:

- The case when a path to a leaf node includes branch nodes that are not created yet. The solution is to generate missing nodes while descending the tree.
- The situation when the tree size exceeds its *capacity*[4] is known as *root overflow*. It can be solved by adding a new level to the tree by creating a new root node and setting the old one as the first child of the new root.

The complexity of the operation is $\mathcal{O}(m \cdot log_m(n))$ as it traverses and creates new nodes from the root to a leaf.

Listing 9 demonstrates how the tree capacity is calculated and managed.

2.1.4 Pop

The purpose of pop is to remove values from the end of an RB-tree. It accepts root as input and returns both a removed value and a new version of a structure.

Pop is responsible for reducing the capacity of RB-trees when needed, including the removal of empty branches and leaves. Similar to the update and the push operations, it relies on the path copying algorithm to avoid modifying the existing version of the tree.

Since RB-tree is a complete[5] tree, all entries to the right of *NIL* must be absent as well. As shown in Listing 10, checking if the first entry is *NIL* is sufficient. If *NIL*, the empty child will be replaced with a *NIL* reference in the parent node.

A root is redundant if it contains only a single child node, the case if the second entry is *NIL*. The original root is *demoted* by replacing it with the first child node, which becomes the new root.

[4]Capacity is the maximum number of elements a tree can accommodate.

[5]A complete m-ary tree is a tree that must be filled on every level except for the last one. If the last level is partially filled, all nodes of the tree must be "as far left as possible".

```
function CAPACITY(height)
    return m $\ll$ ((height - 1) $\cdot$ x)

function PUSH(root, value)
    newRoot $\leftarrow$ NIL
    if CAPACITY(root$_{height}$) $<=$ root$_{size}$ then
        newRoot $\leftarrow$ CREATENODE
        newRoot[0] $\leftarrow$ CLONE(root)
        newRoot$_{height}$ $\leftarrow$ root$_{height}$ $+ 1$
    else
        newRoot $\leftarrow$ CLONE(root)
    node $\leftarrow$ newRoot
    key $\leftarrow$ newRoot$_{size}$
    for level $\leftarrow$ newRoot$_{height}$ - 1, 1 do
        index $\leftarrow$ (key $\ggg$ (level $\cdot$ x)) & mask
        childNode $\leftarrow$ node[index]
        newChildNode $\leftarrow$ NIL
        if childNode $=$ NIL then
            newChildNode $\leftarrow$ CREATENODE
        else
            newChildNode $\leftarrow$ CLONE(childNode)
        node[index] $\leftarrow$ newChildNode
        node $\leftarrow$ newChildNode
    index $\leftarrow$ key & mask
    node[index] $\leftarrow$ value
    newRoot$_{size}$ $\leftarrow$ newRoot$_{size}$ $+ 1$
    return newRoot
```

Listing 9: Pseudocode for the RB-tree's push operation

The pop operation traverses a tree and creates m new nodes. Hence, its complexity is $\mathcal{O}(m \cdot log_m(n))$.

2.2 Relaxed radix balanced tree

Relaxed radix balanced tree or RRB-tree is a *confluently* persistent data structure that extends RB-tree to support concatenation and splitting in $\mathcal{O}(log(n))$

```
function PopNode(node, key)
    newNode ← Clone(node)
    value ← $NIL$
    if $node_{height} = 0$ then
        index ← key & mask
        value ← newNode[index]
        newNode[index] ← $NIL$
    else
        index ← (key $\ggg$ ($newNode_{height} \cdot x$)) & mask
        value, childNode ← PopNode(newNode[index], key)
        newNode[index] ← childNode
    if newNode[0] = $NIL$ then
        return value, $NIL$
    else
        return value, newNode

function Pop(root)
    value, newRoot ← PopNode(root, $root_{size}$ - 1)
    if newRoot[1] = $NIL$) then
        newRoot ← newRoot[0]
    return value, newRoot
```

Listing 10: Pseudocode for the RB-tree's pop operation

rather than linear time without compromising the performance of other operations.

An invariant that RB-tree maintains is that all nodes except the right-most ones have to be full. This enables a simple radix search implementation, but on the other hand, makes efficient sub-linear concatenation impossible.

RRB-trees relax this constraint by allowing nodes to be partially full, and introduce a rebalancing algorithm to ensure that the tree height does not exceed the $\mathcal{O}(log(n))$ bound to keep performance guarantees for other operations.

In this section, we will look at how RRB-trees work, specifically the concatenation algorithm and the relaxed variant of the radix search. For details of the implementation of the split, relaxed push, and pop operations, please refer to the project source code[6] or to [11].

2.2.1 Relaxed radix search

With relaxed RRB-tree constraints, there is no efficient way to calculate the size of the subtree without additional metadata. Hence, the *sizes* array is introduced to keep track of the subtree size. The *sizes* array is m elements long, where each value represents the subtree size at the corresponding index. The sizes table values are re-calculated when the subtree is modified, for example, when concatenating or pushing new values.

When the radix search encounters a relaxed node, it compares the entire search key against entries in the size table. A subtree contains the desired value if the corresponding size table entry is bigger than or equal to the search key. Before descending into a subtree and repeating the search step, the search key is subtracted the size of the subtree.

When a balanced node is encountered, the search process falls back to the balanced version of the radix search algorithm Section 2.1.1.

Since the tree traversal involves scanning an m long *sizes* array for every node from the root to a leaf, the complexity of relaxed radix search is $\mathcal{O}(m \cdot log_m(n))$. In theory, the FINDINDEX function in Listing 11 can be replaced with binary search resulting in slightly better $\mathcal{O}(log_2(m) \cdot log_m(n))$ performance. However, it is nearly impossible to measure the difference for such small sized arrays, as linear scan can be as fast if not faster due to its simplicity and CPU cache locality[7].

2.2.2 Concatenation

The concatenation algorithm used in this project is from [17], which produces a slightly more balanced tree than the initially proposed version from [2]. It achieves it by allowing the relaxed tree nodes to have m children instead of $\mathrm{m} - 1$.

[6]https://github.com/arazabishov/pvec-rs
[7]https://dirtyhandscoding.wordpress.com/2017/08/25/performance-comparison-linear-search-vs-binary-search/

```
function FindIndex(sizes, idx)
    candidate ←0
    if candidate < $m$ - 1 and sizes[candidate] <= idx then
        candidate++
    return candidate

function RelaxedRadixSearch(root, key)
    node ← root
    idx ← key
    for level ← $root_{height}$ - 1, 1 do
        if $node_{sizes}$ = NIL then
            index ← (key ⋙ (level · x)) & mask
            node ← node[index]
        else
            sizes ← $node_{sizes}$
            index ← FindIndex(sizes, idx)
            node ← node[index]
            if index != 0 then
                idx ← idx - sizes[index - 1]
    index ← idx & mask
    return node[index]
```

Listing 11: Pseudocode of relaxed radix search

The algorithm consists of two stages: descending the tree, and then, merging and rebalancing nodes. The time complexity of the presented concatenation algorithm is $\mathcal{O}(m^2 \cdot log_m(n))$.

The recursive function in Listing 12 descends the tree by selecting the rightmost node of the left tree and the leftmost node of the right tree. If one of the trees is taller than the other, the function descends into a taller tree only until nodes of both trees are at the same level.

When the bottom level with leaf nodes is reached, the function stops descending and starts merging and rebalancing nodes to ensure the $\mathcal{O}(log_m(n))$ bound on the tree height.

The Rebalance function accepts three lists of nodes as arguments. The left and right lists constitute all nodes of both trees at the given level except two:

```
function CONCAT(leftNode, rightNode)
    if $\text{leftNode}_{\text{height}} > \text{rightNode}_{\text{height}}$ then
        mergedNode ← CONCAT($\text{leftNode}_{\text{last}}$, rightNode)
        return REBALANCE($\text{leftNode}_{\text{init}}$, mergedNode, NIL)
    else if $\text{leftNode}_{\text{height}} < \text{rightNode}_{\text{height}}$ then
        mergedNode ← CONCAT(leftNode, $\text{rightNode}_{\text{first}}$)
        return REBALANCE(NIL, mergedNode, $\text{rightNode}_{\text{tail}}$)
    else
        mergedNode ← NIL
        if $\text{leftNode}_{\text{height}} = 0$ then
            mergedNode ← CONCAT(leftNode, rightNode)
        else
            if $\text{leftNode}_{\text{height}} = 1$ then
                mergedNode ← CONCAT($\text{leftNode}_{\text{last}}$, $\text{rightNode}_{\text{first}}$)
            else
                mergedNode ← CONCAT($\text{leftNode}_{\text{last}}$, $\text{rightNode}_{\text{first}}$)
        return REBALANCE($\text{leftNode}_{\text{init}}$, mergedNode, $\text{rightNode}_{\text{tail}}$)
```

Listing 12: Concatenation algorithm of RRB-tree

the rightmost node of the left tree and the leftmost node of the right tree. Those two nodes are already rebalanced and passed as the middle argument. Three lists are concatenated together into a single merged list.

The goal of rebalancing is to arrange the children of merged nodes in such a way that all nodes except the rightmost branch are filled with values. When the REBALANCE function completes re-arranging nodes, it returns a new branch containing rebalanced nodes. Pseudocode for rebalancing can be found in Listing 23.

The rebalancing process is illustrated in Figures 2.2–2.5. Note, the presented figures exclude parts of trees that are not important for conveying the idea to preserve space.

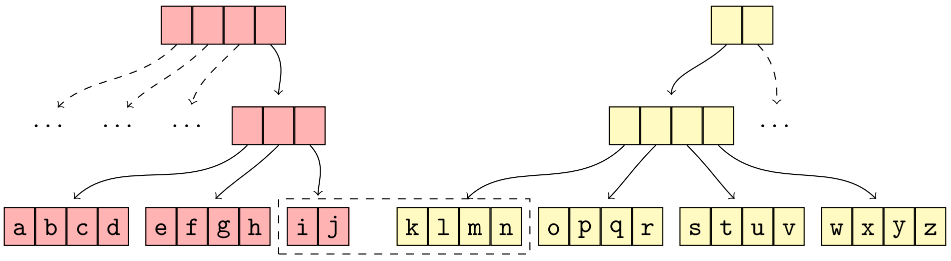


Figure 2.2: Illustration of the rebalancing algorithm at level 0

Concatenation starts at the bottom of the tree by merging the leaf nodes. The result is a new rebalanced branch node with leaves that will be used when rebalancing nodes at level 1.

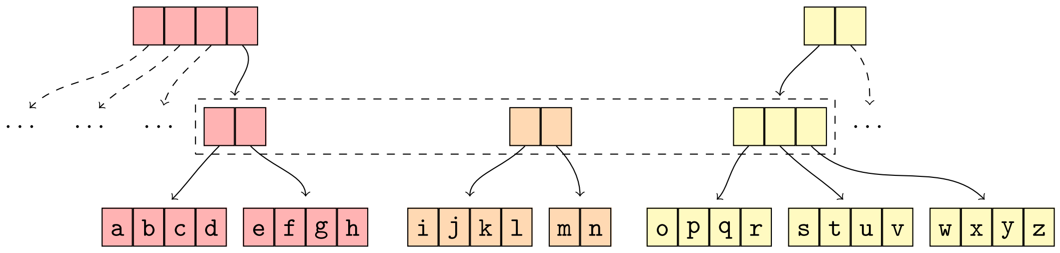


Figure 2.3: Illustration of the rebalancing algorithm at level 1

Rebalancing at level 1 involves the left and right branch nodes together with the newly created branch node from the previous step. The algorithm avoids processing nodes where rebalancing is not beneficial. For example, in Figure 2.4 where the *full* nodes from the left-hand side tree are reused without rebalancing them.

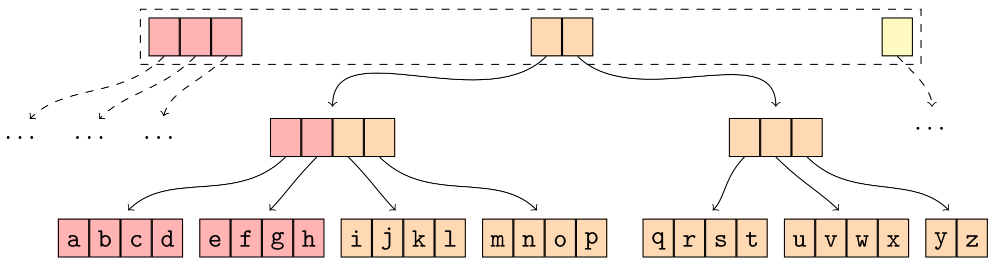


Figure 2.4: Illustration of the rebalancing algorithm at level 2

Figure 2.5 is the last step that rebalances the top-level nodes of the left and right trees, producing a new root. If the root contains only a single child, its child will be promoted to be the root to avoid unnecessary overhead.

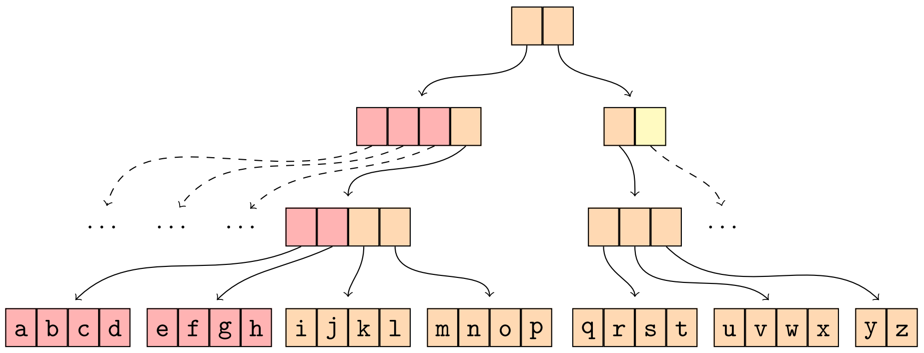


Figure 2.5: Illustration of the rebalancing algorithm: a rebalanced tree

The formal description and analysis of concatenation algorithm and its implementation is thoroughly presented in [11].

2.3 Transience

When modified, conventional persistent data structures always return a new version that includes the change. It is a reliable and straightforward way to ensure that existing versions are not changed, a behavior that is essential to robust programs.

However, there are scenarios when persistence is not useful, such as when working with a collection in isolated environment like a pure function. For example, building a new vector using the *push* operation creates a new version on every call, immediately dismissing the previous one.

Even though persistent data structures minimize the cost of creating new versions by *path copying*, memory allocations are still very resource consuming. Thus, *transient* data types were introduced to the Clojure programming language to solve this problem.

Transience is an optimization that allows us to update persistent data structures in place without creating new versions in performance-critical code. Once a transient data structure is constructed, it can be converted back to persistent and safely shared.

```clojure
(defn vecpersistent [n]                 ;; uses a persistent vector
    (loop [i 0 v []]
        (if (< i n)
        (recur (inc i) (conj v i))
        v)))

(defn vectransient [n]                  ;; uses a transient vector
    (loop [i 0 v (transient [])]
        (if (< i n)
        (recur (inc i) (conj! v i))
        (persistent! v))))
```

Listing 13: An example of working with transient and persistent vectors in Clojure

As demonstrated in Listing 13, working with transient types requires special syntax such as `transient`, `conj!`, and `persistent!`[8].

[8]https://clojure.org/reference/transients

2.3.1 Guarantees and limitations

Clojure ensures that transience does not violate the guarantees of persistent data structures by enforcing the following rules:

- A transient vector created from a persistent one does not modify the original version. The cost of creating a transient instance is $\mathcal{O}(1)$, as well as transforming it back to persistent.
- Transients are confined to the thread they are created on, effectively forbidding sharing between threads and eliminating the possibility of race conditions.
- Transients are not allowed to be used after converting them back to persistent to avoid indirectly modifying a persistent version.

Chapter 3

Persistent vectors in Rust

The purpose of this chapter is twofold: to present the ***pvec-rs*** project and to discuss its unique design features in the context of Rust. First, we will take a look at the project structure, the types offered by the library, and how it can be configured.

Then, we will look at the design and optimizations of persistent vectors in Rust and their specifics:

- First, we will take a look at the memory layout of `RrbTree` and how it supports the implementation of cloning and path copying.
- Then, transience and dynamic representation optimizations will be introduced in the context of Rust's ownership and borrowing rules.
- At last, we will talk about thread-safety and the requirements for sharing objects between threads.

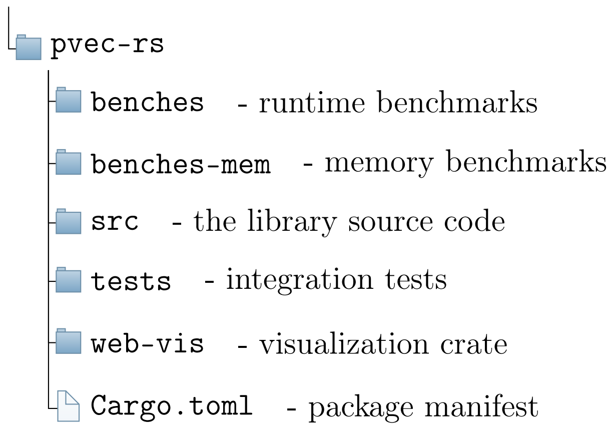
3.1 Presentation of the project

The ***pvec-rs*** project consists of three crates:

- The library crate itself with the persistent vector implementation.

- ***web-vis***: a crate with web application for visualization of the data structure in the browser.
- ***benches-mem***: The benchmarking crate for measuring the memory footprint of persistent vectors. Its usage is described in detail in Chapter 4.

The structure of the library is demonstrated in Figure 3.1. The `pvec-rs` directory is the root of the library crate. The definition of `RrbTree`, `RbVec`, `RrbVec`, and `PVec` are all located under the `src` folder. The `benches` and `tests` folders are for runtime benchmarks and integration tests correspondingly.



```
pvec-rs
├ benches      - runtime benchmarks
├ benches-mem  - memory benchmarks
├ src          - the library source code
├ tests        - integration tests
├ web-vis      - visualization crate
└ Cargo.toml   - package manifest
```

Figure 3.1: The folder structure of ***pvec-rs***

3.1.1 Configurability

The ***pvec-rs*** project can be configured using the feature flags described below:

`small_branch`	Sets the branching factor used by `RrbTree` to 4.
`arc`	Selecting the thread-safe version of the `Rc` pointer.
`rayon_iter`	if specified together with the `arc` flag, provides the parallel iterator implementation for Rayon.
`serde_serializer`	***pvec-rs*** provides the JSON[1] serializer if specified.

Table 3.1: Feature flags for the library configuration

By default, the branching factor used by the tree-based vector types is equal to 32. If `small_branch` flag is specified, the branching factor will be 4. This flag is useful for integration testing to reveal issues with the re-balancing algorithm earlier as the tree ends-up being significantly taller compared to using the branching factor of 32.

To compile persistent vectors for use in multi-threaded environment Rust requires to implement `Send` and `Sync` traits. When compiled with the `arc` feature flag, those traits will be automatically implemented for all vector types enabling multi-threaded use cases.

To access the ***rayon***'s parallel iterator implementations, the library user has to use both the `arc` and `rayon_iter` feature flags when including the library.

3.1.2 The library APIs

The ***pvec-rs*** project is a Rust library crate that offers several persistent vector implementations listed below:

`RbVec`	A persistent vector based on the balanced `RrbTree`.
`RrbVec`	The `RrbTree`-based persistent vector implementation with efficient concatenation and splitting. Its underlying `RrbTree` instance remains balanced unless concatenated or splitted, after which it becomes relaxed.
`PVec`	A vector that starts as the standard and switches to `RrbVec` when cloned.

Table 3.2: Vector implementations provided by ***pvec-rs***

`RbVec` and `RrbVec` are based on `RrbTree`. Both types implement the *tail*[2] optimization and offer effectively constant `push`, `pop`, `get`, and `get_mut` operations, with the difference that `RrbVec` offers efficient `append` and `split_off`. `PVec`, on the other hand, inherits performance properties of the underlying representation that is either `Vec` or `RrbVec`.

All vectors provide methods with identical method signatures with the difference in how much each operation costs. The table below lists available

[2]See Appendix B for details on the *tail* optimization.

functions and their complexity:

A vector function	`Vec`	`RbVec`	`RrbVec`
`fn push(&mut self, e: T)` Appends an element to the back of a collection	1	$log_m(n)$	$log_m(n)$
`fn pop(&mut self) -> Option<T>` Returns the last removed value or `None` if vector is empty.	1	$m \cdot log_m(n)$	$m \cdot log_m(n)$
`fn get(&self, i: usize) -> Option<&T>` Returns a reference to an element.	1	$log_m(n)$	$log_m(n)$
`fn get_mut(&mut self, i: usize) -> Option<&mut T>` Returns a mutable reference to an element.	1	$m \cdot log_m(n)$	$m \cdot log_m(n)$
`fn append(&mut self, v: &mut Vec<T>)` Concatenates two vectors by moving all values of `v` into `self`, leaving other empty.	n	n	$m^2 \cdot log_m(n)$
`fn split_off(&mut self, at: usize) -> Vec<T>` Splits the collection into two at the given index.	n	n	$m \cdot log_m(n)$
`fn len(&self) -> usize` Returns the number of elements in the vector.	1	1	1
`fn is_empty(&self) -> bool` Returns `true` if the vector contains no elements.	1	1	1
`fn into_iter(self) -> Iterator` Creates an iterator from `self`.	n	n	n
`fn into_par_iter(self) -> IntoParallelIterator` Converts `self` into a parallel iterator. Note, it requires the `arc` and `rayon_iter` feature flags.	n	n	n

Table 3.3: A table of methods supported by persistent vectors

The complexity of operations is expressed using the Big $\mathcal{O}$ notation. The $\mathcal{O}(1)$ value was substituted by 1 for brevity.

The characteristics of the `RbVec` and `RrbVec` operations are effectively constant given that $m = 32$, and the input size n is bound by the maximum capacity of the 32-bit integer resulting in $log_m(n) < 7$.

Core and bulk operations Operations that interact with a single value, such as push, pop, and get, will be referred to as ***core*** operations. Operations that work with multiple values will be referred to as ***bulk*** operations, such as append, split, and iterators.

Iterators

Iterators[3] are a powerful alternative to *for* loops that offer a variety of operators for processing collection elements.

The foundation of iterators for ***pvec-rs*** vectors is the `RrbTree` iterator implementation that presents a tree as a collection of leaf nodes. Instead of reading individual values using the radix search, the tree iterator returns the entire leaf node. Vector iterators cache the node and read values from it until it is exhausted, after which they request the next node from the tree iterator, minimizing the number of the tree-traversals and improving performance. This optimization can be thought of as the generalization of the tail optimization described in Appendix B.

Parallel iterators

Parallel iterators process collection elements on multiple threads. The parallel iterator implementation for ***pvec-rs*** is based on the data parallelism framework ***rayon***, and can be enabled using the `arc` and `rayon_iter` feature flags. Compared to sequential iterators, the parallel ones do not guarantee the deterministic order of processing values.

The library documentation can be found at: https://docs.rs/pvec/.

3.2 Memory layout

To define persistent vectors in Rust, we first need to understand how different parts of the data structure can be represented in the computer memory, and which Rust constructs are the most suitable for this purpose.

[3] https://doc.rust-lang.org/std/iter/trait.Iterator.html

The foundation of a confluently persistent vector is `RrbTree`, with additional fields such as a tail, size, and height. `RrbTree` consists of infinitely nested nodes, which form a directed acyclic graph[4] in memory.

As the persistent vector can be arbitrarily large, the Rust compiler is not able to measure its size during compilation. Hence, the memory allocation on the stack is not possible without additional constraints, such as the fixed vector capacity.

In fact, the Rust compiler will abort the compilation if a *recursive data type*[5] definition is encountered. A recursive data type is a type that contains itself, such as a tree node. The solution is to use dynamic memory allocation instead of static. Rust offers a particular type of pointers for this purpose known as smart pointers[6], such as `Box`, `Rc`, etc.

As `RrbTree` employs structural sharing, several tree instances might point to the same sub-tree. In other words, one node can be referenced by several parent nodes simultaneously. Even though `Box` enables recursive types, it does not allow shared ownership of the underlying value. A smart pointer that supports a notion of shared ownership is known as `Rc` or reference counting pointer.

3.2.1 Reference counting pointers

In a nutshell, a reference counting pointer allows shared ownership of the object wrapped into it. Every time a potential owner needs a reference to the value, the pointer itself is cloned instead of the underlying object. The reference count is incremented on each clone, and decremented when a pointer goes out of the scope. If the reference count reaches zero, the underlying value is destroyed.

[4]A directed acyclic graph is a data structure that consists of nodes connected with directed edges, in which moving from node to node by following edges will never lead to the same node again.

[5]https://doc.rust-lang.org/book/ch15-01-box.html#enabling-recursive-types-with-boxes

[6]See Section 1.1.2 for more details.

Copy-on-write semantics

Rust's `Rc` conforms to the ownership and borrowing rules that are enforced during runtime rather than compile-time. It allows shared immutable access to the value as well as regular reference does, and permits unique mutable access only if other references do not exist.

The `Rc::make_mut` method allows us to safely acquire a mutable pointer to the value regardless of the reference count. If there are no other pointers to the value, then `Rc::make_mut` immediately returns a mutable reference. Otherwise, it clones the inner value to a new allocation to ensure unique ownership and then returns a reference to it. This behavior is known as *copy-on-write*.

Path copying

When updating the tree, the path leading from the root node to the affected leaf is copied to preserve the original tree from changes. The copy-on-write semantics of `Rc` is the foundation for the path copying algorithm implementation in `RrbTree`.

Rust permits updating objects only through mutable references. To update a value in the tree, one has to acquire a mutable reference to each node that forms a path to the value. Since nodes are decorated with `Rc` pointer, a safe way to acquire a mutable reference is by calling the `Rc::make_mut` method. If the tree has been cloned prior to the update, this call will copy each node while descending from the root to the leaf node, effectively performing *path copying*.

3.2.2 Structures, enumerations and cache locality

With the knowledge of how `Rc` helps to manage the memory and its semantics we can move on to the definition of the `RrbTree` node using the following Rust constructs:

- Structures or *structs* are used to define a custom data type that lets you package together multiple related values.
- Enumerations or *enums* allow to define a type by enumerating its possible *variants*.

Conceptually, the `RrbTree` consists of three node types: a balanced branch node, a relaxed branch node, and a leaf. Depending on the tree level, branch nodes reference either other branch nodes or leaves. Thus, the node definition has to be generic over the child node type.

Rust does not support inheritance in structs. It does, however, provide alternative ways of defining data types that are conceptually related: enumerations and *trait objects*[7].

Trait objects are objects that share common behavior by implementing the same interface. The main advantage of trait objects is their extensibility. Enums, on the other hand, cannot be extended with more variants outside of their declaration. Since tree nodes are an implementation detail of persistent vectors and are not expected to be extended, enums were favored over trait objects.

Each enum variant can have a different set of fields. Hence, the enum size is capped by its largest variant to guarantee that all variants can be used interchangeably.

Defining `RrbTree` nodes in Rust

We can finally bring everything together using structs and enums in combination with `Rc` to define the `RrbTree` node. Listing 14 presents the structure as it is in the library.

Each node type has received its own struct definition. An eagle-eyed reader will notice that the `Node` enum in Listing 14 could have been declared in a more concise way by having the node fields defined directly within the variant. Even though it seems to be a more intuitive approach, it comes at a hidden cost.

The enum size and the cache locality As illustrated in Figure 3.2, enumerations take up as much space as their largest variant[8]. If the node fields

[7] https://doc.rust-lang.org/book/ch17-02-trait-objects.html

[8] The illustration contains a subset of containers from the Rust container cheat sheet: https://docs.google.com/presentation/d/1q-c7UAyrUlM-eZyTo1pd8SZ0qwA_wYxmPZVOQkoDmH4.

```rust
#[cfg(feature = "arc")]
type SharedPtr<T> = Arc<T>;

#[cfg(not(feature = "arc"))]
type SharedPtr<T> = Rc<T>;
// ^ SharedPtr<T> is a type alias that is assigned either Arc<T> or
// Rc<T> depending on the thread-safety requirements and configuration.

enum Node<T> {
    RelaxedBranch(SharedPtr<RelaxedBranch<T>>),
    Branch(SharedPtr<Branch<T>>),
    Leaf(SharedPtr<Leaf<T>>),
}

struct RelaxedBranch<T> {
    children: [Option<Node<T>>; BRANCH_FACTOR],
    sizes: [Option<usize>; BRANCH_FACTOR],
    len: usize,
}

struct Branch<T> {
    children: [Option<Node<T>>; BRANCH_FACTOR],
    len: usize,
}

struct Leaf<T> {
    elements: [Option<T>; BRANCH_FACTOR],
    len: usize,
}
```

Listing 14: Definition of the `RrbTree` node

were declared within the enum variants directly, the enum size would be equal to the size of `RelaxedBranch`, resulting in the balanced tree reserving as much space as if it was relaxed.

The more space nodes use, the more expensive the memory allocations are.

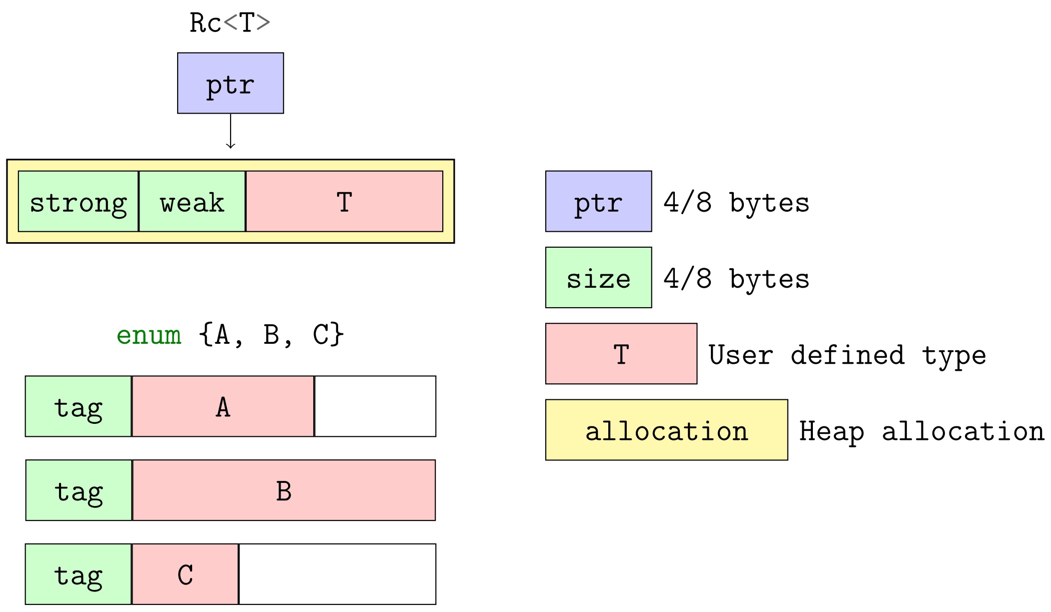


Figure 3.2: The memory layout of Rust containers

Also, bigger nodes are more likely to not fit into the CPU cache lines, negatively impacting the performance even more.

To avoid the unnecessarily large memory footprint of the `Node` enum, the decision was made to extract the node fields into structs as demonstrated in the Listing 14. By wrapping the struct instances into a smart pointer such as `Rc`, we explicitly move the allocation of the node to the heap.

The size of the `Rc` pointer is independent of the type it encapsulates. Hence, the size of the `Node` enum boils down to the size of the enum tag, the weak and strong reference count fields of `Rc`, and the actual pointer to the heap.

This, in turn, means that the `Node` enum variants will be equally sized, taking as little space as possible. When compiling the library for a machine with the 64-bit CPU architecture, the enum size is 16 bytes, where 8 bytes are reserved for the enum tag, and 8 bytes for the reference-counted pointer.

Figure 3.3 visualizes how the enum and struct definitions of the `RrbTree` node

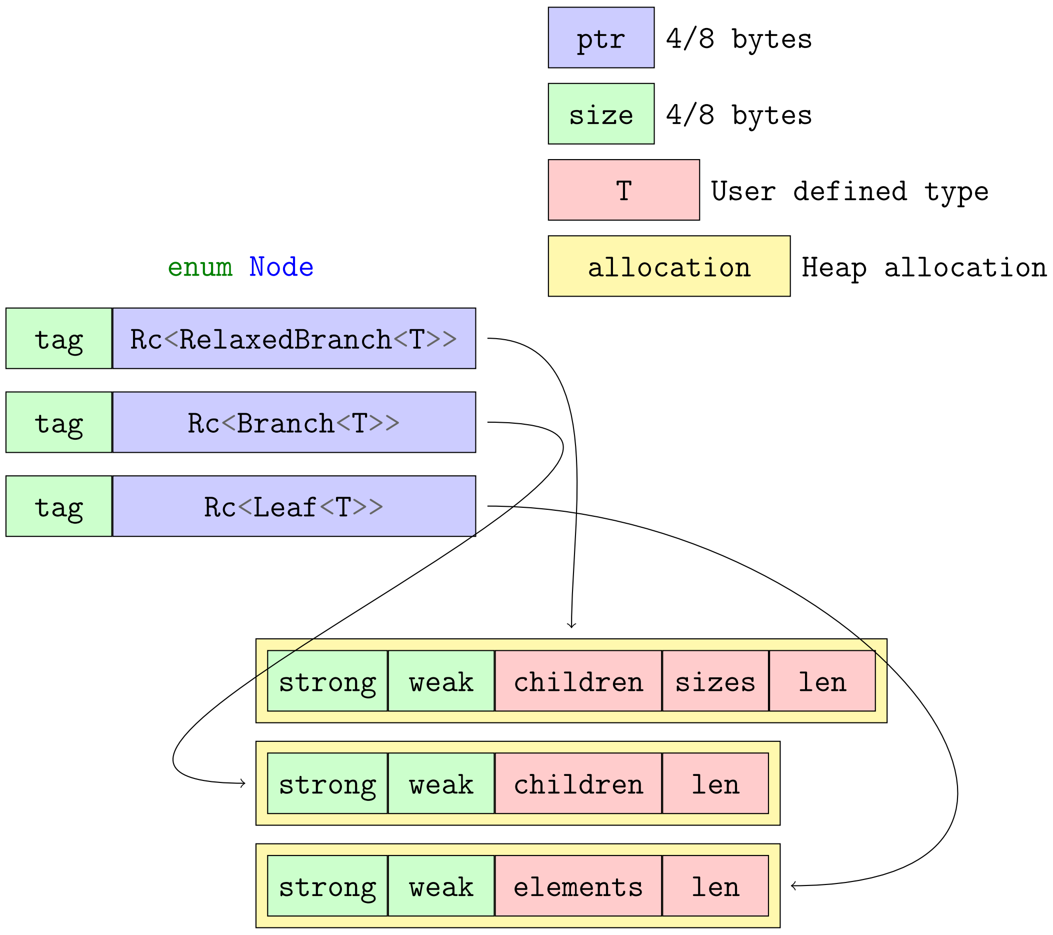


Figure 3.3: The memory layout of the `RrbTree` node

are arranged in memory.

3.3 Pay only for the features you use

The ***pvec-rs*** project draws inspiration from one of the Rust key design features — zero-cost abstractions. In this section, we will look at how `PVec` follows that design principle by offering optimizations such as transience and dynamic representation, without sacrificing Rust's safety guarantees.

3.3.1 Transience as the language feature

Popular persistent collection libraries in other languages, such as *immutable-js*[9] for JavaScript, or collections in the standard library of *Scala*[10], often provide an interface that is different from the interface of the standard data structures.

As demonstrated in Listing 15, instead of updating the vector in-place, the operation returns a new instance that contains the change. When a collection is set up like that, the library can guarantee that the original vector will stay unmodified, eliminating the whole class of bugs such as race conditions.

This, in turn, means that every modification will cause a copy. For example, in a pure function where vector processing is scoped strictly to the function body, there is no risk of introducing a race condition, and therefore, no need to create new copies. A solution to this problem was first introduced in the *Clojure* programming language in the form of *transient* data types. A *transient*, persistent vector is a vector that is persistent but can be updated in-place without generating unnecessary copies[11].

Rust's compiler, however, eliminates race conditions by forbidding simultaneous mutation of value by enforcing ownership and borrowing rules. The ownership rule essentially means that an object can be owned only by a single entity at a time. Borrowing rules state that any number of immutable references can be created given that there is no *mutable* references existing. Furthermore, if a mutable reference exists, Rust ensures that it is the only *unique* reference.

This leads us to the question of whether the traditional persistent collection interface is needed in Rust. The short answer is no because the compiler protects developers from common mistakes. Nicholas Matsakis, the lead engineer of the Rust compiler, elaborated why in Rust persistent vectors can have an ordinary interface in his blog[12].

[9] https://immutable-js.github.io/immutable-js/

[10] https://docs.scala-lang.org/overviews/parallel-collections/overview.html

[11] See Section 2.3 for more details.

[12] http://smallcultfollowing.com/babysteps/blog/2018/02/01/in-rust-ordinary-vectors-are-values/

Unique access as transience

Unique access as a term stems from the fact that in Rust, a mutable reference to an object must be *unique*. By leveraging this guarantee, persistent vectors in Rust can have a conventional mutable interface without sacrificing safety.

```rust
fn main() {
    let vec_1 = immutable_api(42);

    let vec_2 = mutable_api(42);
    let mut cloned_vec_2 = vec_2.clone();
    // ^ increments the reference count, but no copying yet

    cloned_vec_2.push(34);
    // ^ an attempt to update the tree with rc > 1
    // triggers the path copying algorithm
}

fn immutable_api(size: usize) -> PersistentVec<usize> {
    let mut vec = PersistentVec::new();
    for i in 0..size {
        vec = vec.push(i); // <- creates a new instance on every push
    }
    vec
}

fn mutable_api(size: usize) -> PersistentVec<usize> {
    let mut vec = PersistentVec::new();
    for i in 0..size {
        vec.push(i); // <- new instances are not created
    }
    vec
}
```

Listing 15: The persistent and conventional interfaces of vectors

As demonstrated in `mutable_api` in Listing 15, the conventional interface allows to update the vector in-place without creating copies.

This is possible because of the conditional *copy-on-write* semantics of `Rc` pointers that are used at the core of `RrbTree`. If the reference count is at most one, `Rc::make_mut` will not copy the underlying value but rather update it in-place.

When cloned on line 5, the reference count of the root node of the underlying `RrbTree` is incremented by one. The path copying algorithm is executed only when trying to mutate the cloned instance on line 8, leaving the original `vec_2` unmodified.

Path copying that is delayed to the first update means that the clone operation is effectively free in cases when a vector is cloned and shared for read-only purposes.

Transience in Clojure is different Transience in Clojure[13] is not identical to the unique access optimization in Rust. These are the main differences:

- In Clojure, transient vectors are confined to the thread they are created on. Therefore, they cannot be modified on another thread. Persistent vectors in Rust can be safely moved between threads as long as the ownership and borrowing rules are followed.
- Calling any method on a transient vector after it transitions back to persistent is prohibited and will cause an exception in Clojure. In Rust, there are no special restrictions imposed on vectors.
- To initialize a transient collection in Clojure, one has to use special syntax such as the `transient` function, or `persistent!` to convert a transient instance back to persistent.

3.3.2 Dynamic representation

A unique property of `PVec` is that it features an interface identical to the standard vector, contrary to the conventional persistent interface that is significantly different.

If two vector types have identical interfaces, they can be used interchangeably without any additional changes made to the program. Hence, it becomes

[13] https://clojure.org/reference/transients

possible to dynamically choose a vector type depending on the context and performance requirements.

The dynamic representation explores this idea by using both standard and the tree-based vectors to ensure the best possible performance, depending on the situation.

Deciding when to switch the representation

Standard vectors are very fast in almost all operations due to their efficient, hardware friendly design. Exceptions are operations that rely on copying, such as concat, split, and clone. Tree-based vectors, on the other hand, offers effectively $\mathcal{O}(1)$ clones and good all-around performance.

To take the best of both worlds, a vector can start as standard vector and when it is cloned, it will be transformed into a tree-based one. Essentially, by switching the representation only when cloning, we offset the cost of using trees until they are beneficial[14]. This optimization fits into the overall "Pay only for the features you use" idea, as it prevents the user from paying the abstraction cost until it is used.

Switching the representation

Essentially, a persistent vector type `PVec`, can be backed by two different vector types at runtime. It starts as `Vec`, and then switches to the `RrbVec` on the first clone. The process of switching from one vector type to another is called *spilling*, as it maps a contiguous chunk of memory into smaller pieces that are used as leaf nodes to build `RrbVec`.

[14]Nicholas Matsakis proposed a variation of this idea as future work in his blog: http://smallcultfollowing.com/babysteps/blog/2018/02/01/in-rust-ordinary-vectors-are-values/

```
struct PVec<T>(Representation<T>);

enum Representation<T> {
    Flat(Vec<T>),
    Tree(RrbVec<T>),
}
```

Listing 16: Memory layout of `PVec`

`PVec` has two representations: `Flat` that is using `Vec`, and `Tree` that is backed by `RrbVec`. The initial capacity of both representations is equal to the branching factor of 32 by default.

Spilling happens by converting the flat vector into an iterator of sub-arrays that are 32 elements wide and pushing them directly onto `RrbTree` as leaf nodes. If the last chunk is smaller than 32, then it will be used as the tail for the new `RrbVec`. See Figure 3.4 that visualizes the transition.

One could have considered splitting up the original vector into chunks and reusing them instead of copying. Unfortunately, heap allocators do not have a mechanism for safely splitting an existing allocation into smaller ones. The Rust's allocator interface[15] reflects this limitation by exposing functions that allow allocating and freeing memory but no subdivision of existing allocations.

Additionally, the cost of spilling needs to be paid only once, and will be amortized by $\mathcal{O}(1)$ consequent clones.

[15] https://doc.rust-lang.org/std/alloc/struct.System.html

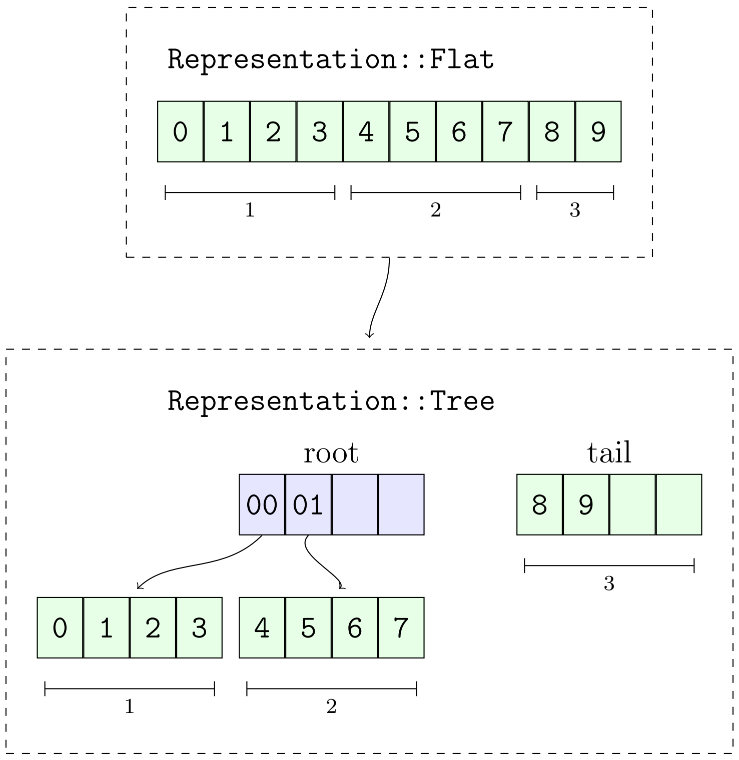


Figure 3.4: Switching the vector representation

Chapter 4

Performance evaluation

The Big $\mathcal{O}$ notation is a useful theoretical tool for reasoning about scalability and performance, and by design, it does not consider factors such as execution environment. It disregards constant factors as they are not significant for the growth rate of functions. It also does not consider the architecture of the CPU and of the memory [4], which indeed influence performance. Furthermore, the same applies to software, such as operating systems, schedulers, virtual machines, et cetera. Hence, often algorithms, which are expected to be equally fast based on $\mathcal{O}$, may differ substantially in performance when evaluated experimentally.

Thus, in this chapter, I will introduce a methodology for the experimental performance evaluation of `RrbVec`, `PVec`, and their variants, in comparison to implementations from `ImVec` and Rust's standard library. We will look at:

- First, identifying directions for performance comparisons.
- Then a methodology for collecting reliable measurements.

To begin with, I will present the focus areas.

4.1 Evaluation dimensions

To evaluate the efficiency of the proposed optimizations, we need to test various tree-based vectors, such as `RbVec`, `RrbVec`, and `PVec`. Implementations from `ImVec` and the standard library will be evaluated as well. All vector variants are specified in Section 4.1.

`Vec`	A vector implementation from the Rust's standard library
`RbVec`	The `RbTree`-based vector implementation
`RrbVec`	The `RrbTree`-based vector implementation
`PVec`	The `RrbTree`-based vector implementation with dynamic representation
`ImVec`	The `RrbTree`-based vector implementation from the third party library ***im-rs***[1]

Table 4.1: A table of vector implementations

4.1.1 Time and space

There are two main dimensions for evaluation: time and space.

Runtime benchmarks are categorized by the scope of what they test: individual operations and overall performance:

1. The first category evaluates vector operations in isolation. These tests will be executed on a single thread, and thus, will be referred to as *sequential benchmarks* from now on.

2. Tests that replicate common use cases that involve more than one operation and exercise the vector as the whole belong to the second category. They will be executed in a more complex environment on multiple threads and will be referred to as *parallel benchmarks*.

Space or memory benchmarks are designed to:

1. Measure the memory overhead of the tree-based vectors.

[1]Both `ImVec` and `PVec` use `RrbTree` at its core. It has been developed independently in parallel to `PVec` at the time of writing this paper: https://crates.io/crates/im.

2. Evaluate whether structural sharing helps to reduce the memory footprint.

4.1.2 Objectives

The objectives are the questions that we will address when reviewing benchmark results. They apply both to the runtime and memory tests.

Balanced vs. relaxed vectors

As instances of `RrbTree` are generally not perfectly balanced and involve the use of size tables for the radix search, they are expected to be somewhat slower compared to `RbTree` instances in core operations. The goal is to measure the overhead induced by the relaxed nodes if present at all.

Before each benchmark run, an instance of `RrbVec` will be prepared by concatenating small vectors together. The number of the relaxed nodes is affected by the vector size.

Dynamic representation

A substantial advantage of the tree-based vectors is that they are cheap to copy. Other operations, however, are faster for `Vec` due to its optimal memory layout. Dynamic representation offers a strong performance based on `Vec` until the vector is cloned, switching its representation to `RrbVec`.

Ideally, `PVec` should demonstrate the same performance of the type that represents it. The goal is to measure the `PVec` performance in practice and to check if it adds overhead over the types used as its representations: `Vec` and `RrbVec`.

Unique access

While `RrbVec` performs well as a persistent data structure, it can perform even better when persistence is not required. An example is a function creating and returning an instance of `RrbVec`, where all versions except the returned one are disregarded.

Luckily, the persistent vector implementation presented in this project takes advantage of Rust's compiler capabilities of tracking object aliasing. Thus, it avoids redundant copying on mutation if the given object is uniquely accessed. This behavior is similar to transience in the Clojure's persistent vector implementation [11], but not entirely identical.

In Rust, non-transient, persistent behavior can be enforced by cloning the object before performing a mutation. The objective is to measure the cost of using the clone operation in the persistent vector.

4.2 Measuring the runtime

Experimental performance evaluation introduces a set of unique challenges. For instance, depending on the workload, operating systems may allocate more resources for demanding tasks, by reducing runtime for others [8]. Such non-deterministic behavior may lead to profiling results that vary from run to run significantly.

Benchmarking frameworks were introduced to solve this problem. They are designed to get stable measurements by executing the same test thousands of times. Some of them, such as criterion for Haskell[2] and Rust[3], JMH for Java[4], and ScalaMeter for Scala[5], introduce statistical methods for the detection and elimination of exceptionally different runs, known as outliers.

4.2.1 Benchmarking frameworks for Rust

There are several benchmarking frameworks available for Rust, and unfortunately, none of them has reached a stable release yet. However, some of them are being actively used in the Rust community and have proven to produce reliable results.

There are several criteria which a suitable framework has to meet:

[2]https://hackage.haskell.org/package/criterion
[3]https://crates.io/crates/criterion
[4]https://openjdk.java.net/projects/code-tools/jmh/
[5]https://scalameter.github.io/

- Collecting multiple samples where each sample consists of multiple runs to ensure consistent results.
- Detection and elimination of outliers.
- A way for setting up a benchmark before each run.
- A way for preventing compiler optimizing benchmark code away.
- An option to measure execution time without *drop*[6].

Rust's benchmark tests

Rust's testing framework provides an experimental feature[7] that enables developers to write test benchmarks. Those benchmarks are executed thousands of times until results are stabilized. Also, it provides a black-box function[8] which is opaque for the compiler.

However, Rust's testing framework does not detect and eliminate anomalies. It also does not provide APIs for setup routines, which makes it impossible to create benchmarks that rely on preconditions.

Criterion for Rust

Criterion for Rust is a powerful and statistically rigorous tool for profiling code. It features outlier elimination, setup routines, and is capable of generating graphs using gnuplot[9]. At the moment of writing, it is the only framework that has an option to avoid the timing of *drop*.

It is compatible with the stable release of the Rust compiler. Thus, Criterion was chosen as a benchmarking framework for this project.

[6]See Section 1.1.3 for an introduction to *drop*.

[7]At the moment of writing in May 2020, benchmarks are available only in the nightly build of Rust.

[8]Black box function contains inline assembly instructions, which compiler cannot make any assumptions about. Hence, it prevents the compiler from optimizing the code, which otherwise would be considered "dead" or unused.

[9]http://www.gnuplot.info/

Alternatives

Other less popular frameworks, such as ***bencher***[10] and ***easybench***[11], were not considered due to the lack of features necessary for the experiments, such as setup routines, optional measurement of *drop* outlier detection, et cetera.

4.2.2 Configuration and input size

All tree-based vector implementations, such as `RbVec`, `RrbVec` and `PVec` were configured to use a branching factor of 32, as it provides the best tradeoff for the performance of both read and write operations [2].

Each benchmark was executed against a range of input arguments. The input range is specified on the case by case basis depending on the benchmark.

For this project, for each input argument, Criterion is configured to capture 10 samples. The number of runs per sample is determined dynamically by the library to achieve optimal execution time.

4.2.3 Garbage collection

One of the design goals for this project was to avoid using experimental and unsafe Rust language features. Hence, ***pvec-rs***, relies on the memory management means provided by the Rust's standard library only, such as `Rc`. Other automatic memory management mechanisms such as third-party garbage collectors were left out for future work.

Threadsafe reference counting Parallel benchmarks will be evaluated using only `Arc` pointers, as Rust's compiler forbids passing non-threadsafe types between threads, such as `Rc`.

4.3 Parallel benchmarks and Rayon

Unlike the sequential benchmarks that measure operations in isolation, the parallel benchmarks time the whole experiment, including several vector op-

[10]https://crates.io/crates/bencher
[11]https://crates.io/crates/easybench

erations and the framework used to parallelize the work.

This is an acceptable tradeoff, as the objective is to evaluate the overall performance. Additionally, we also want to check whether the relaxed append and split of `RrbVec` have a positive performance impact.

Data parallelism frameworks, such as Rayon[12], Cilk[13], and Scala's parallel collections[14], split the work into smaller chunks to facilitate parallelism. Thus, data structures with fast split and append operations are critical for optimal performance for such frameworks.

In this section, we will take a look at how to configure Rayon and review how it distributes the work across threads.

Rayon Rayon is a data parallelism library for Rust that helps to turn sequential code into parallel one with as little work as possible. Loops and iterators are often used to process collections sequentially. Rayon, on the other hand, offers a potentially more efficient alternative to them in the form of parallel iterators. It takes advantage of modern processors, by dividing the work between available cores as long as it is beneficial.

```rust
// sequential iterator
vec![1, 2, 3]
    .into_iter()
    .for_each(|x| println!("{}", x));

// rayon's parallel iterator
vec![1, 2, 3]
    .into_par_iter()
    .for_each(|x| println!("{}", x));
```

Listing 17: An example of using sequential and parallel iterators

[12]https://crates.io/crates/rayon
[13]http://supertech.lcs.mit.edu/cilk/
[14]https://docs.scala-lang.org/overviews/parallel-collections/overview.html

Parallel iterators The convenience of Rust iterators is in the provided operators that are called *combinators*. Combinators can be chained and combined, allowing a developer to perform complex manipulations of iterators safely and efficiently.

Combinators provided by parallel iterators are similar to the iterator ones but entirely identical. As iterators process values sequentially, there is a set of combinators that expect values to be emitted in a particular order. As the parallel iterators are designed to process data in any order, inherently sequential combinators are not applicable. Thus, Rayon might not be a suitable solution for algorithms relying on the sequential order of execution.

Another requirement for parallel iterators is that the type of values it works with have to implement the `Send` trait. Therefore, using non-threadsafe types such as `Rc` in combination with Rayon is prohibited.

4.3.1 Work splitting

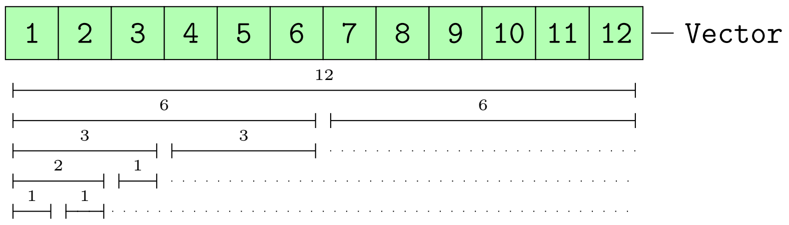


Figure 4.1: Visualization of work splitting in Rayon

At the core, Rayon relies on the fork-join[15] pattern for dividing and distributing the work between threads.

When a parallel iterator receives values from a collection like a vector, Rayon attempts to repeatedly divide the work into chunks among threads until the chunk is small enough for a single thread. For an example, see Figure 4.1.

A method that splits the work is `rayon::join`, and it accepts two *closures*[16].

[15]The fork-join pattern was pioneered in 1963 [12], and was popularized by the Cilk project [3]

[16]Rust's closures are functions that can be saved in a variable or passed as arguments to other functions.

Its usage is demonstrated in Listing 18.

Next, Rayon decides whether it is beneficial to parallelize the work by evaluating the factors such as the number of available threads, the *split factor*[17], and the workload. If the problem is small enough, it is solved sequentially. Otherwise, it is subdivided further. When both closures finish working, the results are combined and returned to the caller.

```
rayon::join(
    || do_something(...),
    || do_something_else(...)
);
```

Listing 18: An example of using Rayon's join

By default, the thread number allocated by Rayon is equal to the number of cores available in the system. To observe how the thread number affects the performance, Rayon's thread pool will be configured to work with 2, 4, and 8 threads.

4.3.2 Work distribution

Ideally, tasks split between threads take the same amount of time to process. Unfortunately, this is often not the case, resulting in poor utilization of resources. To fairly distribute the work, Rayon attaches a work queue to each thread. A thread keeps processing the queue until it becomes empty. When the queue is empty, a thread can steal work from another thread. This technique is known as work-stealing.

4.4 Measuring the memory footprint

This section presents the methodology used to evaluate the memory footprint of both the tree-based and flat vector types. The benchmark suite is expected to measure the overall memory footprint of the vector, including the stack and the heap memory.

[17]The split factor is defined by the minimum and the maximum size of the work chunk.

4.4.1 Memory profiling tools

As of the time of writing, there is no established practice or a framework for measuring the memory usage tailored towards Rust.

However, the operating systems already provide profiling tools for measuring memory usage. These tools should provide reliable results, as the memory footprint of the process does not change as much between runs. Hence, it should be sufficient to run each test once.

Two approaches were considered:

- Measuring heap allocations by retrieving statistics from the system allocator.
- Capturing the memory usage of the process when running the benchmark application.

Allocation tracking

The first option is not ideal because it only evaluates the heap memory usage. Additionally, a custom memory allocator such as ***jemallocator***[18] is required to access statistics. Using a non-default allocator in tests also means that benchmarks will not accurately reflect the performance in production applications.

Process benchmarking

Benchmarking by measuring the process footprint allows to capture both the stack and the heap usage, and it does not depend on any special Rust debug or allocator features. This, in turn, means that the benchmarks can be optimized as if they were compiled for the production use, putting them as close as possible to real use cases.

Implementation A new ***pvec-memory*** crate is introduced to ***pvec-rs*** that includes an application for executing the memory benchmarks. Its responsibilities are to configure, run, and collect the test results. It will be referred to as *memory bencher* from now on.

[18]https://crates.io/crates/jemallocator

The memory bencher contains a list of tests, input sizes, and vector types. It executes benchmarks by running them through the command line program called *time*, which outputs information about the process, including the maximum amount of memory used in bytes. The memory bencher finds this value in the *maximum resident set size* field in the output of *time*.

The program code occupies a part of the process memory and needs to be accounted for. The bencher measures the footprint of the program without running any tests and then subtracts it from the benchmark results.

The *time* utility included in macOS is different from the GNU[19] variant, that comes with more options and outputs the *maximum resident set size* in kilobytes instead of bytes. The memory bencher was only executed on macOS and required changes to support other platforms.

4.5 Presentation of results

The measurements are done in a number of samples that can be configured. For this project, the number of samples is set to ten. Each sample consists of one or typically more iterations of the routine. The elapsed time between the beginning and the end of the iterations, divided by the number of iterations, gives an estimate of the time taken by each iteration.

As measurement progresses, the number of iterations per sample increases. The presented graphs use the mean measured time for each function. Additionally, benchmarks were configured to avoid the timing of the execution of *drop*.

4.5.1 Execution environment

All benchmarks were executed on a computer with an octa-core processor with hyper-threading support, 32GB of DDR4 RAM, and 1TB solid-state drive. The operating system is macOS Catalina 10.15.4, with the stable Rust compiler version 1.42.0.

[19]https://www.gnu.org/software/time/

CPU	2,3 GHz 8-Core Intel Core i9.
RAM	32GB DDR4, 2400MHz.
Disk	1TB SSD.
OS	macOS Catalina v10.15.4.
Rust	v1.42.0.
Criterion	v0.3.1
im-rs	v14.0.0.

Table 4.2: Hardware and software specification used for benchmarking

4.5.2 Running benchmarks

The benchmarks can be compiled and executed using Rust's package manager named *cargo*. By default, sequential benchmarks are executed against the non-threadsafe variant of a persistent vector. To select the threadsafe variant, the user can pass the *arc* feature flag, as demonstrated in Listing 19. A particular benchmark can be specified by name as an argument to cargo.

```
# benches of the non-threadsafe implementation
cargo bench

# benches of the threadsafe implementation
cargo bench --features=arc
```

Listing 19: Executing sequential benchmarks

To execute parallel benchmarks, the user needs to pass both the *arc* and *rayon_iter* feature flags:

```
cargo bench --features=arc,rayon_iter
```

Listing 20: Executing parallel benchmarks

Criterion can also generate HTML reports that include charts created by *gnu-*

plot[20]. Results can be found in the `pvec-rs/target/criterion` directory, and if *gnuplot* is available, the HTML report will be located at `report/index.html`.

For more information on available options and parameters for Criterion, please consult the library documentation[21].

Memory benchmark results

Memory benchmarks produce consistent and reproducible results. Hence it is enough to execute them only once. Due to the differences in how *time* works on different operating systems, macOS is the only platform supported by the memory bencher at the moment. Listing 21 demonstrates how to execute the tests:

```
# navigate to the benches-mem crate
cd benches-mem

# execute the script that invokes
# cargo to compile and run benchmarks
# in the release mode
sh benches.sh
```

Listing 21: Executing memory benchmarks

The generated report is located in the `pvec-rs/target/release/report` directory, and it is subdivided into folders, each named after the benchmark. Each folder contains csv[22] files, where each file is name after the evaluated vector type. The report file contains two columns corresponding to the input size and the memory footprint in bytes.

4.5.3 Verbose benchmark results

Verbose benchmark results are published separately due to the number and the size of the report files. They include additional data, such as:

[20] http://www.gnuplot.info/
[21] https://docs.rs/criterion/0.3.1/criterion/
[22] Comma-separated values.

- Mean, median, and the standard deviation values per benchmark run.
- Violin plots demonstrating the stability of results.

The violin plots reveal how stable the measurements are. The wider the violin is, the more significant is the difference between timings of collected samples.

The benchmark reports are published in two configurations: with a non-threadsafe[23] and a threadsafe[24] reference counting pointers.

[23]https://abishov.com/pvec-rs/reports/rc/report/index.html
[24]https://abishov.com/pvec-rs/reports/arc/report/index.html

Chapter 5

Benchmarks and results

In this chapter, we will define our sequential, parallel, and memory benchmarks, and then evaluate the results of executing them. The overall performance will be discussed, as well as the impact of the following optimizations:

- The effect of `RrbTree` relaxation on performance of all vector operations.
- The performance impact of the unique access optimization.
- The effectiveness of the dynamic representation.

Reading notes Implementations postfixed with `(r)` in the figure legends were configured to use the relaxed RRB-tree in the benchmark. `(s)` stands for standard and will be applied only to `PVec` when the flat representation is used. If not specified, the balanced RB-tree is used.

5.1 Sequential benchmarks

Each benchmark described in this section focuses on a particular operation of a vector. To avoid ambiguous results, each test exercises only one operation at a time. Operations that modify a vector instance, such as push, will have a complementary version of the benchmark, which also uses the clone operation. It is necessary to compare the path copying and naive algorithm used in the tree-based and standard vectors correspondingly.

The following operations were evaluated for vector implementations:

Indexing	Accessing vector values.
Updating	Updating existing values.
Pushing	Adding new values to the end of a vector.
Popping	Removing values at the end of a vector.
Appending	Concatenating values of one vector to another.
Splitting	Slicing one vector into two at a given position.

Table 5.1: Operations evaluated in the sequential benchmarks

Benchmark structure Some benchmarks depend on preconditions. For example, to test indexing, we first need to create a vector with values. Since building a vector instance is not a part of that test, it happens in the setup routine. Hence, benchmarks with preconditions are executed in two steps: the setup and the actual test.

Benchmarking dimensions Every benchmark for a core operation is parameterized over the vector size. By providing different arguments, we can observe how the performance of vectors is affected in response. This is especially insightful for the tree-based implementations, where the size of the vector influences the height of the tree, which has a negative impact on performance. The output of a benchmark for a given size is the mean runtime in ms.

Results This section contains performance numbers for the non-threadsafe vector implementations.

5.1.1 Indexing

In this section, we will define benchmarks for accessing values that model the most common ways of working with vector:

- Sequentially accessing values by index and iterator.
- Accessing values at randomly generated indices.

Use cases listed above address two objectives: first, how much overhead relaxed nodes of `RrbTree` introduce, and second, the efficiency of the dynamic representation in `PVec`.

The indexing benchmarks share the same setup routine for generating a vector. Balanced tree-based vectors are created by pushing 64-bit integers, while the relaxed types are generated by concatenating vectors together.

The vector size is passed as an argument and falls into the range of [20, 1 M].

Figures for the index operation are separated by sequential and random access, where the sequential benchmark results are subdivided into index and iterator figures.

Index sequentially

The benchmark loops over an array of [0, N) indices, and reads values from a vector at the given position. Only immutable references to values are acquired.

Iterators consume the tree-based vectors by chunks, rather than by individual values. They also take ownership of values instead of borrowing them.

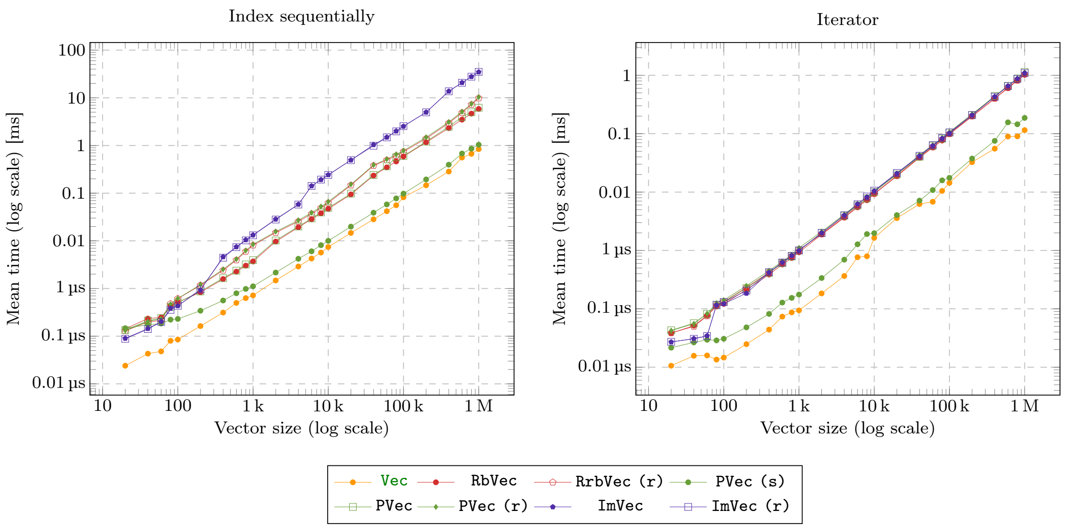


Figure 5.1: Benchmarking results of index sequentially and iterator

Results As expected, `Vec` shows the best results in this test. As it is represented by a contiguous chunk of memory, it takes full advantage of CPU cache locality. Besides, its structure is not affected by the method used to build it, whereas `RbTree` and `RrbTree` based vectors are.

Both balanced and relaxed `ImVec` variants are slower in comparison to `RrbVec` in the [100, 1 M] input range by a factor of 2.06. For smaller inputs, `RrbVec` it is slightly faster by a factor of 1.18.

Balanced vs. relaxed The difference between `RbVec` and `RrbVec` becomes noticeable as the problem size grows. The balanced variant is faster than the relaxed one by a factor of 2.68 in the [100, 1 M] input range. This is expected because `RrbVec` introduces relaxed nodes, which rely on the size tables to compute the path to the value.

This, however, is not the case for small problem sizes in the [0, 100] range, for which the concatenation algorithm produces a balanced tree. Hence, both balanced and relaxed vectors demonstrate similar performance in that range.

Dynamic representation `PVec` switches its internal representation from the standard vector to `RrbVec` as soon as cloned. This is evident from the plot Figure 5.1, where `PVec` is 4.21 faster than `RbVec`, but slower than `Vec` by a factor of 1.75.

Iterator

Results of the iterator benchmarks show approximately ten-fold improvement in performance over sequential indexing. This is expected, as iterators read the contents of the tree by chunks rather than by index.

`Vec` shows the best results, with a difference of 1.98 on average compared to `PVec`, and 9.12 in relation to `RrbVec`. `ImVec` is 1.47 ahead of `RrbVec` in the [20, 100] range.

Balanced vs. relaxed As iterator does not use size tables for index calculation for `RrbVec`, it performs identically well compared to `RbVec`. The same applies to `ImVec`.

Index randomly

In this benchmark, values will be read at random positions. Thus, it is quite likely that tested values will be located far apart in memory, potentially causing a cache miss. Additionally, results show whether the performance degenerates with randomness, as it would with linked lists, for example.

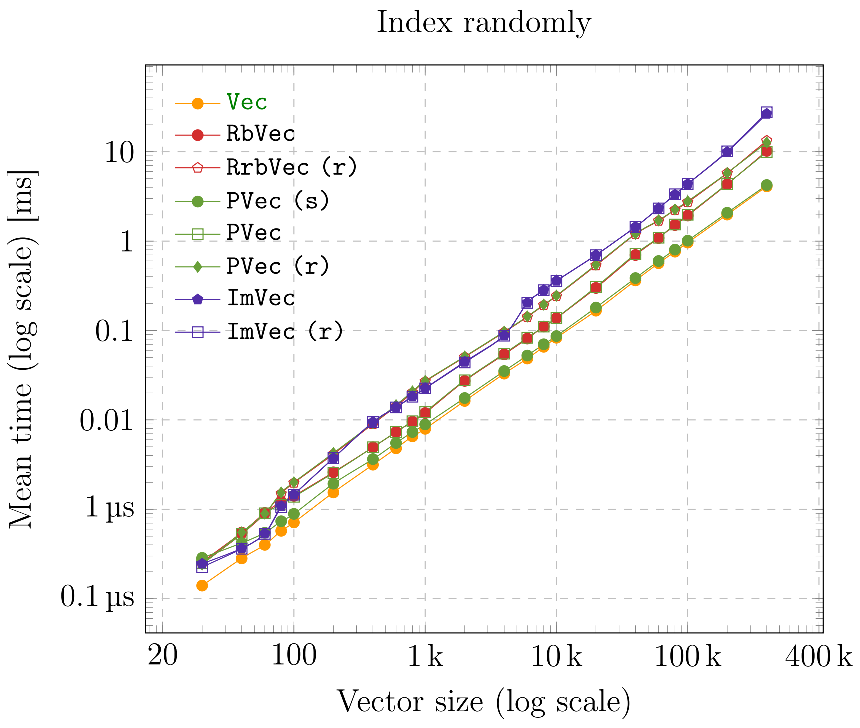


Figure 5.2: Benchmarking results of indexing randomly

By iterating N times, a value is accessed at random index, which is generated within the [0, N) range by using the ***rand*** crate[1]. According to the ***rand*** documentation, generated indices are uniformly distributed. The number generator is explicitly seeded to produce the same stream of randomness between runs.

[1]A Rust library for random number generation: https://crates.io/crates/rand

Results A noticeable difference compared to the sequential benchmark is that the performance gap between `PVec` and `Vec` is reduced from 1.76 to 1.14. The reason why `Vec` has lost its advantage is because of the frequent cache misses on access to random memory locations.

Balanced vs. relaxed `RbVec` outperforms `RrbVec` by 1.90 in the [100, 1 M] range. Both `RrbVec` and `ImVec` are equally fast with insignificant marginal differences.

Dynamic representation It is clear that `PVec`, when flat, is marginally slower compared to `Vec`. The performance difference remains consistent over the input range at 1.75.

5.1.2 Updating

There are two dimensions in which the update operation is evaluated. The first one is the order in which vector values are updated: sequentially and randomly. The second dimension introduces the clone operation to measure the cost of copying:

- Sequentially, with and without *clone*.
- At random positions, with and without *clone*.

The setup routine for all benchmarks is identical. As for the index benchmarks, it generates both balanced and relaxed variants of the tree-based vectors. The type of inserted values is an unsigned 64-bit integer.

The vector size is determined by the benchmark argument. The problem size domain for tests using clone is [20, 20 k], and [20, 100 k] range for benchmarks without clone.

The cost of naive cloning vs. path copying The path copying algorithm of `RbTree` enables cheap copies. `PVec` takes advantage of that by switching from the flat to the tree-based representation when cloned. Hence, the objectives are:

- Compare the performance of naive and path copying algorithms.

- Evaluate the efficiency of dynamic representation in `PVec`.

The overhead of relaxed nodes in `RrbTree` Relaxed nodes are more expensive to clone because of the size tables. Also, `RrbTree` is not perfectly balanced as `RbTree`, potentially causing taller trees. The results reveal if there is measurable overhead in practice.

Extending benchmarks with the clone operation The test keeps track of the cloned instance in a variable to ensure that at least two objects exist simultaneously when the update is executed. This is necessary because `Rc` pointers clone the underlying value on mutation only when the reference count is bigger than one. Thus, a clone must be present in the scope to enforce path copying.

Update sequentially

The test function iterates over vector at indices in the [0, N) range, acquiring a mutable reference to the value and incrementing it.

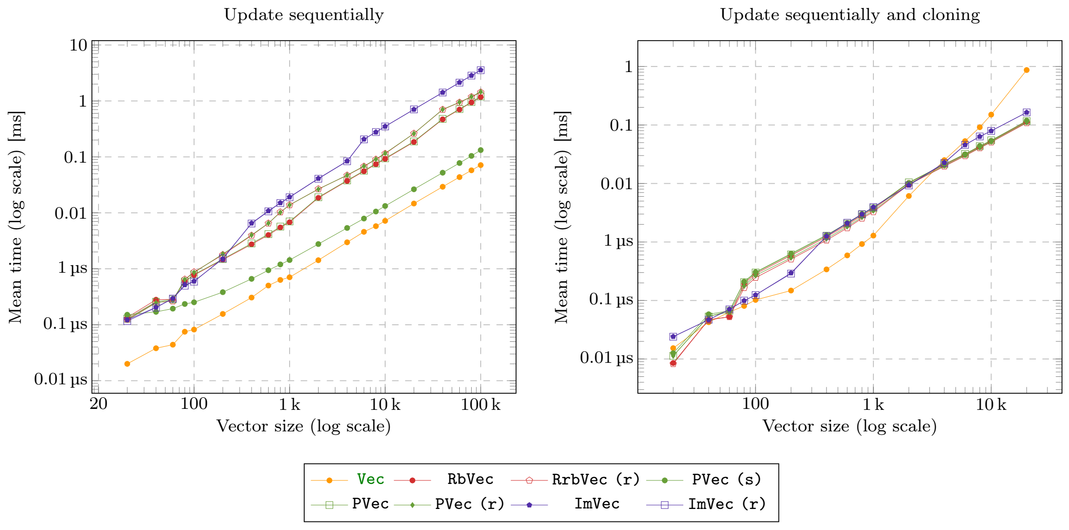


Figure 5.3: Benchmarking results of updating values sequentially

Results The standard vector is the fastest in the sequential updates test, with the closest runner-up being `PVec` with the mean difference of 2.32. When cloned, however, tree-based types, such as `RbVec`, start outperforming the standard vector after surpassing the 4 k size. The difference grows quickly, reaching 7.45 at the size of 20 k. It demonstrates how well path copying scales when cloning large data structures.

Updates are slower for `ImVec` compared to `RrbVec` by 2.47 on average. When cloned, however, the difference is less significant, varying mostly in the range of [20, 400].

Balanced vs. relaxed Relaxed nodes and size tables associated with them were expected to have a negative impact on the performance when copying. Though, the numbers in the clone test do not confirm that assumption. However, if updated without cloning, `RrbVec` is slower than `RbVec` by 1.33.

Dynamic representation When updating a vector, `PVec` is faster than both variants of `RbVec` by a factor of 2.22 on average. It is, however, slower than `Vec`, even though the standard vector is used as a representation. It is expected that `PVec` will introduce some overhead, as essentially, it introduces an additional abstraction layer.

Update randomly

The test function contains a loop, which is executed N times. In the loop body, value is updated by incrementing it, at the index that is randomly generated in the [0, N) range.

Results As with indexing, when values are updated randomly rather than sequentially, the performance gap between the `Vec` and `PVec` became as small as 1.11 due to the frequent cache invalidations caused by random access. Other than that, there are no significant distinctions.

5.1.3 Pushing

The push operation is evaluated by populating an empty and existing vectors. Both tests are also extended with the clone operation.

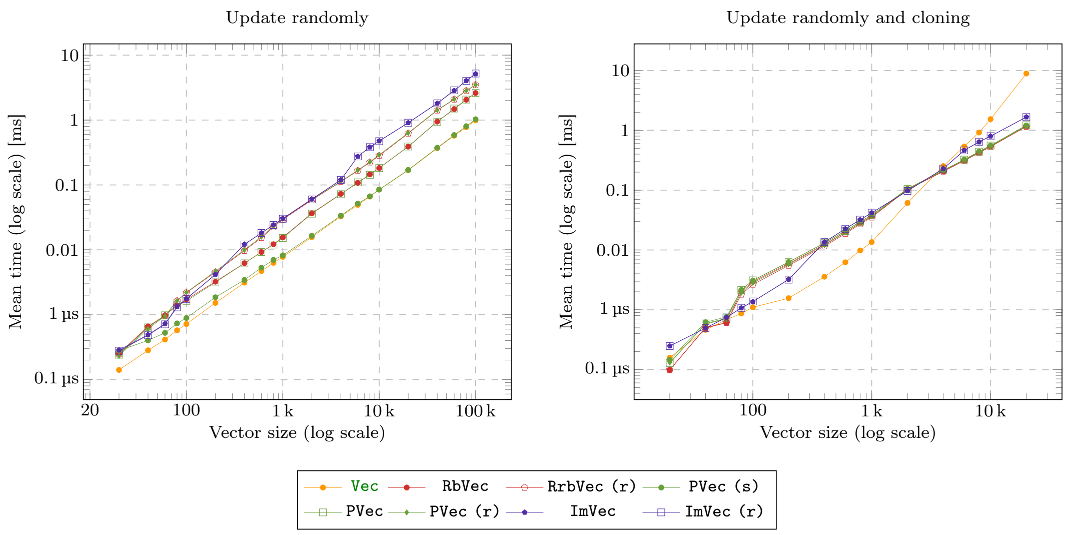


Figure 5.4: Benchmarking results of updating values randomly

The overhead of relaxed nodes in `RrbTree` The push operation is responsible for increasing the vector capacity. While the vector capacity calculation for `RrbTree` relies on the size tables, for `RbTree`, it is sufficient to know the level of the node and the branching factor. Additionally, instantiating relaxed nodes implies the allocation of size tables. All these factors combined are expected to make `RrbTree`'s push slower compared to `RbTree`. Thus, there is a dedicated benchmark that uses prebuilt, `RrbTree`, and `RbTree` based vectors to evaluate the difference.

Extending benchmarks with the clone operation To force `PVec` to switch from `Vec` to `RrbVec`, several preconditions have to be met, including the reference count being bigger than 1. Hence, the benchmarks above are extended to use clone, in the same way as for benchmarks of the update operation. Beyond the evaluation of `PVec`, the results are expected to reveal how tree-based vectors stack up to `Vec`. The input range for benchmarks using clone is [20, 40 k].

Benchmarks are divided into two use cases:

- Building a new vector from scratch by pushing values into it.

- Pushing values into an existing vector, both balanced and relaxed.

Both benchmarks are extended using the clone operation.

Building a vector

As vector is built from scratch in this benchmark, there is no need for a setup routine. The test function runs a loop over the [0, N) range of indices, and pushes the index as a value into a vector. The problem size range is [20, 1 M].

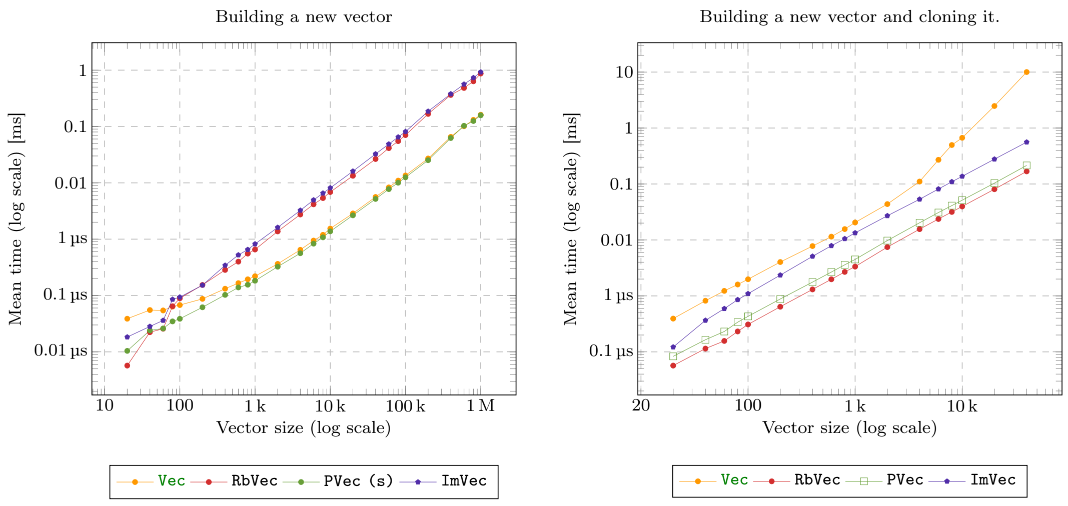


Figure 5.5: Benchmarking results of push

Results When building a vector in the [20, 100] range, tree-based types are slightly faster than `Vec`, as they take advantage of the tail optimization.

`ImVec` is almost as fast as `RbVec` with a difference of 1.18. When cloned, however, `RbVec` outperforms it by a significant amount of 3.61.

In the cloning benchmark, persistent vectors once more demonstrate how effective they are when copied, by `RbVec` being faster than `Vec` by a staggering factor of 12.85.

Dynamic representation Overall, `PVec` is the fastest vector, mostly due to the combination of using `Vec` with the pre-allocated space[2] for small sizes. Closer to the end of the size range, `Vec` and `PVec` align in performance with an average difference of 1.27.

When cloned, `PVec` is losing only to `RbVec` by insignificant 1.35, demonstrating the effectiveness of the dynamic representation.

Adding values to an existing vector

The push operation does not produce relaxed nodes in balanced trees. Hence, there is no way to evaluate the impact of relaxation in the benchmark of building a vector from scratch. Thus, in this benchmark, values are added to an existing vector, where vector can be both balanced and relaxed depending on the setup routine.

Objectives of the benchmark are:

- Check the cost of the complex sub-tree capacity and index computation.
- Measure the overhead of using size tables when cloning relaxed nodes.

The setup routine generates a vector of the fixed size of N and passes it to the test function. The balanced, `RbTree`-based vector is created by pushing values directly into it, while the `RrbTree`-based one, is created by concatenating several vectors together. Once a vector is created, the test function pushes N values onto it.

Results Even though size tables increase the node size in `RrbTree` and add complexity to the implementation, both relaxed and balanced trees show nearly identical performance in this test.

When push is called repeatedly, even for relaxed nodes, only balanced nodes are added to the tree. Eventually, all new nodes at the end of the tree, except the root, will be balanced. Thus, there is nearly no overhead of running push over a relaxed `RrbVec` in the given benchmarks.

[2]`PVec` is initialized to the capacity of branching factor, that is 32 in the test configuration.

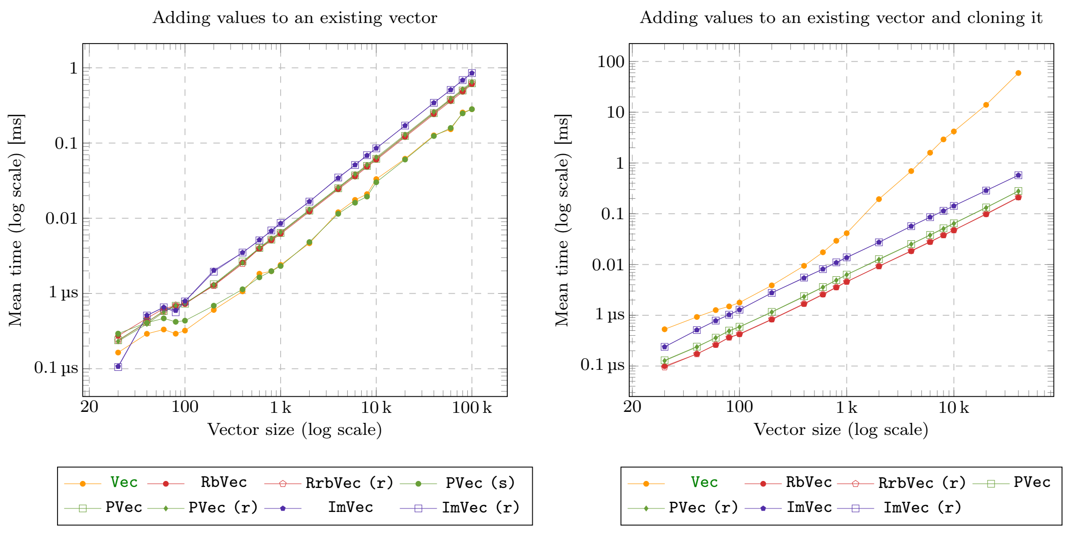


Figure 5.6: Benchmarking results of adding values to an existing vector

5.1.4 Popping

The pop operation manages the vector capacity as well as push. For `Vec`, it means shrinking the array and copying elements over. For tree-based vectors, it implies de-allocating nodes and reducing the height of the tree when necessary.

The benchmark is divided into two tests, namely pop and pop clone. The first test calls pop repeatedly in the loop until the vector is emptied, with the problem size range of [20, 60 k].

In the second benchmark, each pop operation will be followed by a clone. Both tests include balanced and relaxed vector types, which are prepared in the setup routine. The problem size range is [20, 40 k].

The overhead of relaxed nodes in `RrbTree` Lowering the height of `RrbTree` involves the tree capacity calculation using size tables, that comes at an additional cost. Thus, this test includes both balanced and relaxed variants of vectors.

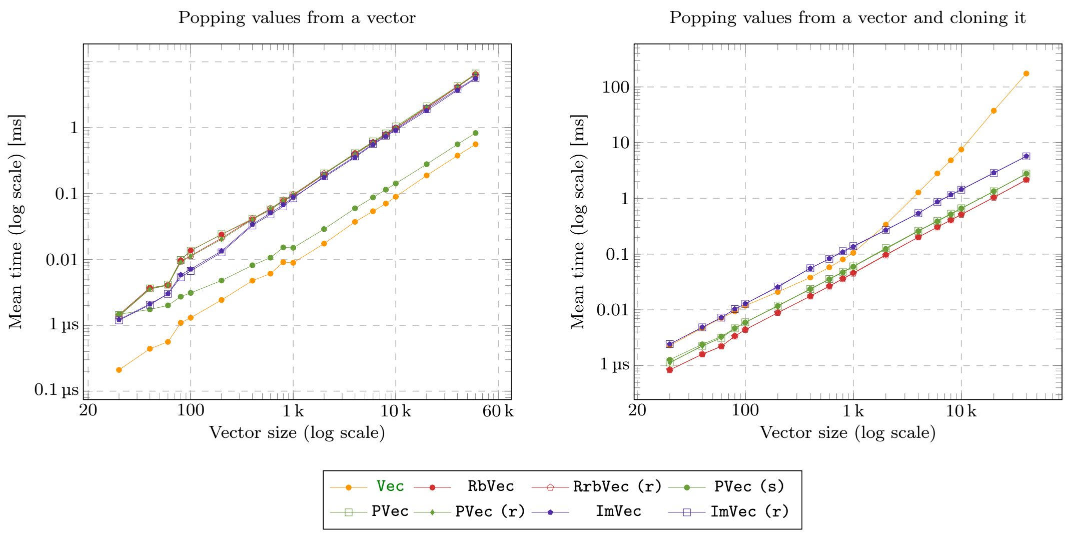


Figure 5.7: Benchmarking results of popping values

Results In the test without clone, `ImVec` is slightly faster than `RbVec` and `RrbVec` by 1.29. The difference between balanced and relaxed trees is only 1.02, showing that the cost of managing the relaxed tree capacity is not expensive.

`PVec` is the second fastest vector with `Vec` being 1.99 faster on average. When cloned, `PVec` switches the representation and becomes as fast as `RbVec`.

On the other hand, `ImVec` is the slowest tree-based vector when cloned, but still faster than `Vec`, especially for large sizes.

5.1.5 Appending

The append operation merges contents of one vector into another. One of the advantages of `RrbTree` is the relatively low cost of append, that is $\mathcal{O}((m^2 \cdot log_m(n))$, in comparison to $\mathcal{O}(max(a, b))$ of `Vec`. The objective is to confirm this assumption experimentally.

Naive vs. relaxed append algorithm The `RbTree`-based vector uses a naive concatenation algorithm that moves values from one vector to another. `RrbTree`, on the other hand, merges and re-balances two trees, which is faster in theory. Due to the hardware design specifics, this might not be true for all vector sizes. Thus, benchmarks will reveal how different algorithms perform depending on the size of concatenated vectors.

Appending vectors The setup routine prepares a collection of vectors, where each consecutive vector is bigger than the previous. The total size of all prepared vectors adds up to the problem size N. The benchmark is parameterized over the vector size, which will be in the [20, 1 M] range.

Each vector is created by a combination of append and push operations. This way `PVec` and `RrbVec` will be forced to use `RrbTree` for internal representation, while `RbVec` will remain balanced. `Vec` remains flat and does not depend on the type of operation used to add values to it.

The benchmark function iterates over generated vectors and appends them into a vector defined as a local variable.

Results Based on the results, the naive copying of `Vec` is the fastest concatenation algorithm up to the problem size of 40 k. However, after surpassing 40 k it quickly degenerates and loses to `RrbVec` by a factor of 6.53, with a maximum difference of 12.85 for the size of 1 M.

Even though not for all input sizes, we can see that `RrbVec`'s concatenation algorithm scales better and eventually outperforms `Vec`.

`ImVec` catches up to `Vec` only at the size of 400 k, still being slower than `RrbVec` by a factor of 1.29 at that point.

Balanced vs. relaxed For the input size up to 2 k, `RbVec` and `RrbVec` show similar results. The simplicity of naive concatenation used by `RbVec` is sufficient to be as fast as `RrbVec` due to the small problem size.

This, however, drastically changes after the 2 k size, where `RbVec` continuously degrades, showing the worst results among all vectors. The performance difference between `RbVec` and `RrbVec` at the size of 1 M is staggering 145.59.

As for `PVec`, it follows the curves of `Vec` and `RrbVec` because of the dynamic representation, as expected.

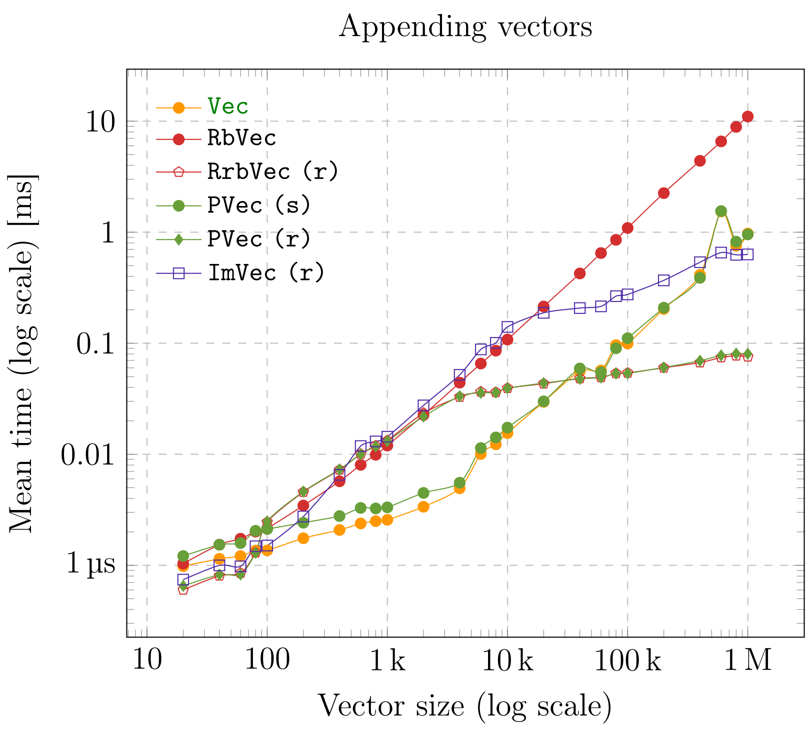


Figure 5.8: Benchmarking results of appending vectors

5.1.6 Splitting

The split operation slices a vector into two parts at the given index. The `RrbTree`'s algorithm theoretically can achieve good performance by avoiding unnecessary copying. However, due to its complexity, it might be outperformed by naive copying for small-sized vectors.

Splitting vectors The test itself anticipates a prepared vector, which is generated in the setup routine. To evaluate both balanced and relaxed variants, it generates them either by using appending or pushing. The vector sizes are within [128, 200 k].

Once a vector is generated, the benchmark function enters the loop with the condition that the vector needs to contain more than 64 elements. In the loop, a vector is split at index 64, the result of which is assigned back to a variable.

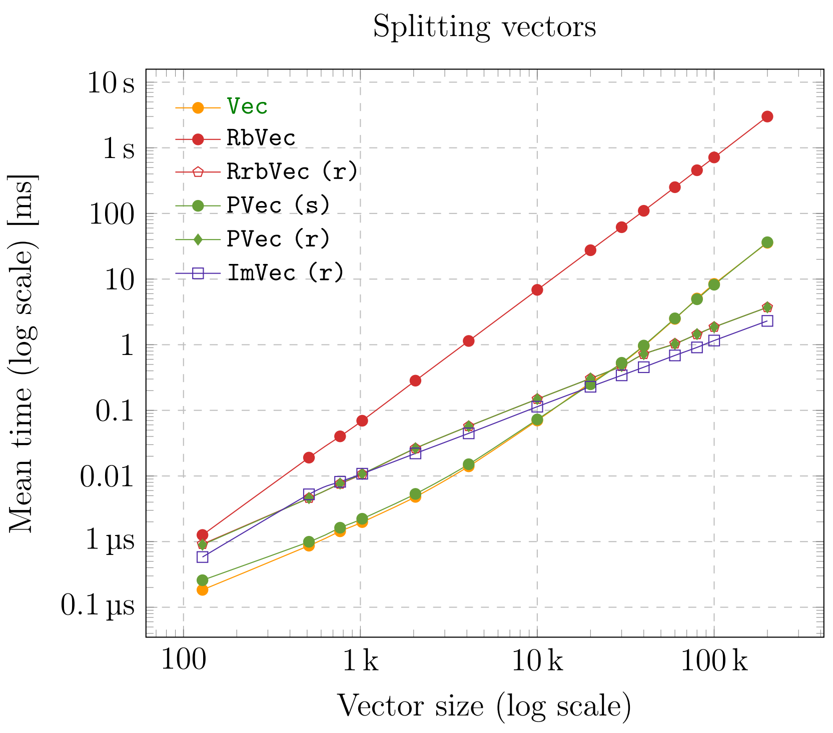


Figure 5.9: Benchmarking results of splitting vectors

Essentially, a vector is being truncated at the front by 64 elements, until it is small enough for the loop to exit.

Results The performance advantage of `Vec` over `RrbVec` is 3.24 up to the size of 20 k, after which it degrades and gets slower than `ImVec` and `RrbVec` by 5.95. The difference is more significant for `RbVec` at the factor of 65.65 for the size of 400 k.

5.2 Parallel benchmarks

The benchmarks were executed against `Vec` and vectors from ***pvec-rs***, where `RbVec`, `RrbVec`, and `PVec` were compiled with the threadsafe reference counting pointer – `Arc`.

At the time of writing, `ImVec` does not implement the `IntoParallelIterator` trait, and therefore, is not included in the parallel benchmarks.

The benchmarks were parameterized over two dimensions: the vector size and the number of threads. To see whether parallelism is beneficial, each benchmark has an analogous, sequential implementation used as the baseline.

The results are discussed from the perspective of:

- Scalability of the tree-based vectors.
- The impact of the relaxed append and split operations.

The results are presented in the form of a three-dimensional graph, where the x and y-axis correspond to the problem size and number of used threads, while z stands for the mean runtime.

5.2.1 Adding elements of two vectors

Given two equally sized vectors of integers, the test function adds values at the corresponding indices and returns a new instance of a vector with results. The benchmark is subdivided into three steps:

1. Transform two vectors into a single sequence of value pairs by merging their parallel iterators.
2. Add values of the emitted key-value pair into a single integer.
3. Reduce individual sums into a vector of results.

The setup routine prepares two vectors of integers in the [0, N] problem size range.

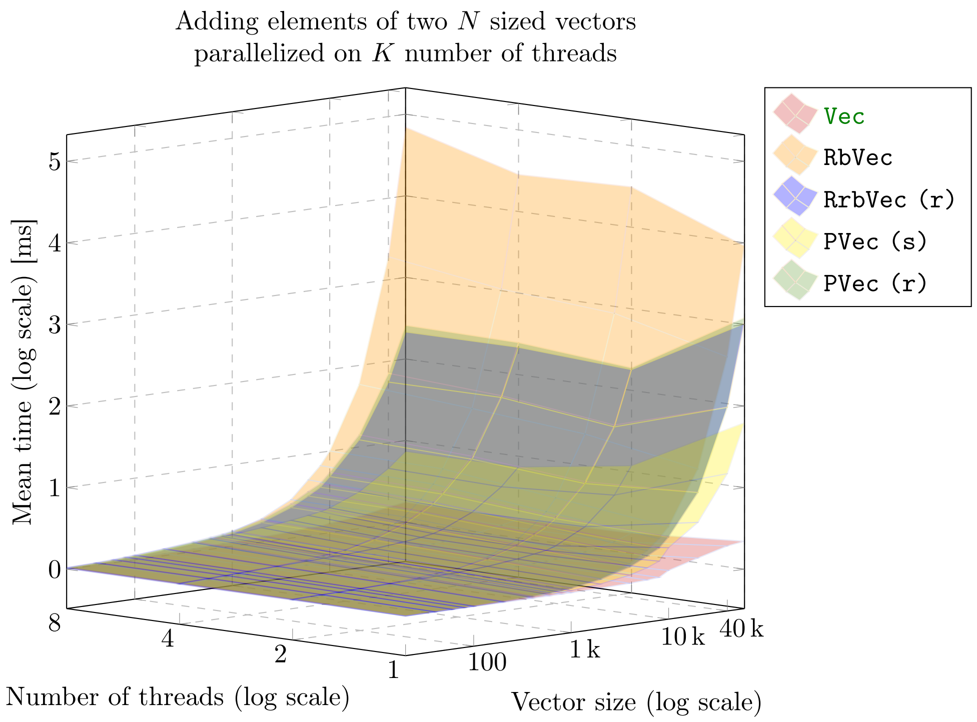


Figure 5.10: Adding elements of two N sized vectors parallelized on K number of threads

Results `RbVec` and `RrbVec` show nearly identical results in sequential benchmarks. This is expected, as append and split operations that create relaxed nodes were not used. Hence, `RrbVec` remains balanced throughout the test, and is backed by the same representation of `RrbTree` as `RbVec`. Both variants are consistently slower compared to `Vec`, with a difference of 3.2-3.5 on average.

When executed in parallel, `RrbVec` starts outperforming `RbVec` at the size of 1024 elements. The reason why difference becomes apparent after surpassing that size is that the concatenation algorithm used in this project produces balanced `RbTree` when the height of the tree does not exceed two levels. With the branching factor of 32, the capacity of the tree of two levels is equal to 1024.

As the vector size grows, Rayon performs more slices to achieve optimal vector size per a single thread. This, in turn, results in a higher number of concatenations necessary to combine execution results. Since `RbVec` has the naive implementation of append and split, its performance degrades with the input size growth. The difference in execution time, depending on size, falls into the factor of 1.0-2.3 range.

To keep available threads busy, Rayon divides the available pool of work into smaller pieces. Hence, the growing number of threads increases the performance gap between `RbVec` and `RrbVec` even further, as split and append are used more frequently. In the test with 2, 4, and 8 threads, `RbVec` is slower than `RrbVec` by a factor of 1.0-2.3, 1.0-2.8, and 1.1-2.12 correspondingly.

Even though `RrbVec`'s append and split operations are faster for large-sized vectors, `Vec` showed the best results in all tests. It is important to keep in mind that appends and splits constitute only a small number of all operations used in this test. Operations such as push and get, which were extensively used in this benchmark, are still faster for `Vec`. Thus, the closest runner up – `PVec`, is slower by a factor of 1.8-1.9 and 1.6-1.7 in the sequential and 4-threaded parallel benchmarks correspondingly.

The effect of parallelism The sequential variant of the benchmark outperformed all subsequent parallel tests. This can be explained by the overhead induced by the distribution of work between multiple threads, which outweighs the benefits of solving a relatively simple problem in parallel. It is an acceptable trade-off, as the purpose of the benchmark is to observe how different vectors perform compared to each other, rather than tuning the parallel benchmark for optimal performance.

5.2.2 Check if a word is a palindrome

The benchmark checks whether a word is a palindrome and annotates it with a boolean flag. As input, we are using a list of English words consisting only of alphabetic characters. The computation stages are the following:

1. Transform the given vector of words into a parallel iterator.

2. Return a tuple of the word and the flag indicating if the word is a palindrome.
3. Reduce the results to a new instance of a vector of tuples.

The benchmark setup The dictionary file contains 370103 words. The benchmark is parameterized over the number of threads and words. The data is loaded into memory only once, and before each run, the setup routine copies N words and passes them to the test as a vector.

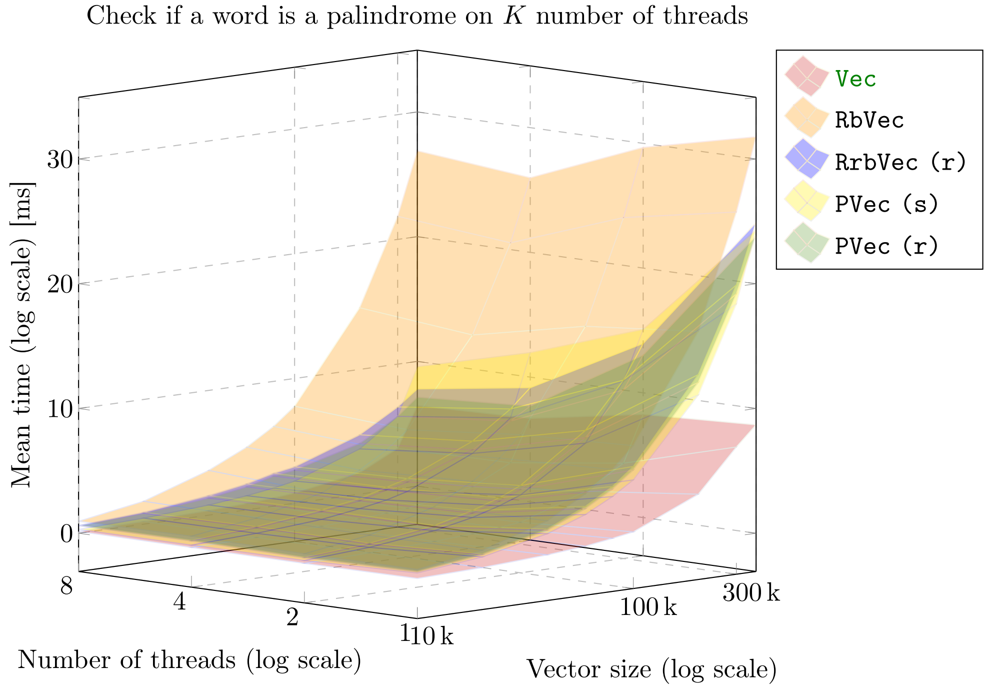


Figure 5.11: Check if a word is a palindrome on K number of threads

Results As expected, `RbVec` and `RrbVec` are equally fast in the sequential test, as both of them are represented by a balanced `RbTree`. When the benchmark is parallelized, `RrbVec` gains advantage due to its efficient slice and concatenation operations. The difference between variants grows along with the increasing count of threads. Specifically, it is 1.5-2.0 for the test run with two threads, and 1.7-2.5 for four threads.

The increase in the thread number causes a higher number of vector splits and concatenations. Thus, the bigger the problem size and the thread number is, the more advantages `RrbVec` provides, demonstrating performance results comparable to `Vec`.

The effect of parallelism The performance gap between the sequential and parallel runs closes with the thread number increasing, and it is less noticeable compared to Section 5.2.1. It is due to the palindrome check being complex enough to benefit from parallelizing the work.

5.3 Memory benchmarks

The goals for the memory tests are the following:

- Measure the memory footprint of the tree-based and standard vectors.
- Evaluate the effectiveness of the structural sharing when cloning a vector.

5.3.1 Building a vector

This benchmark is expected to reveal the memory overhead of using a tree instead of the contiguous memory block as its height, and the node count grows. Given the size N as a parameter, this benchmark builds a vector by pushing N values into it. The problem size range is [6 k, 2 M].

Results The tree-based vectors use the amount of memory proportional to their size. The `RbVec` and `PVec` types, however, consistently use 2.2 times more memory compared to `Vec`. On the other hand, `ImVec` shows excellent results reviling `Vec` with a marginal difference of 1.18 on average.

5.3.2 Updating and cloning a vector

The test builds a new vector of size N, runs a loop from 0 to N, clones a vector, and updates a value at the given index. All cloned instances are accumulated in another vector to observe how well structural sharing helps to save memory. The vector sizes are in [1 k, 60 k] range.

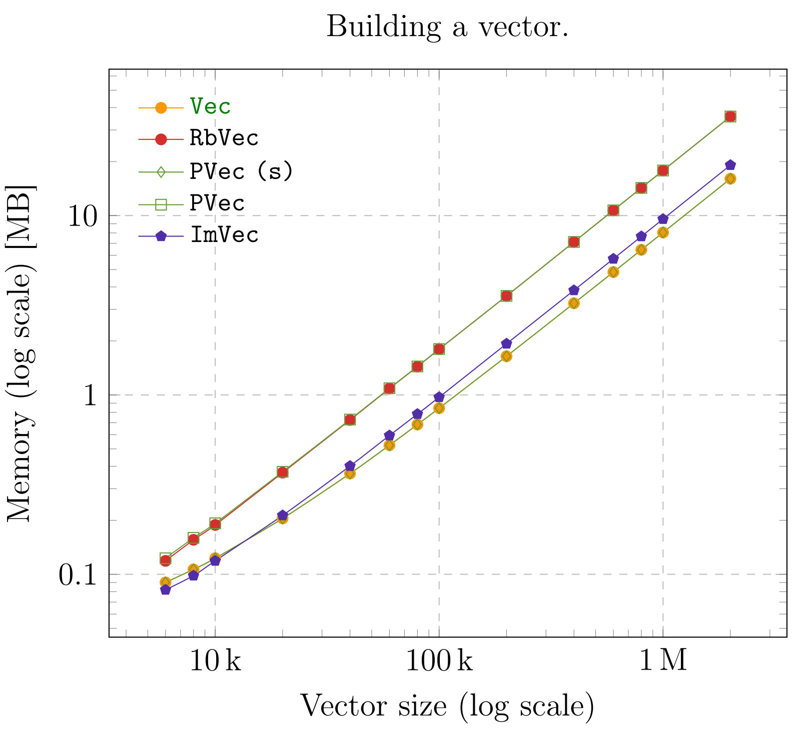


Figure 5.12: The memory footprint of building a vector

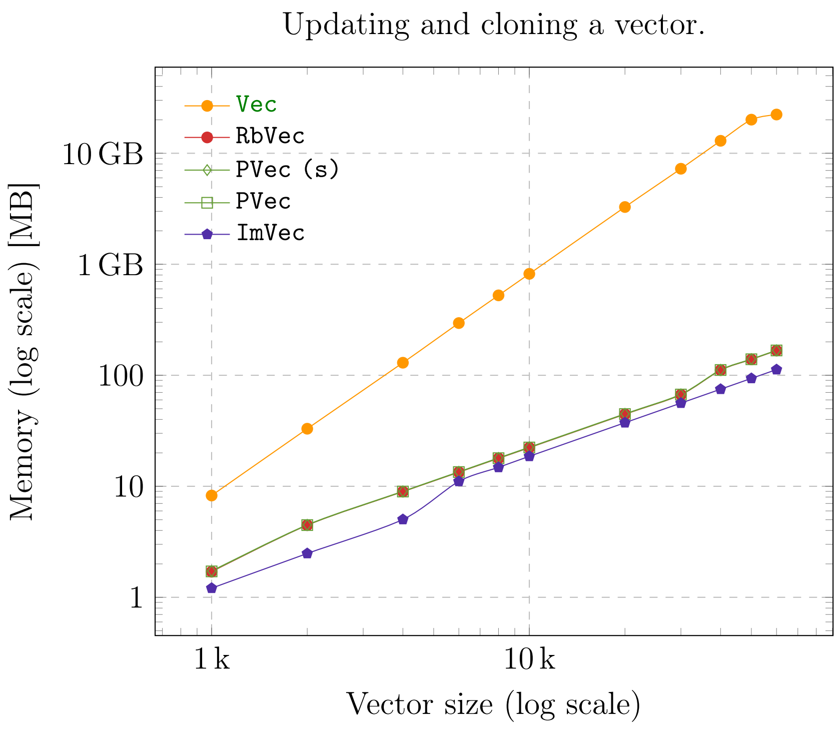


Figure 5.13: The memory footprint of updating and cloning a vector

Results The results demonstrate the strongest advantage of the tree-based vectors – the ability to preserve memory by structural sharing. The tree-based vectors show similar results, with `ImVec` being slightly ahead of `PVec` by consuming 0.71 less memory. `PVec` and `ImVec` are slightly more than a 100 MB large at the peak size. On the other hand, `Vec` does not scale well, consuming more than ten gigabytes of memory at the benchmark input of 60 k.

Chapter 6

Conclusions and future work

In this final chapter, I will look at the state and future of the ***pvec-rs*** project, and how the ideas explored in this thesis can be continued further.

6.1 Reflecting on contributions

This project explores and blends ideas at the intersection of persistent data structures and unique features of Rust to contribute a vector implementation that delivers good performance for all operations, including clone. It makes ***pvec-rs*** a viable alternative in applications where the fast clone operation is critical.

The list of vectors includes `RbVec`, `RrbVec`, and `PVec`, all of which are based on `RrbTree`.

6.1.1 Balanced vs. relaxed

The advantage of the relaxed `RbTree` is the fast appends and splits. `RrbVec` demonstrates significantly better performance for those operations compared to `RbVec`, and even outperforms `Vec` for the large-sized vectors.

Frameworks for parallelism, such as ***rayon***, take advantage of multiple threads by dividing the work between them. Vectors are subdivided by using the split operation, after which the results are collected back by concatenating them. Therefore, `RrbTree`'s fast append and split operations are critical for achieving good performance in parallel use cases.

The overhead of relaxation is present in other operations, but it is not significant enough to outweigh the benefits. Also, constraints are relaxed only when append or split is used, meaning that one does not have to pay the cost of the abstraction before using it.

6.1.2 Pay only for the features you use

The project draws inspiration from one of the Rust key features – zero-cost abstractions. As demonstrated in Chapter 5, `PVec` starts as an ordinary, standard vector that delivers great performance for the core operations. When cloned, it employs a technique introduced in this project named *spilling*, which transitions the vector from the flat to the tree-based representation. When transitioned to a balanced `RrbVec`, `PVec` offers practically $\mathcal{O}(1)$ performance for all operations, including cloning, enabling patterns that extensively rely on copying.

Dynamic representation Tree-based vectors are very cheap to clone, but their core operations, even though very fast, do not match the performance of the standard vector due to the nuances of hardware architecture. The dynamic representation aims to offset this cost by using standard vector, switching to the tree-based representation only when cloned.

Results show that dynamic representation effectively improves the performance of all `PVec` operations, except appending and splitting for large-sized vectors, where tree-based vectors have an advantage. However, since `PVec` is essentially an additional abstraction layer, it introduces a marginal overhead over the pure variants of its representations – `Vec` and `RrbVec`.

Unique access In the paper Improving RRB-Tree Performance through Transience [11], the author mentions that the correct use of transient data types can be checked during compile-time in languages that use linear types [18], such as Rust.

This project implements unique access optimization, that is somewhat similar to transience, but is not entirely the same. For example, transients in Clojure[1] are created by calling a special function – *transient*, and converted back to persistent using *persistent!*. Transient types are also thread-local in Clojure, meaning that they cannot be modified outside of the thread where the transient was created.

In Rust, a vector can be considered transient when it is accessed through a mutable reference without calling a special function such as *transient*. The Rust's compiler ensures that the mutable reference is *unique* and that the operation is safe. With that knowledge, the program can proceed to update vector in-place without creating copies – transitively. Rust also allows moving objects between threads, so a vector instantiated on one thread can be updated on another.

The unique access optimization ensures that the data structure can be mutated in-place only when it is safe. If a vector is cloned before being updated, copy-on-write semantics of `Rc` will enforce path-copying leaving the original untouched.

Benchmarks for mutative operations, such as push, pop, and update that included tests with and without clone showed that updating a data structure in-place was noticeably faster.

Additionally, a "mutable" interface for `PVec` that makes unique access optimization possible, also makes the API of `PVec` identical to `Vec` and conventional to Rust.

Idiomatic, ergonomic Rust interface The idiomatic and convenient interface of ***pvec-rs***, identical to the standard one, simplifies the library's integration into existing codebases. A side effect of this design is that both types of vectors can be used interchangeably in a generic manner. For example, the vector type can be substituted during compile-time using feature flags without changing the source code.

Thread-safety is also an optional feature that can be enabled when compiling. This way, developers do not have to pay the cost of using the parallel vector

[1] https://clojure.org/reference/transients

features in sequential applications.

6.2 Implementation state

While the ***pvec-rs*** core outlined in the thesis is complete, some features and optimizations were left out of the scope. This section describes features and ideas that can be explored further.

6.2.1 Supporting all operations of Vec

The API surface of the ***pvec-rs*** does not expose the same set of methods as the standard vector does. Available methods are listed in Section 3.1.2.

Having efficient appending and splitting allows us to implement several other operations, such as inserting or deleting an element at any index. The complexity of these operations is bound by the complexity of the discrete operations used to implement them. Thus, uniform performance characteristics across core operations are critical for achieving good all-around performance for the general-purpose vector.

For example, element insertion at the given index can be implemented by splitting the vector at the given index, pushing a new element into the left sub-vector, and then concatenating two sub-vectors back together.

Therefore, the operations that can be implemented by combining or re-using core operations were intentionally left as future work due to the time constraints.

6.2.2 Improving the dynamic representation

Automatically switching to the flat representation A distinct feature of `PVec` is the ability to start as a standard vector and then switch to `RrbVec` when cloned. However, there is no mechanism to switch back to the flat representation, for example, when all cloned instances are destroyed.

One way to achieve this is by flattening the `RrbTree` into a standard vector when the underlying tree is not shared with any clones. One could use Rust's destructors to observe when `PVec` clones go out of the scope. The challenge is to be able to say when the tree is not shared anymore. A brute force approach would be traversing the tree and counting references, but obviously, it is very in-efficient.

An annotated example of this optimization in use is provided in Listing 22.

```rust
let mut vec_1 = PVec::new();
// ^ start as a standard Vec

for i in 0..512 {
    vec_1.push(i);
}

vec_1 = vec_1.clone();
// ^ force switch to RrbVec

let vec_2 = vec_1.clone();
{ // <-- moving vec_2 to the new scope
    vec_2
} // <-- vec_2 goes out of the scope and is
  // destroyed, vec_1 switches back to standard Vec

// execution continues
```

Listing 22: An example of switching back to the flat representation

Starting as an array allocated on the stack The flat vector representation is efficient because of its cache-friendly memory layout. Since the vector size is not known at the compile-time, it is allocated on the heap. In comparison to the stack, heap allocation is more complex and expensive as it requires the memory allocator to track and manage allocated blocks of memory. Additionally, heap-allocated objects are more likely to cause cache invalidation, as CPU will have to reach a memory segment that potentially is located far outside of its caches.

In an attempt to improve the cache locality properties of the standard vector, authors of the ***smallvec***[2] library introduced a vector implementation that stores a certain number of elements on the stack, and falls back to the heap for larger sizes.

The dynamic representation can be extended with the new representation type that allocates vector on the stack. The vector first will be allocated on the stack, then spill to the heap when exceeding a certain size threshold, and switched to `RrbVec` when cloned.

One has to be cautious in implementing this optimization. Internally, `PVec` is backed by the `Representation` enum, and in Rust, the enum size is bounded by its largest variant. The variant that holds the stack-allocated buffer will quickly supersede `Flat` and `Tree` representations if set to be sufficiently large, increasing the overall memory footprint of `PVec`.

6.2.3 Focus and display optimizations

The notion of *focus* was introduced in Scala's immutable vector implementation and was further studied in [17]. Instead of keeping track only of the vector *tail*, the focus is generalized to work with the leaf node, which was last modified. The basis for this is the principle of spatial locality, a heuristic that assumes that collocated elements are more likely to be accessed one after another.

Since the vector has to be thread-safe, focus either has to be modified when the vector itself is modified or to be protected from the concurrent access. The latter comes at additional performance and maintenance costs.

Display is a way to keep track of the entire tree branch, from the root to a leaf, where a leaf is the *focused* node. Introducing display to `RrbTree` requires additional coordination when the tree is modified.

Limitations of Rust The strict ownership and borrowing rules introduce additional complexity in implementing the *display*. It is forbidden to acquire and keep mutable references to the node and its children simultaneously. That

[2]https://docs.rs/smallvec/1.2.0/smallvec/

is a necessary property for display, which essentially is a stack of pointers to nodes that form a path from the root to the leaf nodes.

Alternatively, rather than keeping a stack of mutable pointers, one could use `Rc`. The side effect of this choice is that ownership of `Rc` demands to clone. This, in turn, increments the reference count. When the reference count is bigger than one, any attempts to acquire a mutable pointer will result in the clone of the underlying value. Since display and focus are updated only when the vector itself is modified, it will result in path-copying every time.

The second option is to use the interior mutability pattern in Rust, or `RefCell`. `RefCell` is a container that enforces compile-time rules of the borrow checker at runtime. Offsetting these checks helps to implement the display, but also adds overhead to every other operation, as all tree nodes have to be decorated with `RefCell`.

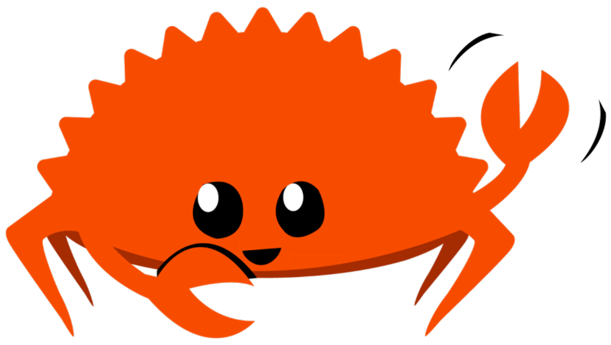
Even though *display* and *focus* seem to optimize some specific use cases potentially, the additional implementation complexity could cause more bugs and harm performance of other operations making `RrbTree` less efficient as a general-purpose data structure.

Especially in Rust, the options listed above require either using the unsafe subset of the language features, sacrificing the simplicity, and possibly the reliability and performance. Adding *focus* and *display* to `RrbVec` is therefore left as future work.

6.3 Towards the library of persistent data structures for Rust

Vector is only one of many other general-purpose data structures provided by the Rust standard library, such as `LinkedList`, `HashMap`, `HashSet`, and others. The ideas discussed in this thesis can be used to implement persistent variants of those data structures. For example, the hash array mapped tries [1] can be used as a foundation for `HashMap`.

In fact, there are other projects that implement persistent collections for Rust today, such as ***imrs***[3] and ***rpds***[4]. Even though they do not offer the same optimizations and interface as ***pvec-rs***, they are a viable alternative for someone who needs a wider selection of persistent data structures today.



[3] https://docs.rs/im/14.3.0/im/
[4] https://docs.rs/rpds/0.7.0/rpds/

Appendix A

RRB-tree algorithms

A.1 Rebalancing algorithm

```
function REBALANCE(left, middle, right)
    height ← MAX($left_{height}$,$middle_{height}$,$right_{height}$)
    root, subtree, node ← CREATENODE(height)
    for mergedNode in left + middle + right do
        if $node_{len} = 0$ and $mergedNode_{len} == m$ then
            CHECKSUBTREE(root, subtree)
            subtree[$subtree_{len}$] ← mergedNode
            $subtree_{len}$++
        else
            for childNode in mergedNode do
                if $node_{len} = m$ then
                    CHECKSUBTREE(root, subtree)
                    subtree[$subtree_{len}$] ← node
                    $subtree_{len}$++
                node[$node_{len}$] ← childNode
                $node_{len}$++
    CHECKSUBTREE(root, subtree)
    if $node_{len}$ != 0 then
        subtree[$subtree_{len}$] ← node
        $subtree_{len}$++
    if $subtree_{len}$ != 0 then
        root[$root_{len}$] ← subtree
        $root_{len}$++
    return root

procedure CHECKSUBTREE(root, subtree)
    if $subtree_{len} = m$ then
        root[$root_{len}$] ← subtree
        $root_{len}$++
```

Listing 23: Rebalancing algorithm for RRB-tree

Appendix B

Tail optimization for persistent vectors

In practice, changes are often applied to the end or *tail* of the data structure. The stack is designed for such use cases, by offering the $\mathcal{O}(1)$ performance for the push and pop operations. Even though RRB-tree has similar performance characteristics, its push and pop implementations include pesky constant factors in the form of *radix search* and *path copying* algorithms.

The *tail* optimization is intended to offset this cost by reducing the count of the RRB-tree accesses. Instead of adding or removing elements one by one, changes are batched in the array of size m. This array could be thought of as a leaf node that will be attached to the tree only when it is full.

B.1 Optimizing the push operation

As demonstrated in Listing 24, the new value is set into a cloned tail at the $\text{tail}_{\text{size}}$ position. Since the tail is the rightmost leaf node, its size can be used as an index for the new value. If the tail is full, it is pushed into the tree and replaced with an empty tail.

```
function Push(vec, value)
    newTail ← Clone($vec_{tail}$)
    newTail[$tail_{size}$] ← value
    $newTail_{size}$ ← $tail_{size} + 1$
    newRoot ← $vec_{root}$
    if $newTail_{size}$ = m then
        newRoot ← Push($vec_{root}$, newTail)
        newTail ← CreateNode
    return CreateVec(newRoot, newTail)
```

Listing 24: Tail optimization for persistent vector's push operation

B.2 Optimizing the pop operation

Since a tail might contain values, pop has to remove them first before modifying the RRB-tree. If the tail is empty, it will be replaced with the rightmost leaf of RRB-tree. See Listing 25 for detailed steps.

```
function Pop(vec)
    newTail ← Clone($vec_{tail}$)
    newRoot ← *NIL*
    value ← newTail[$newTail_{size}$ - 1]
    $newTail_{size}$ ← $newTail_{size}$ - 1
    if $newTail_{size}$ = 0 then
        newRoot, newTail ← Pop($vec_{root}$)
    else
        newRoot ← $vec_{root}$
    return CreateVec(newRoot, newTail)
```

Listing 25: Tail optimization for the persistent vector's pop operation

B.3 Adapting the update and radix search operations

Changes for both update and radix search are very similar, with the difference that update has to ensure that the original version of vector stays unmodified.

The radix search implementation has to take into account that some of the values can be in the tail. A value is located within the tree if the key is less than the tree size. In this case, the search process is delegated to RRB-tree. Otherwise, the index for value in the tail is calculated by subtracting the tree size from the key.

```
function UPDATE(vec, key, value)
    root ← $vec_{root}$
    tail ← $vec_{tail}$
    if key < $root_{size}$ then
        newRoot ← UPDATE(root, key, value)
        return CREATEVEC(newRoot, tail)
    else
        newTail ← CLONE(tail)
        newTail[key - $root_{size}$] ← value
        return CREATEVEC(root, newTail)
```

Listing 26: Using the tail optimization in the update operation

```
function RADIXSEARCH(vec, key)
    root ← $vec_{root}$
    tail ← $vec_{tail}$
    if key < $root_{size}$ then
        return RRBTREERADIXSEARCH(root, key)
    else
        return tail[key - $root_{size}$]
```

Listing 27: Using the tail optimization in the radix search operation

List of Listings

List of Figures

List of Tables

Bibliography

[1] Philip Bagwell. “Ideal Hash Trees”. In: (2001).

[2] Philip Bagwell and Tiark Rompf. “RRB-Trees: Efficient Immutable Vectors”. In: (2011).

[3] The Cilk Project Developers. *The Cilk Project*. URL: http://supertech.csail.mit.edu/cilk/.

[4] Ulrich Drepper. “What every programmer should know about memory”. In: *Red Hat, Inc* 11 (2007), p. 2007.

[5] James Driscoll, Neil Sarnak, Daniel Sleator, and Robert Tarjan. “Making Data Structures Persistent”. In: *J. Comput. Syst. Sci.* 38.1 (1989), pp. 86–124.

[6] James Driscoll, Daniel Sleator, and Robert Tarjan. “Fully Persistent Lists with Catenation”. In: *J. ACM* 41.5 (1994), pp. 943–959.

[7] Linus Färnstrand. “Parallelization in Rust with fork-join and friends: Creating the fork-join framework”. MA thesis. Chalmers University of Technology, 2015.

[8] Andy Georges, Dries Buytaert, and Lieven Eeckhout. “Statistically rigorous Java performance evaluation”. In: *ACM SIGPLAN Notices*. Vol. 42. 10. ACM. 2007.

[9] Rich Hickey. “The Clojure programming language”. In: *Proceedings of the 2008 Symposium on Dynamic Languages, DLS 2008, July 8, 2008, Paphos, Cyprus*. 2008, p. 1.

[10] Steve Klabnik, Carol Nichols, et al. *The Rust Programming Language — Second Edition*. No Starch Press, 2018. URL: https://doc.rust-lang.org/book/second-edition/.

[11] Jean Niklas L’orange. “Improving RRB-Tree Performance through Transience”. MA thesis. Norwegian University of Science and Technology, 2014.

[12] Linus Nyman and Mikael Laakso. “Anecdotes: Notes on the History of Fork and Join”. In: *IEEE Annals of the History of Computing* 38.3 (2016), pp. 84–87.

[13] Chris Okasaki. *Purely Functional Data Structures*. Cambridge University Press, 1999.

[14] Eric Reed. “Patina: A formalization of the Rust programming language”. In: *University of Washington, Department of Computer Science and Engineering, Tech. Rep. UW-CSE-15-03-02* (2015).

[15] Neil Sarnak and Robert Endre Tarjan. “Planar point location using persistent search trees”. In: *Communications of the ACM* 29.7 (1986), pp. 669–679.

[16] Nathanael Schärli, Stéphane Ducasse, Oscar Nierstrasz, and Andrew Black. “Traits: Composable Units of Behavior”. In: pp. 248–274.

[17] Nicolas Stucki, Tiark Rompf, Vlad Ureche, and Phil Bagwell. “RRB vector: a practical general purpose immutable sequence”. In: *ACM SIGPLAN Notices*. Vol. 50. 9. ACM. 2015, pp. 342–354.

[18] Philip Wadler. “Linear types can change the world!” In: *Programming concepts and methods*. Vol. 3. 4. 1990.