

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

MASTER THESIS

---

# Relaxed Radix Balanced Trees as Immutable Vectors Scala

---

*Author:*

Nicolas STUCKI

*Supervisor:*

Vlad URECHE

*A thesis submitted in fulfilment of the requirements  
for the degree of Master in Computer Science*

*in the*

LAMP  
Computer Science

December 2014

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

# *Abstract*

School of Computer and Communications  
Computer Science

Master in Computer Science

## **Relaxed Radix Balanced Trees as Imutable Vectors Scala**

by Nicolas STUCKI

The Thesis Abstract is written here (and usually kept to just this page). The page is kept centered vertically so can expand into the blank space above the title too...

# Contents

<b>Abstract</b>	<b>i</b>
<b>Contents</b>	<b>ii</b>
<b>List of Figures</b>	<b>iv</b>
<b>List of Tables</b>	<b>vi</b>
<b>Abbreviations</b>	<b>vii</b>
<b>1 Introduction</b>	<b>2</b>
1.1 Main Section 1 . . . . .	2
1.1.1 Subsection 1 . . . . .	2
1.1.2 Subsection 2 . . . . .	2
1.2 Main Section 2 . . . . .	2
<b>2 Vector Structure</b>	<b>4</b>
2.1 Radix Balanced Vectors . . . . .	4
2.1.1 Tree structure . . . . .	4
2.1.2 Operations . . . . .	4
2.1.2.1 Apply . . . . .	4
2.1.2.2 Updated . . . . .	4
2.1.2.3 Additions . . . . .	4
Append and Prepend . . . . .	4
Concatenation and Insert . . . . .	4
2.1.2.4 Splits . . . . .	5
2.2 Parallel Vectors . . . . .	5
2.2.1 Splitter Iterator . . . . .	5
2.2.2 Combiner Builder . . . . .	5
2.3 Relaxed Radix Balanced Vectors . . . . .	5
2.3.1 Tree structure . . . . .	5
2.3.2 Operations . . . . .	5
2.3.2.1 Apply (get element at index) . . . . .	5
2.3.2.2 Updated . . . . .	5
2.3.2.3 Additions . . . . .	5
Append and Prepend . . . . .	5
Insert . . . . .	5

Concatenation . . . . .	5
2.3.2.4 Splits . . . . .	6
<b>3 Implementation and Optimizations</b>	<b>8</b>
3.1 Where does time go? . . . . .	8
3.1.1 Arrays . . . . .	8
3.1.2 Computing indices . . . . .	8
3.2 Displays . . . . .	9
3.2.1 As cache . . . . .	9
3.2.2 For transient states . . . . .	9
3.3 Builder . . . . .	10
3.4 Iterator . . . . .	10
3.5 Relaxing the Radix . . . . .	10
3.5.1 Displays . . . . .	10
3.5.2 Builder . . . . .	10
3.5.3 Iterator . . . . .	10
<b>4 Performance</b>	<b>11</b>
4.1 In practice: Running on JVM . . . . .	12
4.1.1 Cost of Abstraction and JIT Inline . . . . .	12
4.2 Measuring performance . . . . .	12
4.3 Generators . . . . .	12
4.4 Benchmarks . . . . .	12
4.4.1 Apply . . . . .	12
4.4.2 Concatenation . . . . .	12
4.4.3 Append . . . . .	12
4.4.4 Prepend . . . . .	12
4.4.5 Splits . . . . .	12
4.4.6 Iterator . . . . .	12
4.4.7 Builder . . . . .	12
4.4.8 Parallel split-combine . . . . .	12
4.4.9 Memory footprint . . . . .	12
<b>5 Testing</b>	<b>26</b>
5.1 Teststing correctness . . . . .	26
5.1.1 Invariant Assertions . . . . .	26
5.1.2 Unit tests . . . . .	26
5.2 Main Section 2 . . . . .	26
<b>6 Related Work</b>	<b>27</b>
6.1 RRB-Vectors in Clojure . . . . .	27
<b>7 Conclusions</b>	<b>28</b>

# List of Figures

2.1	Radix Balanced Tree Structure . . . . .	4
2.2	Radix Balanced Tree . . . . .	5
2.3	Concatenation example with blocks of size 4: Rebalancing level 0 . . . . .	6
2.4	Concatenation example with blocks of size 4: Rebalancing level 1 . . . . .	6
2.5	Concatenation example with blocks of size 4: Rebalancing level 2 . . . . .	6
2.6	Concatenation example with blocks of size 4: Rebalancing level 3 . . . . .	7
3.1	Accessing element at index 526843 in a tree of depth 5. Empty nodes represent collapses subtrees. . . . .	8
3.2	Radix Balanced Tree . . . . .	9
3.3	Radix Balanced Tree Transient state . . . . .	9
4.1	Time to execute 10k apply operations on sequential indices. . . . .	12
4.2	Time to execute 10k apply operations on random indices. . . . .	13
4.3	Time to execute 10k apply operations on sequential indices. Comparing performances for different block sizes and different implementation of the concatenation inner branch rebalancing (Copmlete/Quick). . . . .	13
4.4	Execution time for a concatenation operation on two vectors. In theory (and in practice) Vector concatenation is $O(left+right)$ and the rrbVector concatenation operation is $O(\log_{32}(left + right))$ . . . . .	14
4.5	Time to execute 256 append operations. This shows the amortized cost of the append operation. . . . .	15
4.6	Time to execute 256 append operations. This shows the amortized cost of the append operation. Comparing performances for different block sizes and different implementation of the concatenation inner branch rebalanc- ing (Copmlete/Quick). . . . .	16
4.7	Time to execute 256 prepend operations. This shows the amortized cost of the prepend operation. . . . .	17
4.8	Time to execute 256 prepend operations. This shows the amortized cost of the append operation. Comparing performances for different block sizes and different implementation of the concatenation inner branch re- balancing (Copmlete/Quick). . . . .	18
4.9	Execution time of take and drop. . . . .	19
4.10	Excecution time to iterate through all the elements of the vector. . . . .	20
4.11	Excecution time to iterate through all the elements of the vector. Com- paring performances for different block sizes and different implementation of the concatenation inner branch rebalancing (Copmlete/Quick). . . . .	21
4.12	Execution time to build a vector of a given size. . . . .	22

---

4.13 Execution time to build a vector of a given size. Comparing performances for different block sizes. . . . .	23
4.14 Benchmark on map and parallel map using the function ( $x \Rightarrow x$ ) to show the difference time used in the framework. This time represents the time spent in the splitters and combiners of the parallel collection (iterator and builder for the sequential version). . . . .	23
4.15 Benchmark on map and parallel map using the function ( $x \Rightarrow x$ ) to show the difference time used in the framework. This time represents the time spent in the splitters and combiners of the parallel collection. . . . .	24
4.16 Memory Footprint . . . . .	25

# List of Tables

# Abbreviations

<b>JIT</b>	<b>J</b> ust <b>I</b> n <b>T</b> ime
<b>RB</b>	<b>R</b> adix <b>B</b> alanced
<b>RRB</b>	<b>R</b> elaxed <b>R</b> adix <b>B</b> alanced



I

# Chapter 1

## Introduction

### 1.1 Main Section 1

#### 1.1.1 Subsection 1

#### 1.1.2 Subsection 2

### 1.2 Main Section 2

## I

## Chapter 2

# Vector Structure

### 2.1 Radix Balanced Vectors

#### 2.1.1 Tree structure

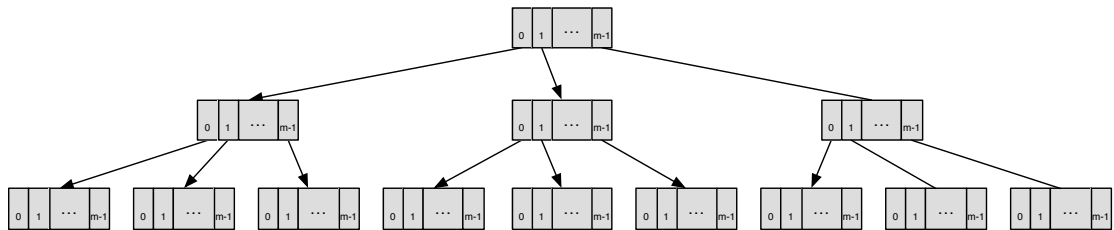


FIGURE 2.1: Radix Balanced Tree Structure

#### 2.1.2 Operations

##### 2.1.2.1 Apply

##### 2.1.2.2 Updated

##### 2.1.2.3 Additions

#### Append and Prepend

#### Concatenation and Insert

### 2.1.2.4 Splits

## 2.2 Parallel Vectors

### 2.2.1 Splitter Iterator

### 2.2.2 Combiner Builder

## 2.3 Relaxed Radix Balanced Vectors

### 2.3.1 Tree structure

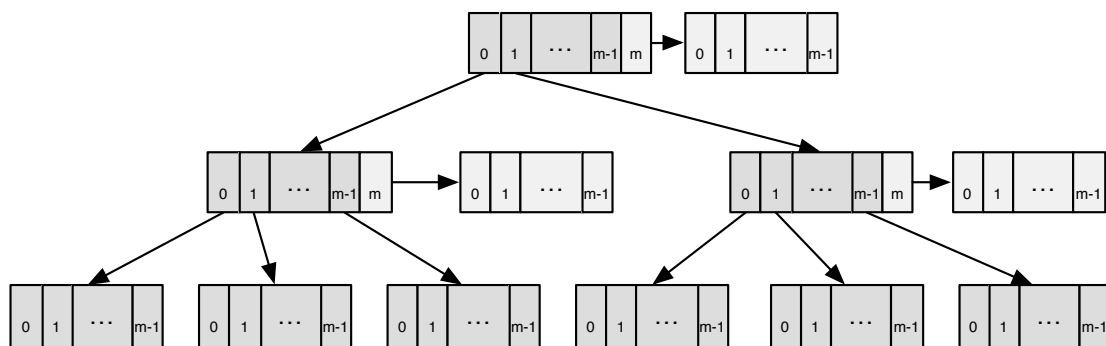


FIGURE 2.2: Radix Balanced Tree

### 2.3.2 Operations

#### 2.3.2.1 Apply (get element at index)

#### 2.3.2.2 Updated

#### 2.3.2.3 Additions

#### Append and Prepend

#### Insert

#### Concatenation

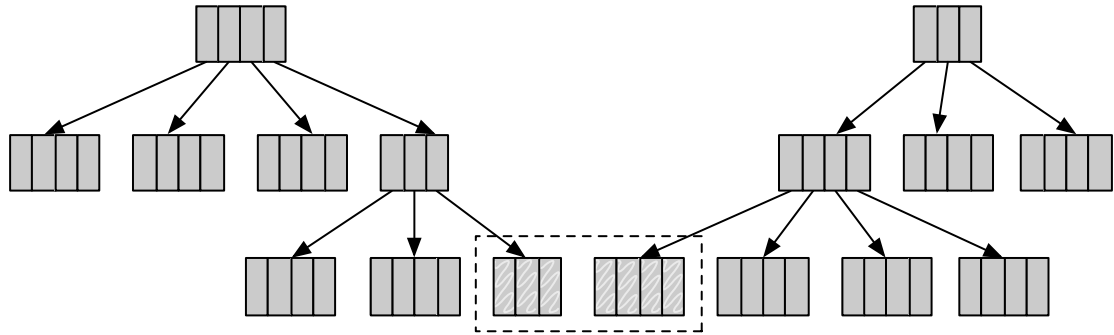


FIGURE 2.3: Concatenation example with blocks of size 4: Rebalancing level 0

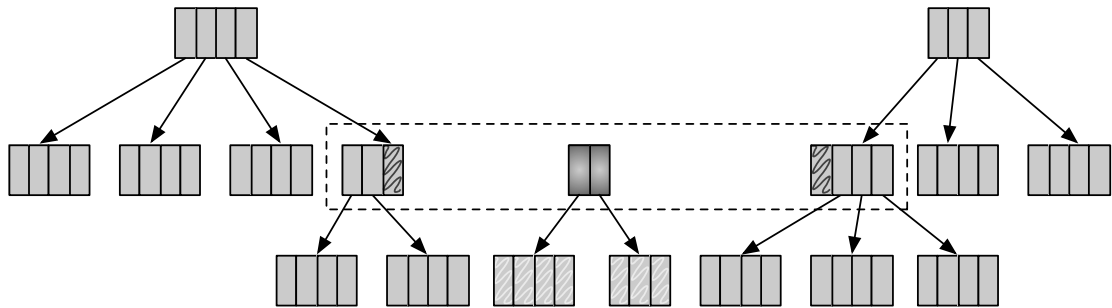


FIGURE 2.4: Concatenation example with blocks of size 4: Rebalancing level 1

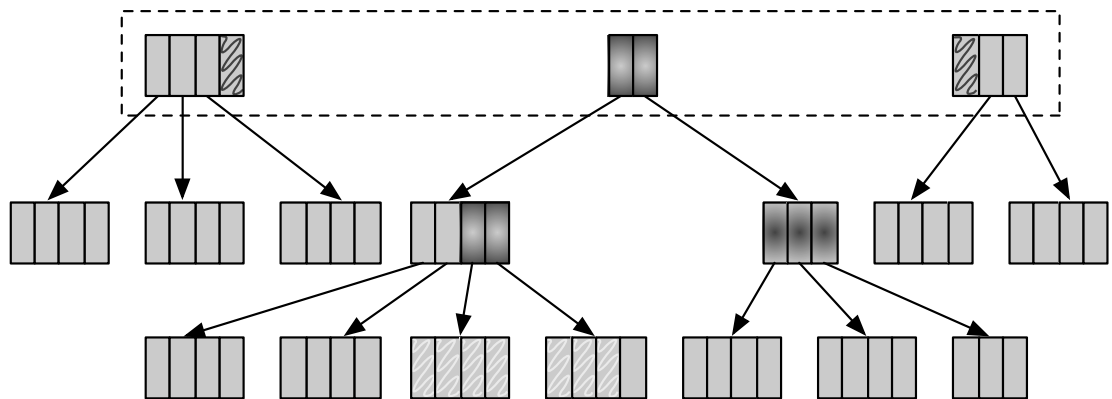


FIGURE 2.5: Concatenation example with blocks of size 4: Rebalancing level 2

### 2.3.2.4 Splits

I

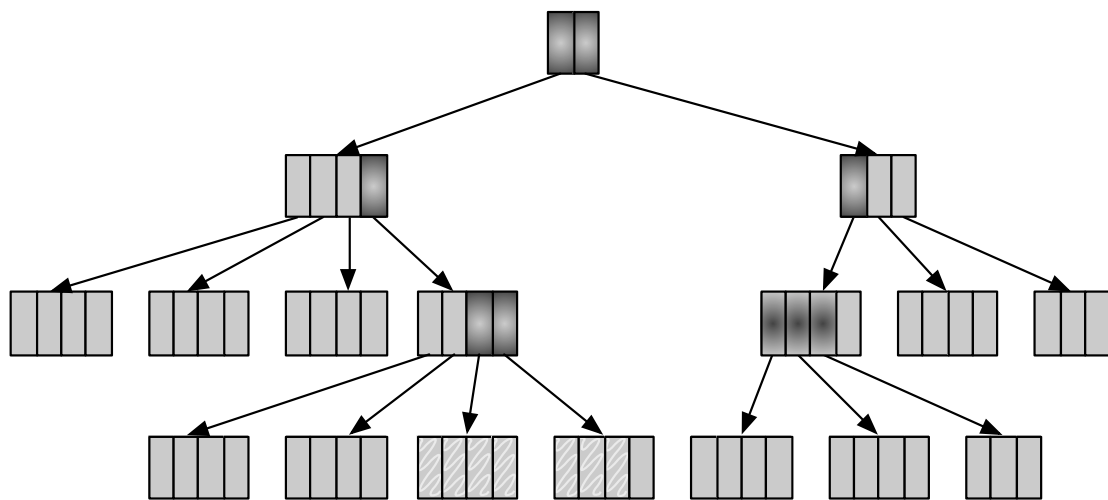
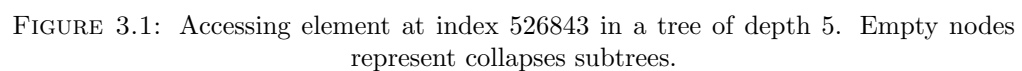


FIGURE 2.6: Concatenation example with blocks of size 4: Rebalancing level 3

# Implementation and Optimizations

### 3.1.2 Computing indices





## 3.2 Displays

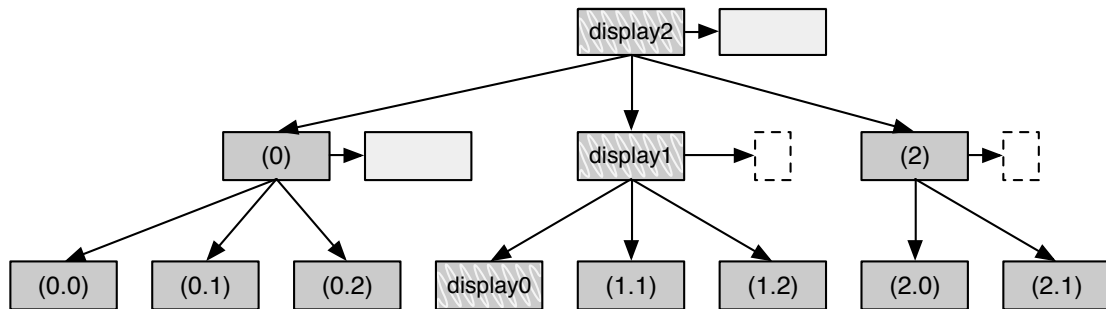


FIGURE 3.2: Radix Balanced Tree

### 3.2.1 As cache

### 3.2.2 For transient states

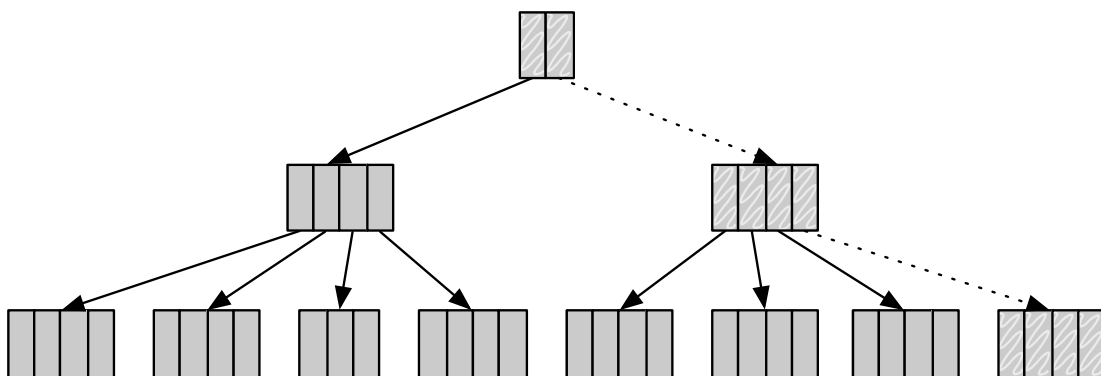


FIGURE 3.3: Radix Balanced Tree Transient state

### **3.3 Builder**

### **3.4 Iterator**

### **3.5 Relaxing the Radix**

#### **3.5.1 Displays**

#### **3.5.2 Builder**

#### **3.5.3 Iterator**

I



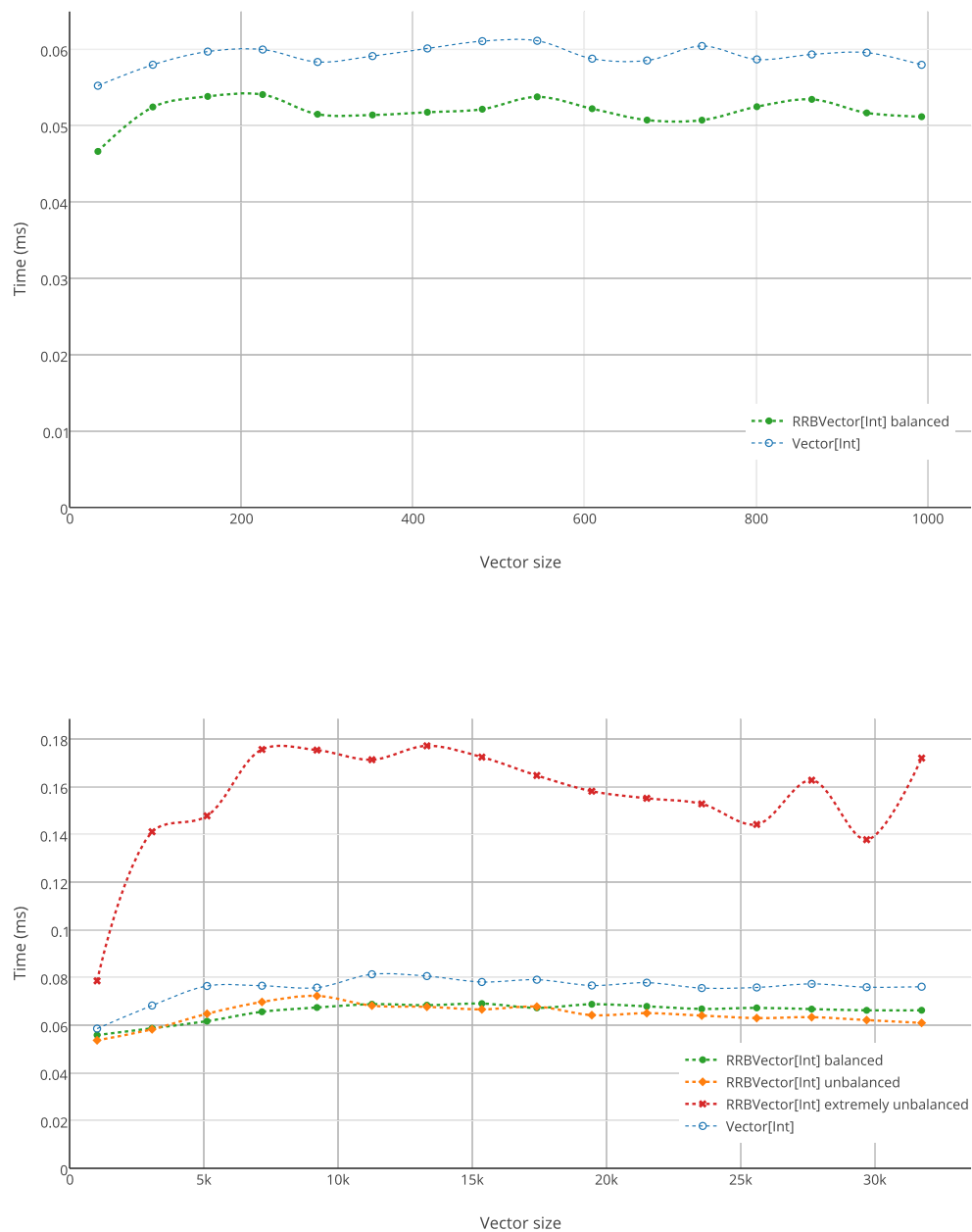


FIGURE 4.1: Time to execute 10k apply operations on sequential indices.

## Chapter 4

# Performance

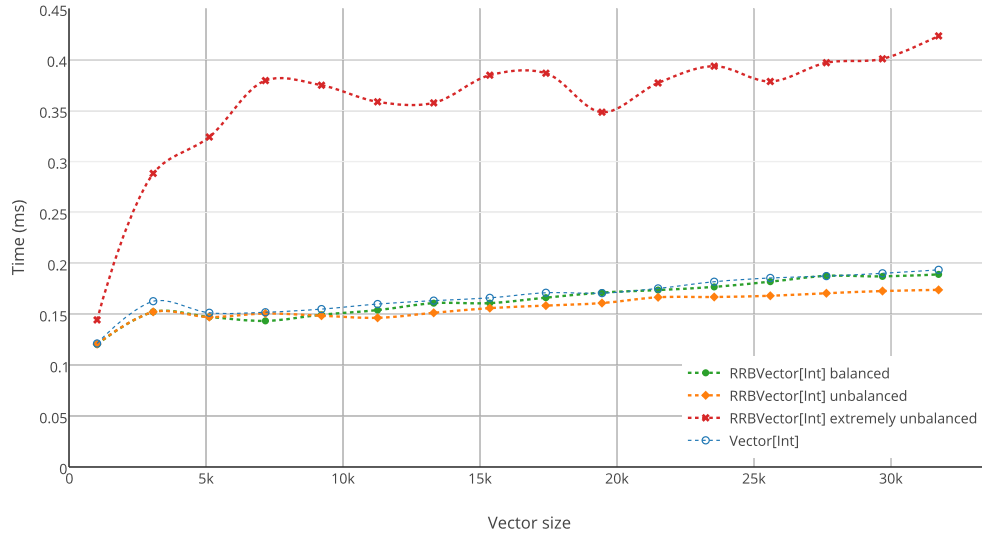


FIGURE 4.2: Time to execute 10k apply operations on random indices.

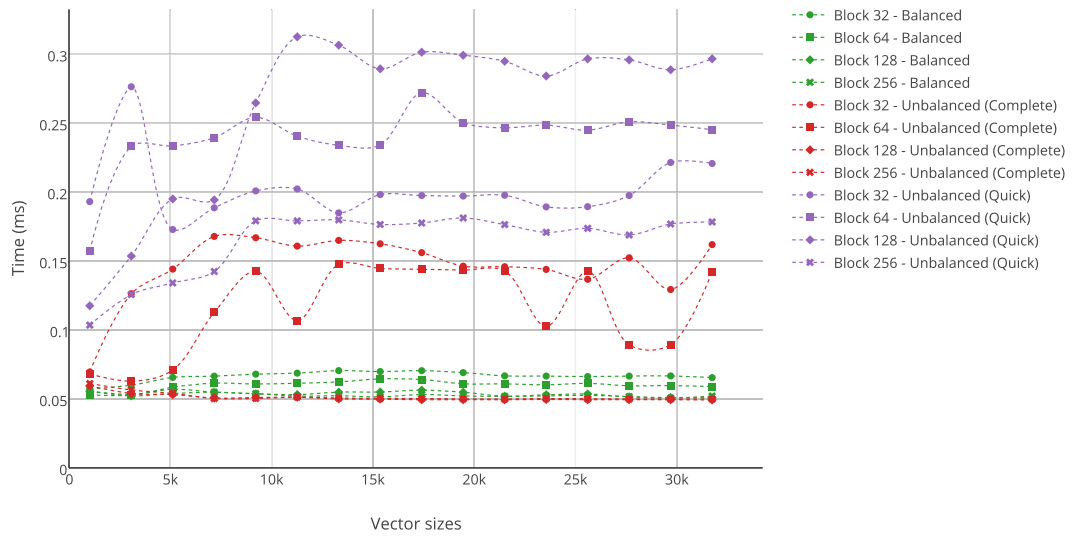


FIGURE 4.3: Time to execute 10k apply operations on sequential indices. Comparing performances for different block sizes and different implementation of the concatenation inner branch rebalancing (Complete/Quick).

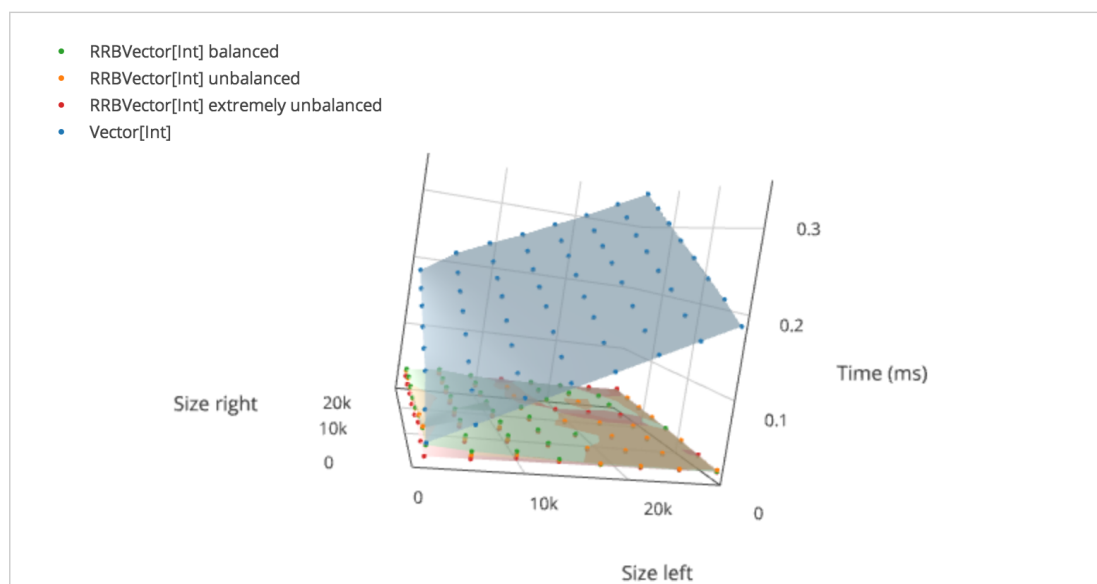


FIGURE 4.4: Execution time for a concatenation operation on two vectors. In theory (and in practice) Vector concatenation is  $O(\text{left} + \text{right})$  and the rrbVector concatenation operation is  $O(\log_{32}(\text{left} + \text{right}))$ .

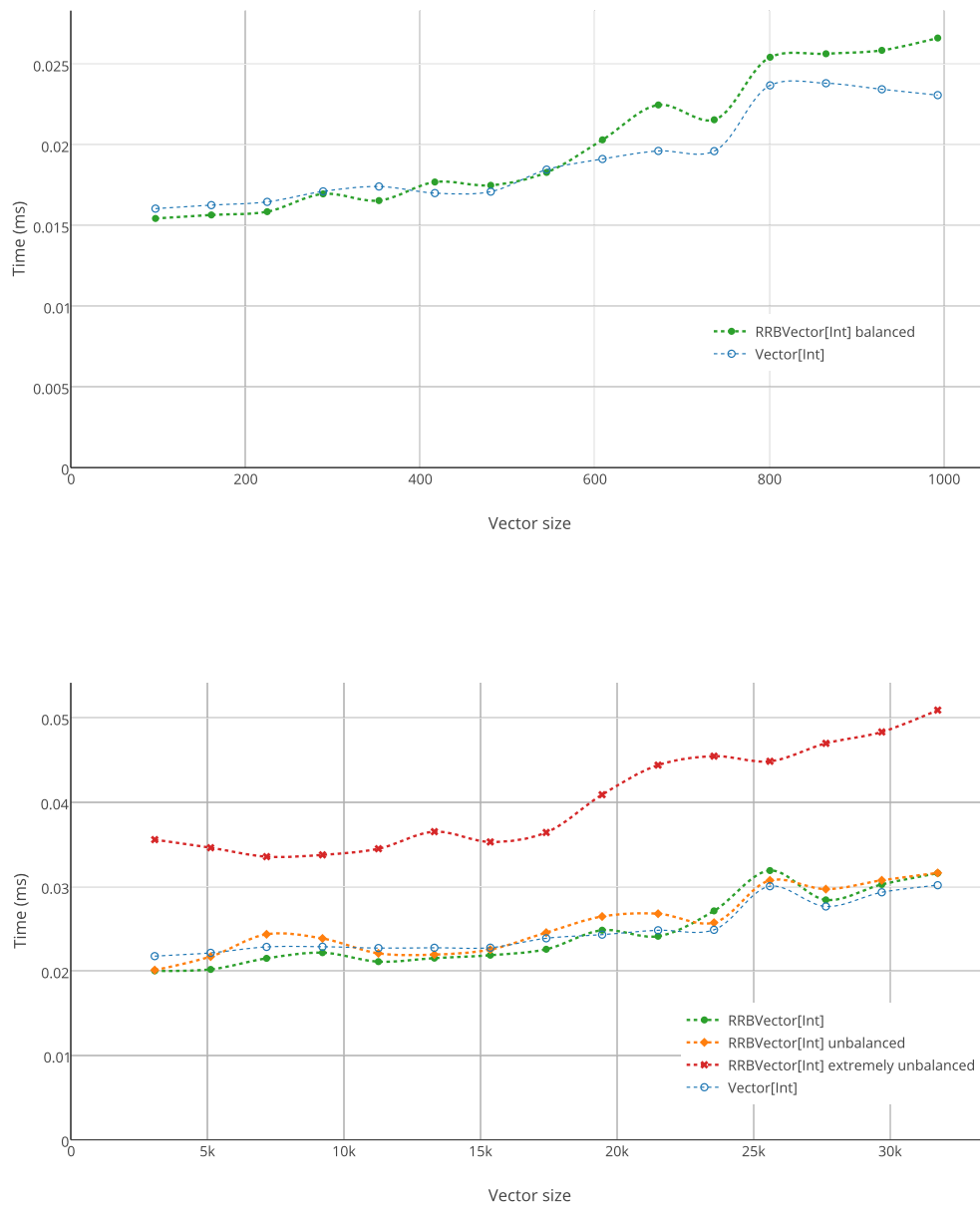


FIGURE 4.5: Time to execute 256 append operations. This shows the amortized cost of the append operation.

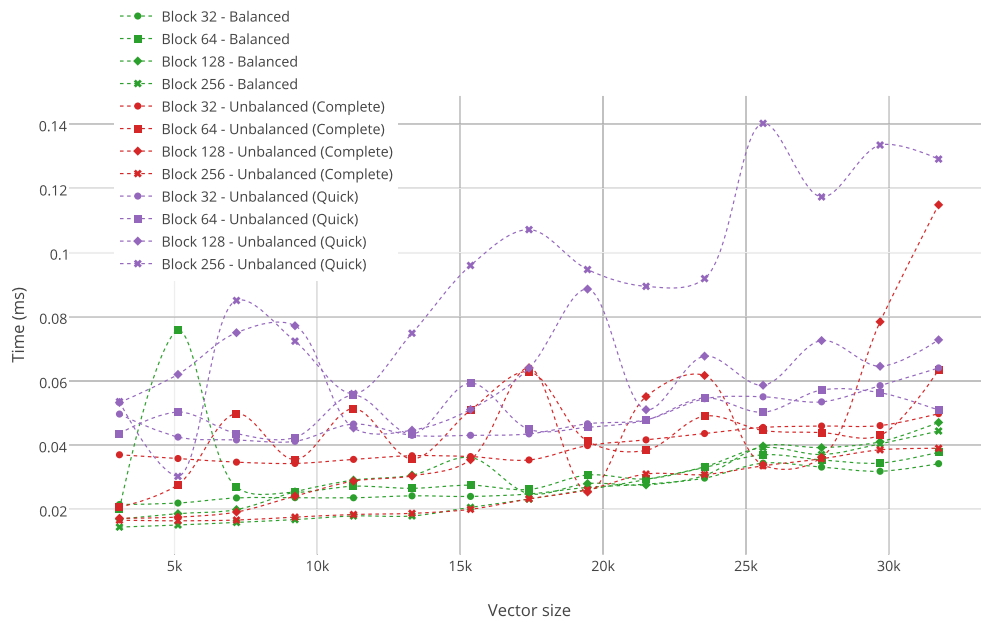


FIGURE 4.6: Time to execute 256 append operations. This shows the amortized cost of the append operation. Comparing performances for different block sizes and different implementation of the concatenation inner branch rebalancing (Complete/Quick).



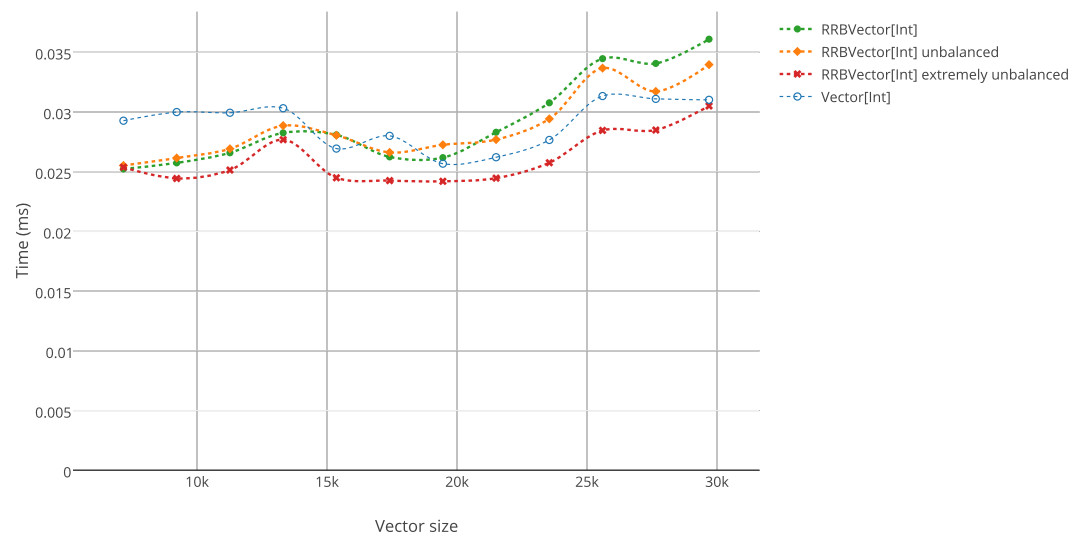
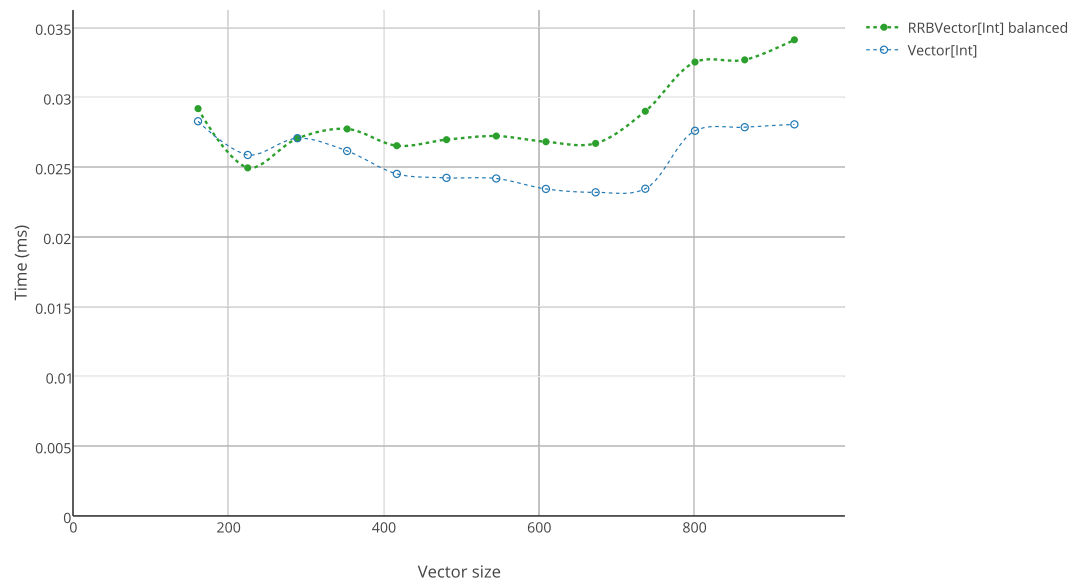


FIGURE 4.7: Time to execute 256 prepend operations. This shows the amortized cost of the prepend operation.

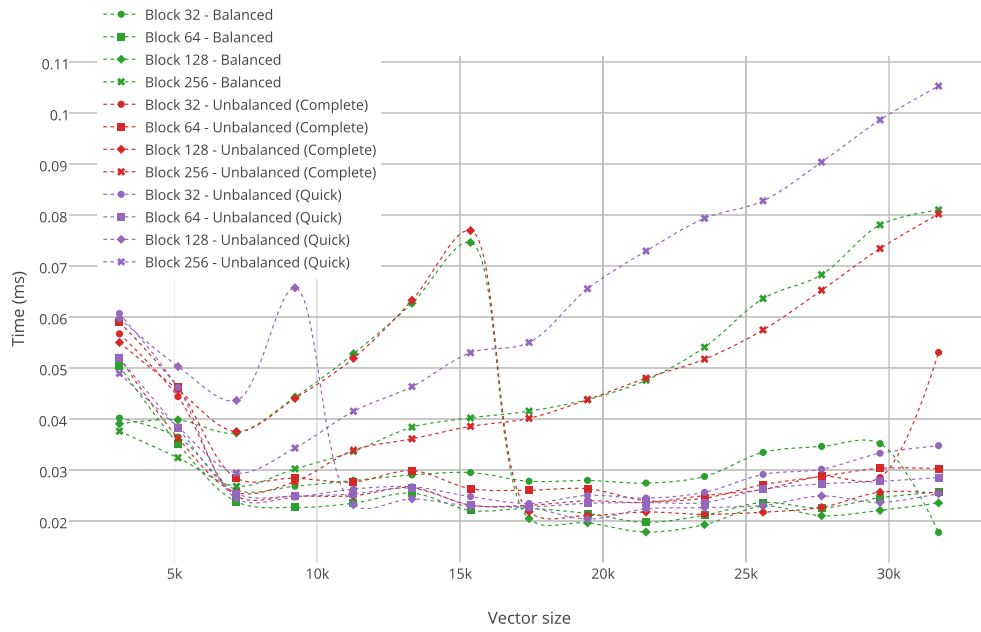


FIGURE 4.8: Time to execute 256 prepend operations. This shows the amortized cost of the append operation. Comparing performances for different block sizes and different implementation of the concatenation inner branch rebalancing (Complete/Quick).

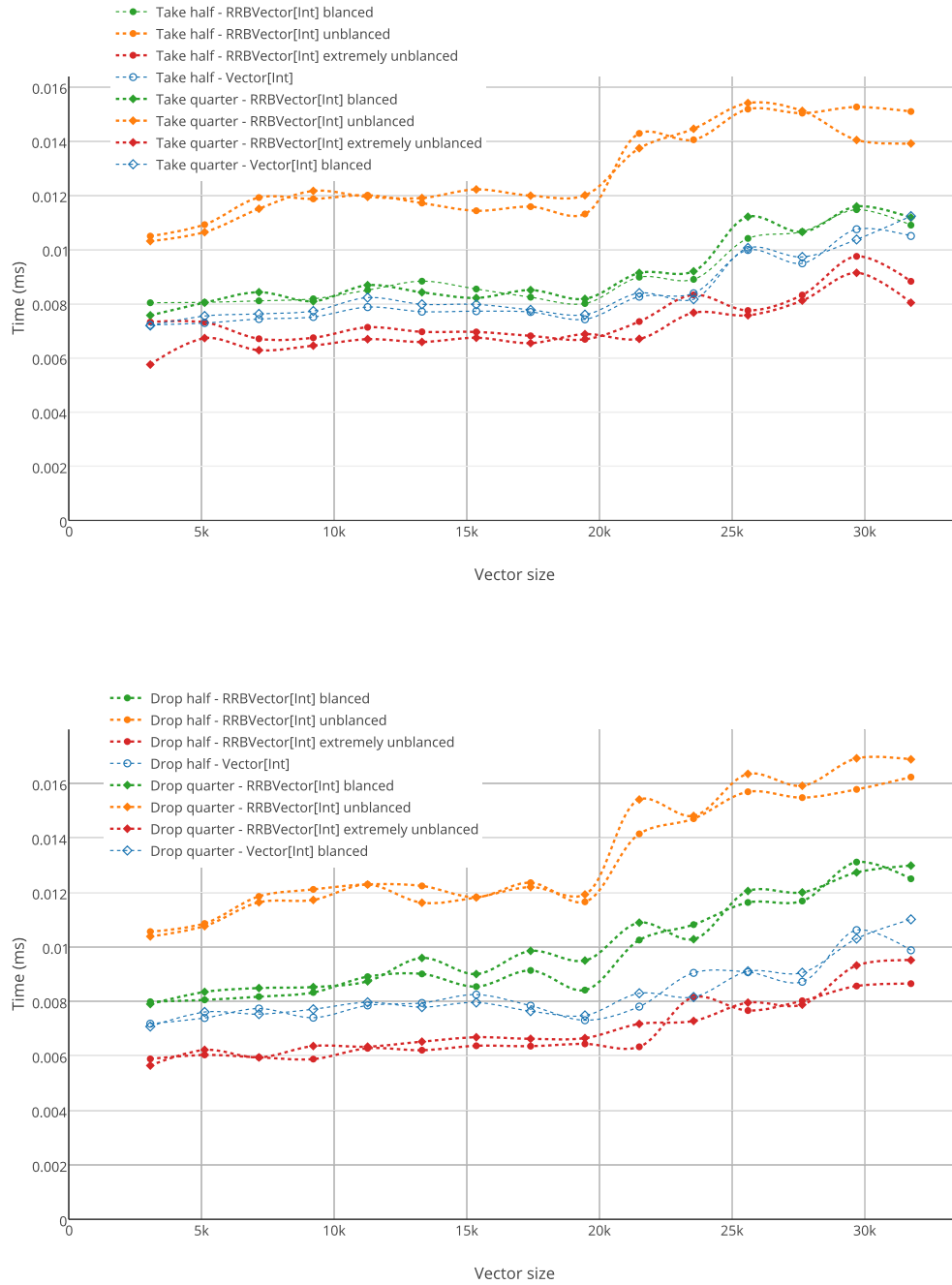


FIGURE 4.9: Execution time of take and drop.

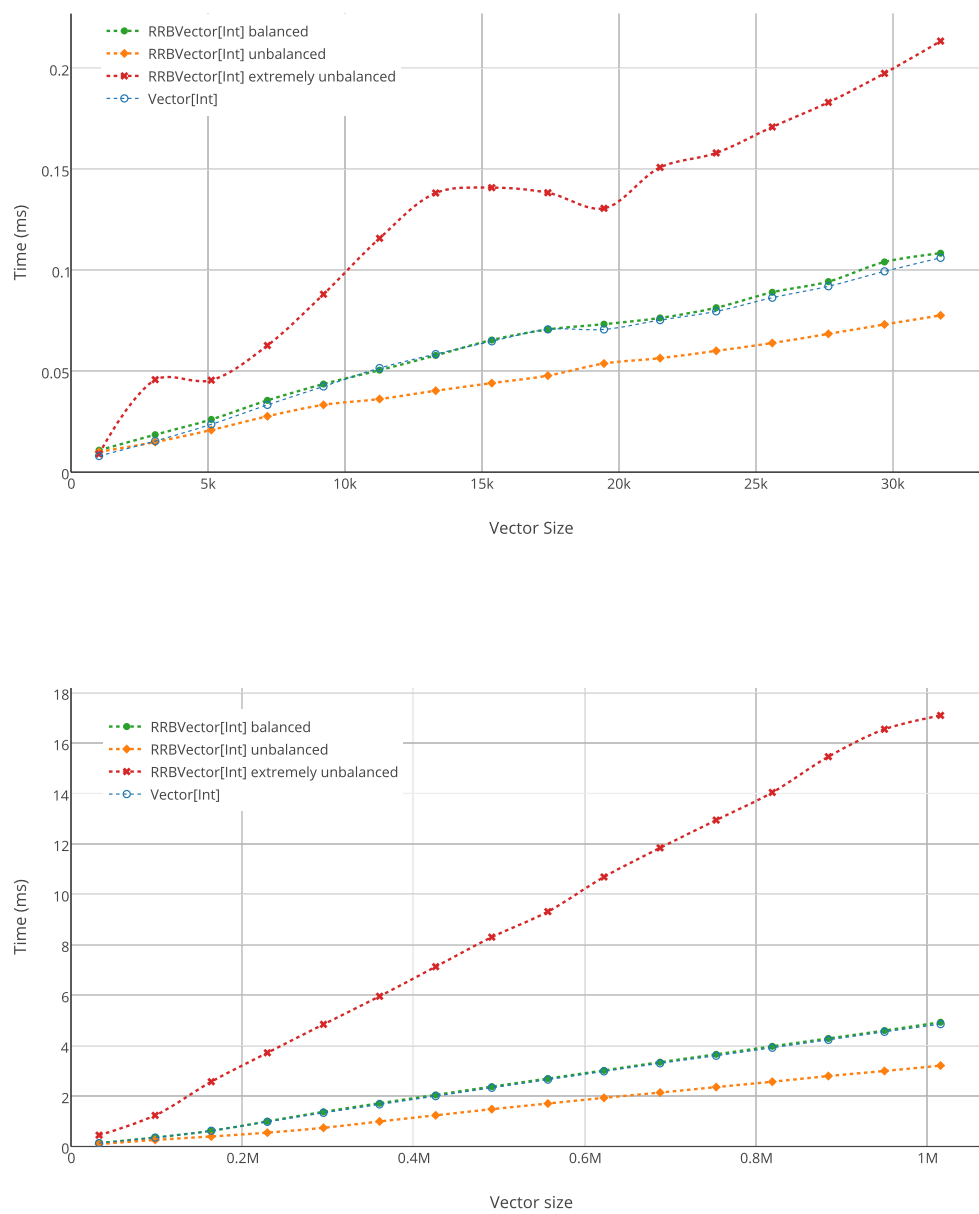


FIGURE 4.10: Execution time to iterate through all the elements of the vector.

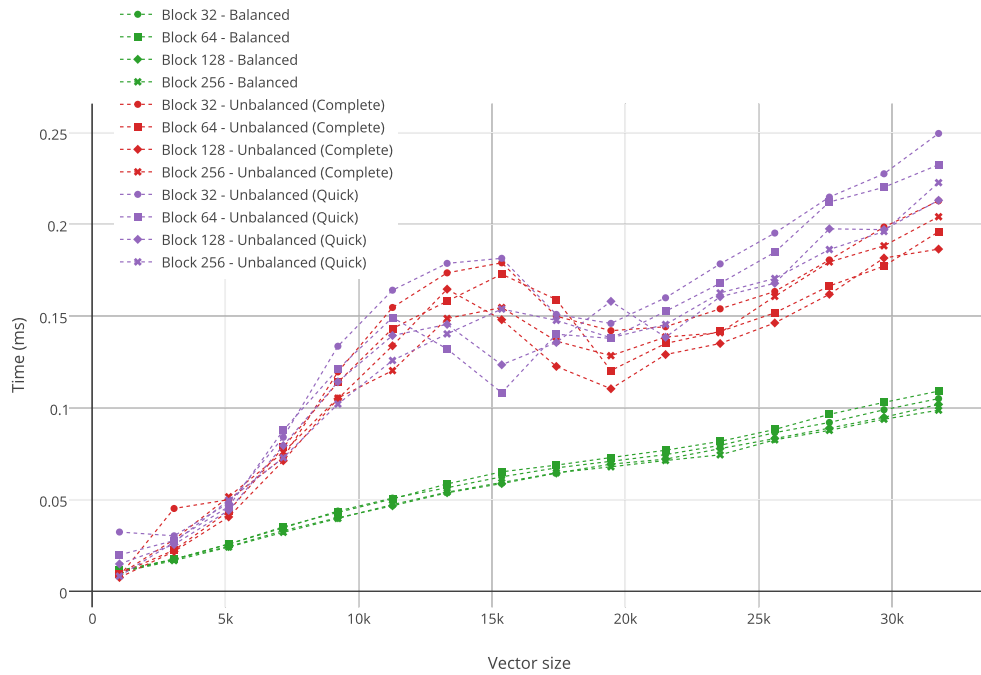


FIGURE 4.11: Execution time to iterate through all the elements of the vector. Comparing performances for different block sizes and different implementation of the concatenation inner branch rebalancing (Complete/Quick).

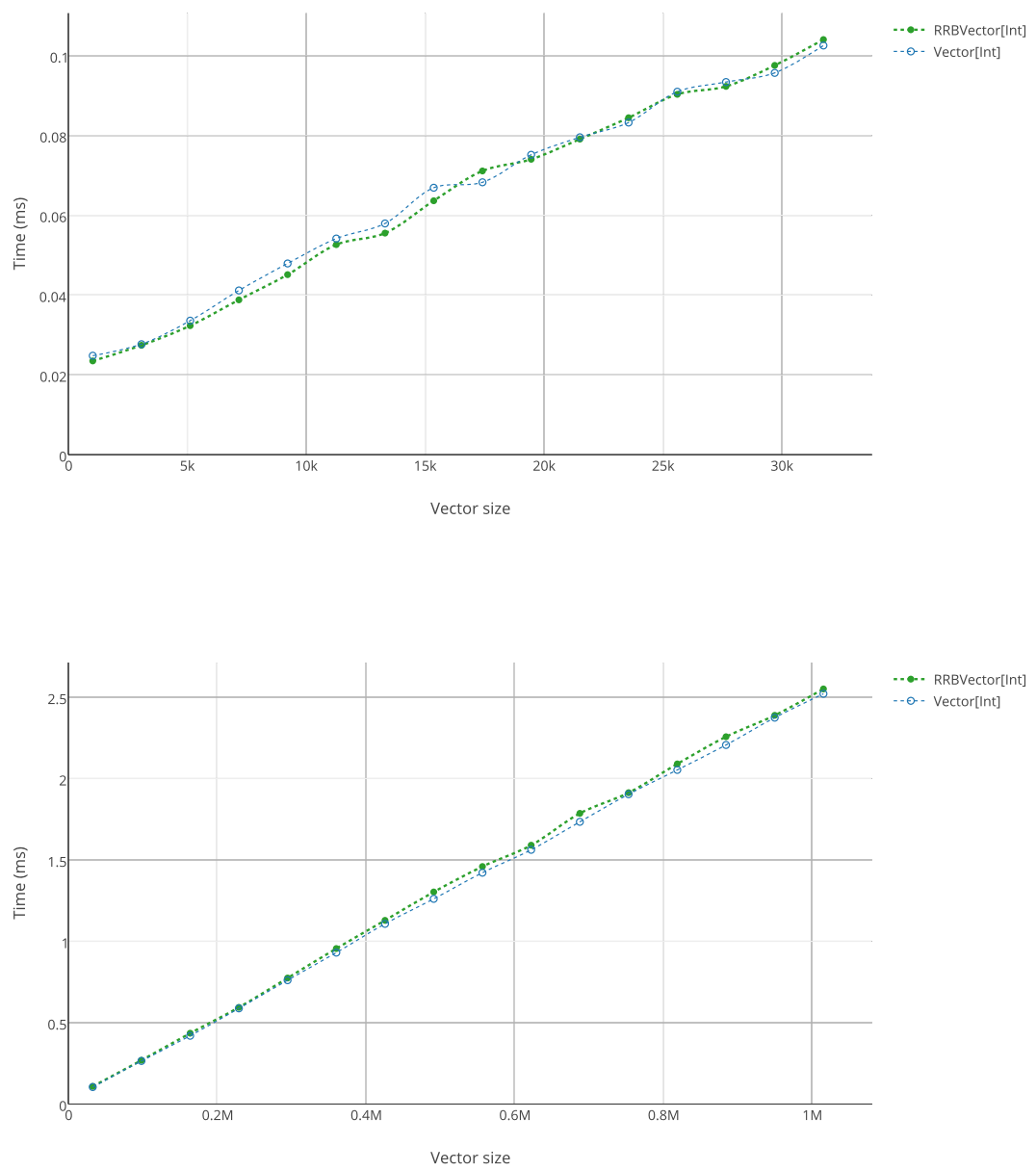


FIGURE 4.12: Execution time to build a vector of a given size.

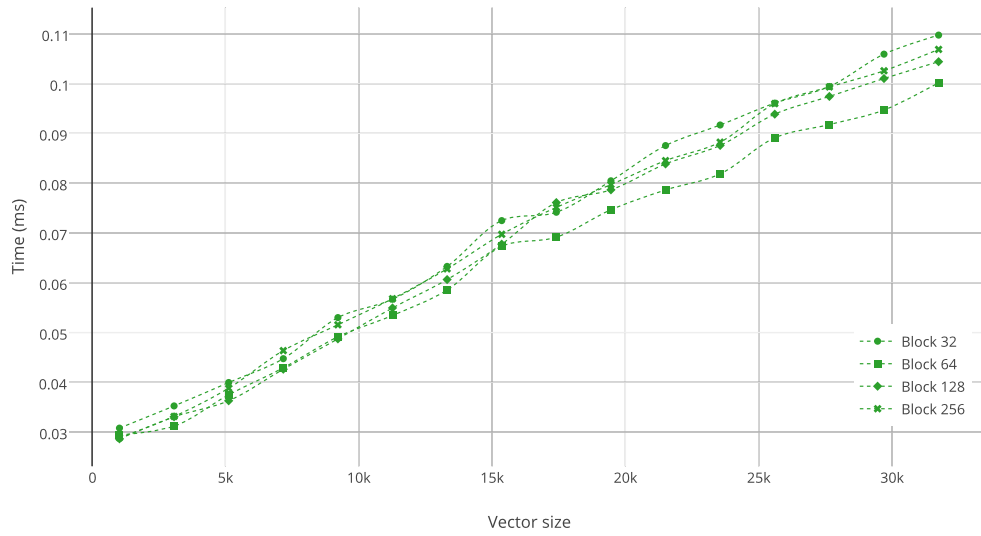


FIGURE 4.13: Execution time to build a vector of a given size. Comparing performances for different block sizes.

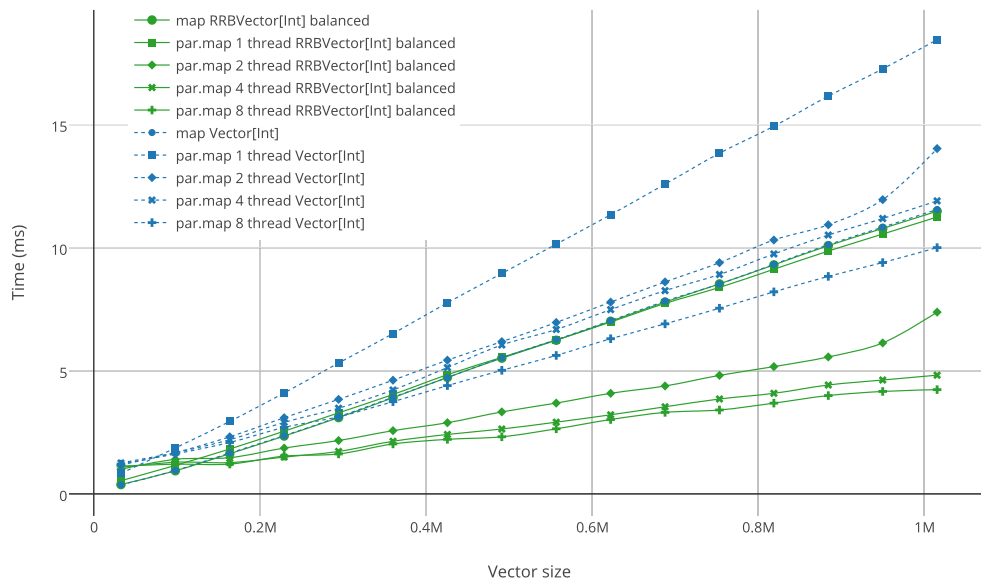


FIGURE 4.14: Benchmark on map and parallel map using the function  $(x \Rightarrow x)$  to show the difference time used in the framework. This time represents the time spent in the splitters and combiners of the parallel collection (iterator and builder for the sequential version).

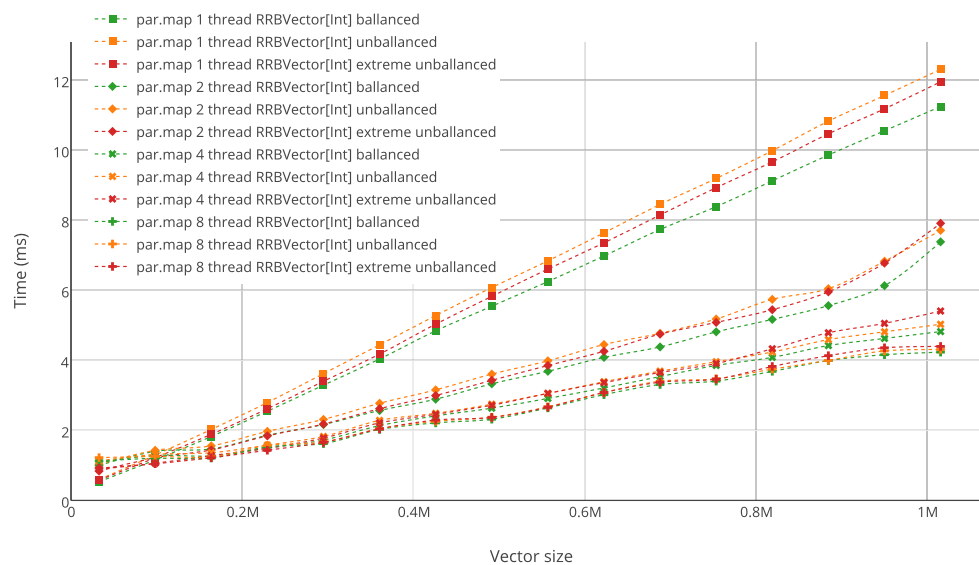


FIGURE 4.15: Benchmark on map and parallel map using the function  $(x \Rightarrow x)$  to show the difference time used in the framework. This time represents the time spent in the splitters and combiners of the parallel collection.



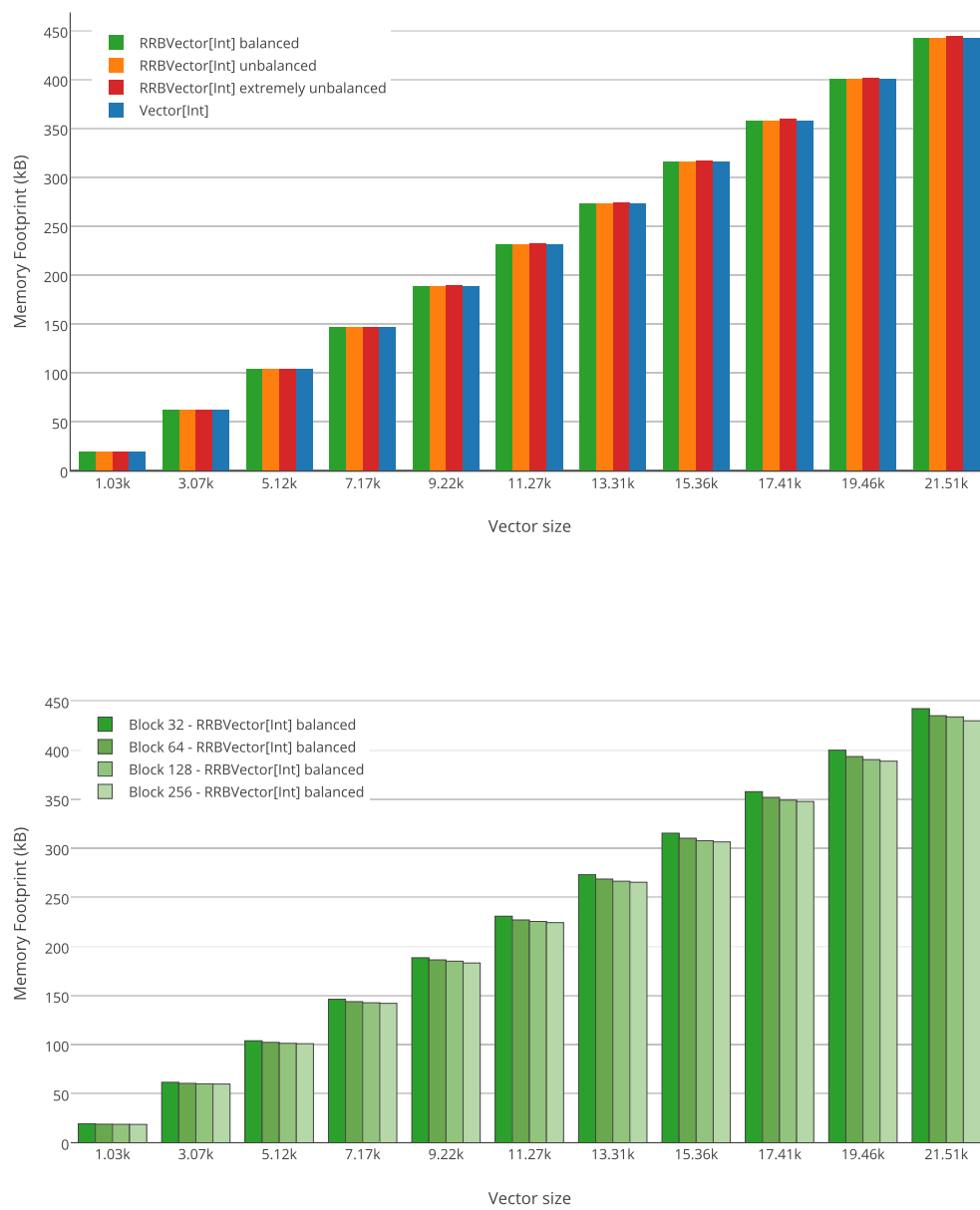


FIGURE 4.16: Memory Footprint

## Chapter 5

# Testing

### 5.1 Teststing correctness

#### 5.1.1 Invariant Assertions

#### 5.1.2 Unit tests

### 5.2 Main Section 2

I

## Chapter 6

# Related Work

### 6.1 RRB-Vectors in Clojure

I

## Chapter 7

## Conclusions