

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

Abstract	i
Contents	ii
List of Figures	iv
List of Tables	vi
Abbreviations	vii
1 Introduction	2
1.1 Main Section 1	2
1.2 Main Section 2	2
2 Vector Structure	4
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	5
2.1.2.3 Additions	5
Append	5
Prepend	5
Concatenation and Insert	5
2.1.2.4 Splits	6
2.2 Parallel Vectors	6
2.2.1 Splitter Iterator	6
2.2.2 Combiner Builder	6
2.3 Relaxed Radix Balanced Vectors	6
2.3.1 Relaxed Tree structure	6
2.3.2 Relaxed Operations	7
2.3.2.1 Apply (get element at index)	7
2.3.2.2 Updated	7
2.3.2.3 Additions	7
Append	7
Prepend	7
Concatenation	7

Insert	7
2.3.2.4 Splits	8
2.3.2.5 Parallel Vector	8
3 Implementation and Optimizations	9
3.1 Where is time spent?	9
3.1.1 Arrays	9
3.1.2 Computing indices	9
3.1.3 Abstractions	10
3.2 Displays	10
3.2.1 As cache	10
3.2.2 For transient states	10
3.3 Builder	11
3.4 Iterator	11
3.5 Relaxing the Radix	11
3.5.1 Relaxing Displays	11
3.5.2 Relaxing the Builder	11
3.5.3 Relaxing Iterator	11
4 Performance	12
4.1 In practice: Running on JVM	13
4.1.1 Cost of Abstraction and JIT Inline	13
4.2 Measuring performance	13
4.3 Generators	13
4.4 Benchmarks	13
4.4.1 Apply	13
4.4.2 Concatenation	13
4.4.3 Append	13
4.4.4 Prepend	13
4.4.5 Splits	13
4.4.6 Iterator	13
4.4.7 Builder	13
4.4.8 Parallel split-combine	13
4.4.9 Memory footprint	13
5 Testing	27
5.1 Teststing correctness	27
5.1.1 Unit tests	27
5.1.2 Invariant Assertions	27
6 Related Work	28
6.1 RRB-Vectors in Clojure	28
7 Conclusions	29

List of Figures

2.1	Radix Balanced Tree Structure	4
2.2	Radix Balanced Tree	6
2.3	Relaxed radix example	6
2.4	Concatenation example with blocks of size 4: Rebalancing level 0	7
2.5	Concatenation example with blocks of size 4: Rebalancing level 1	7
2.6	Concatenation example with blocks of size 4: Rebalancing level 2	8
2.7	Concatenation example with blocks of size 4: Rebalancing level 3	8
3.1	Accessing element at index 526843 in a tree of depth 5. Empty nodes represent collapses subtrees.	9
3.2	Displays	10
3.3	Radix Balanced Tree Transient state	10
3.4	Radix Balanced Tree	11
4.1	Time to execute 10k apply operations on sequential indices.	13
4.2	Time to execute 10k apply operations on random indices.	14
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).	14
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 $O(\log_{32}(left + right))$	15
4.5	Time to execute 256 append operations. This shows the amortized cost of the append operation.	16
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).	17
4.7	Time to execute 256 prepend operations. This shows the amortized cost of the prepend operation.	18
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).	19
4.9	Execution time of take and drop.	20
4.10	Excecuton time to iterate through all the elements of the vector.	21
4.11	Excecuton 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).	22

4.12	Execution time to build a vector of a given size.	23
4.13	Execution time to build a vector of a given size. Comparing performances for different block sizes.	24
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).	24
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.	25
4.16	Memory Footprint	26

List of Tables

Abbreviations

JIT	J ust I n T ime
RB	R adix B alanced
RRB	R elaxed R adix B alanced

I

Chapter 1

Introduction

1.1 Main Section 1

1.2 Main Section 2

I

Chapter 2

Vector Structure

2.1 Radix Balanced Vectors

2.1.1 Tree structure

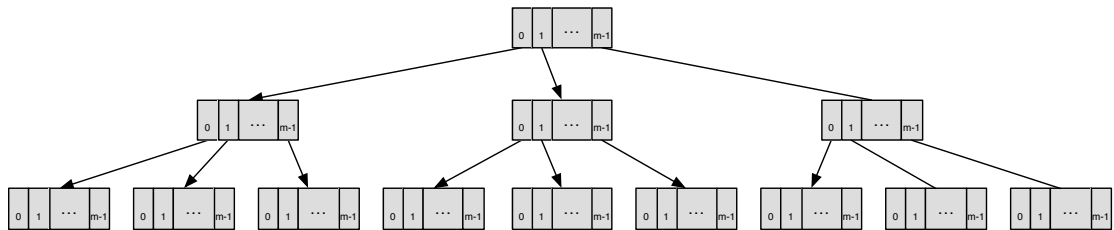


FIGURE 2.1: Radix Balanced Tree Structure

2.1.2 Operations

2.1.2.1 Apply

```
def apply(index: Int): A = {  
  def getElem(node: Array[AnyRef], depth: Int): A = {  
    val indexInNode = // compute index  
    if(depth == 1) node(indexInNode)  
    else getElem(node(indexInNode), depth-1)  
  }  
  getElem(vectorRoot, vectorDepth)  
}
```

2.1.2.2 Updated

```
def updated(index: Int, elem: A) = {  
  def updatedNode(node: Array[AnyRef], depth: Int) = {  
    val indexInNode = // compute index  
    val copy = clone(node)  
    if(depth == 1) {  
      copy(indexInNode) = elem  
    } else {  
      copy(indexInNode) =  
        updatedNode(node(indexInNode), depth-1)  
    }  
    copy  
  }  
  new Vector(updatedNode(vectorRoot, vectorDepth), ...)  
}
```

2.1.2.3 Additions

Append

Prepend

Concatenation and Insert

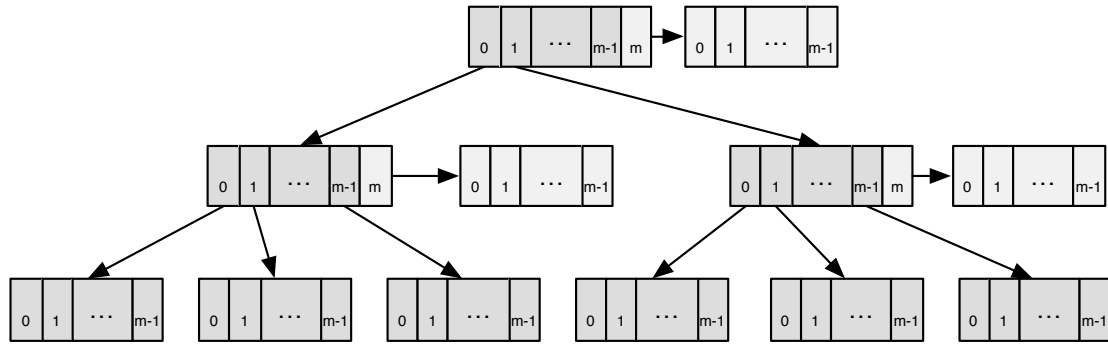


FIGURE 2.2: Radix Balanced Tree

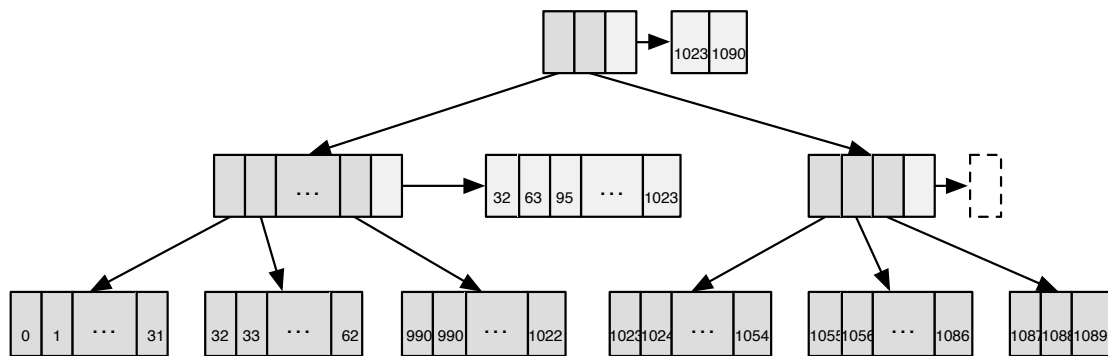


FIGURE 2.3: Relaxed radix example

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 Relaxed Tree structure

2.3.2 Relaxed Operations

2.3.2.1 Apply (get element at index)

2.3.2.2 Updated

2.3.2.3 Additions

Append

Prepend

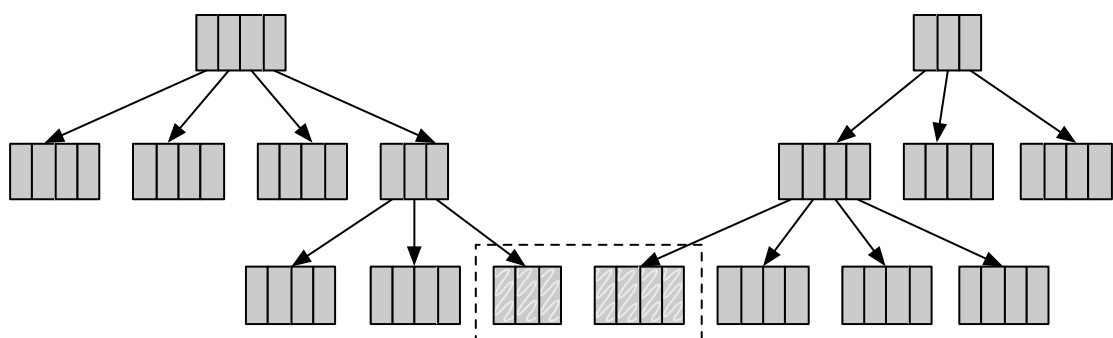


FIGURE 2.4: Concatenation example with blocks of size 4: Rebalancing level 0

Concatenation

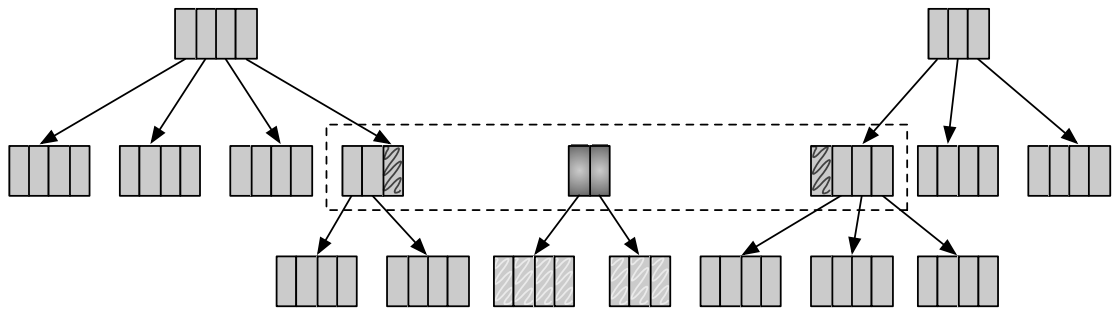


FIGURE 2.5: Concatenation example with blocks of size 4: Rebalancing level 1

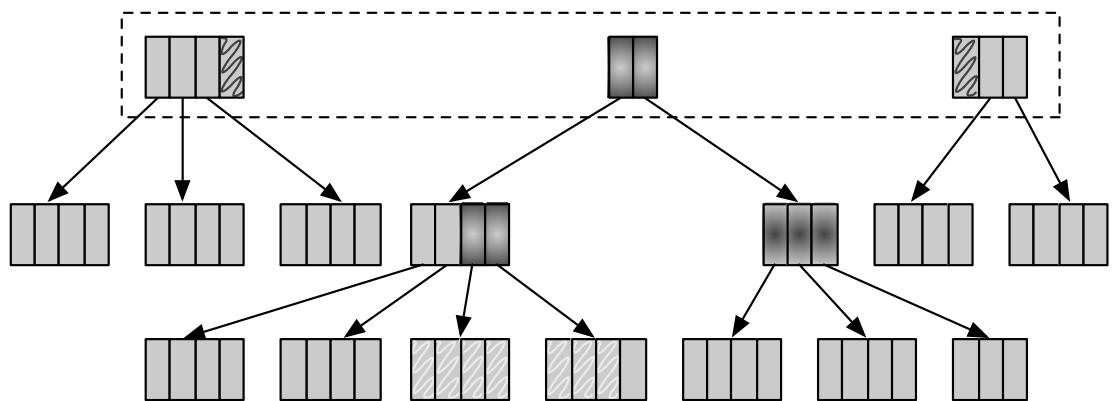


FIGURE 2.6: Concatenation example with blocks of size 4: Rebalancing level 2

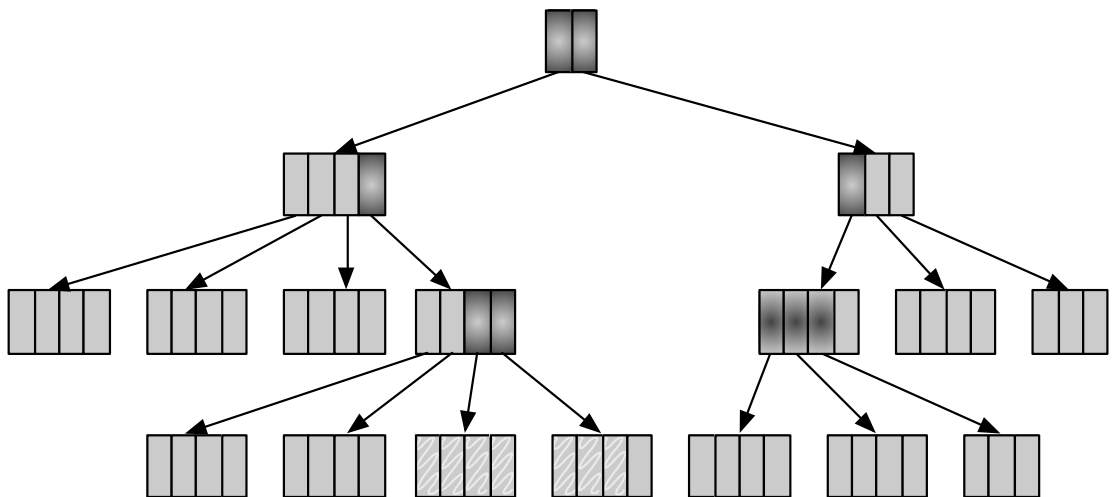


FIGURE 2.7: Concatenation example with blocks of size 4: Rebalancing level 3

Insert

2.3.2.4 Splits**2.3.2.5 Parallel Vector**

I

Chapter 3

Implementation and Optimizations

3.1 Where is time spent?

3.1.1 Arrays

3.1.2 Computing indices

$$526843 = \underbrace{00}_{0} \underbrace{000000}_{0} \underbrace{10000}_{16} \underbrace{00010}_{2} \underbrace{01111}_{15} \underbrace{11011}_{27}$$

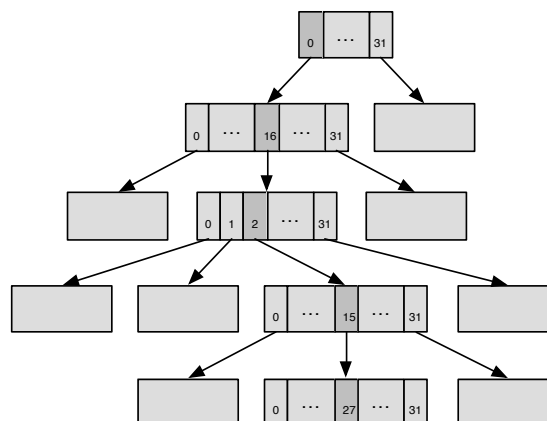


FIGURE 3.1: Accessing element at index 526843 in a tree of depth 5. Empty nodes represent collapses subtrees.

3.1.3 Abstractions

3.2 Displays

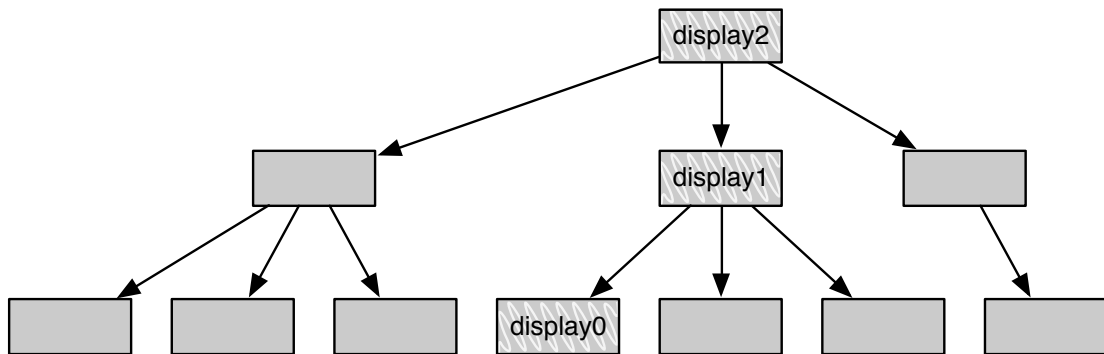


FIGURE 3.2: Displays

3.2.1 As cache

3.2.2 For transient states

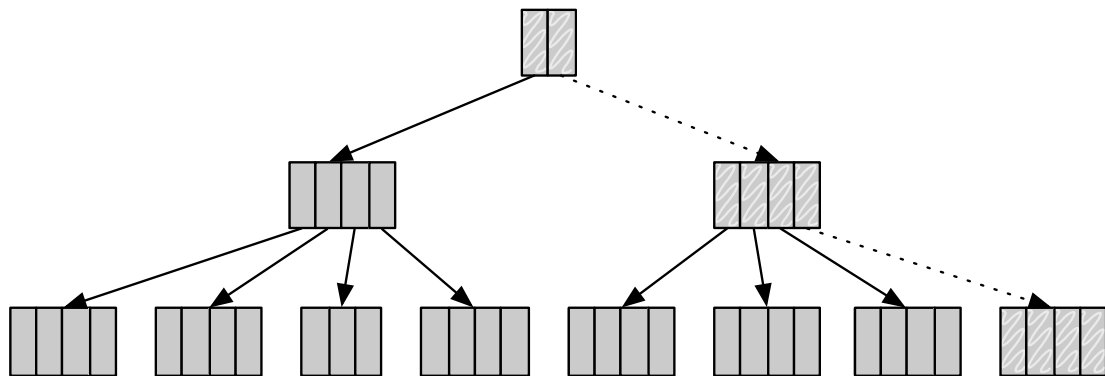


FIGURE 3.3: Radix Balanced Tree Transient state

3.3 Builder

3.4 Iterator

3.5 Relaxing the Radix

3.5.1 Relaxing Displays

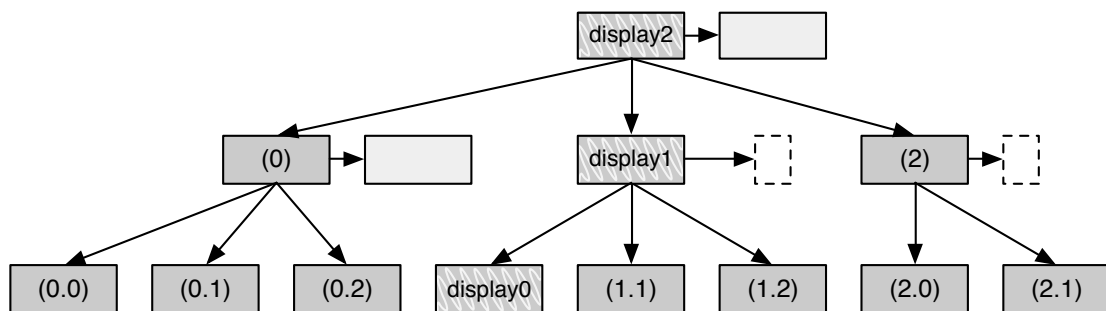


FIGURE 3.4: Radix Balanced Tree

3.5.2 Relaxing the Builder

3.5.3 Relaxing Iterator

I

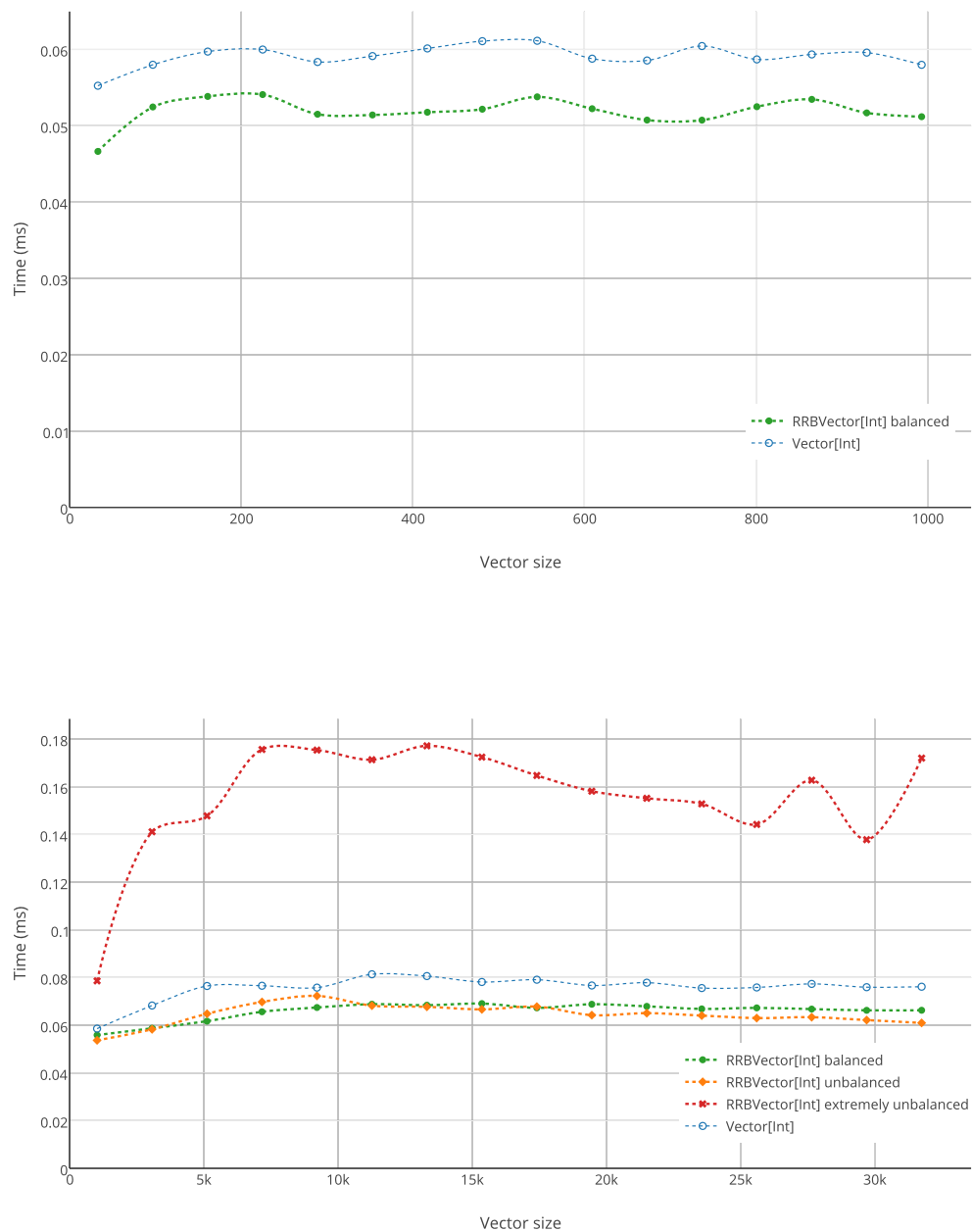


FIGURE 4.1: Time to execute 10k apply operations on sequential indices.

Chapter 4

Performance

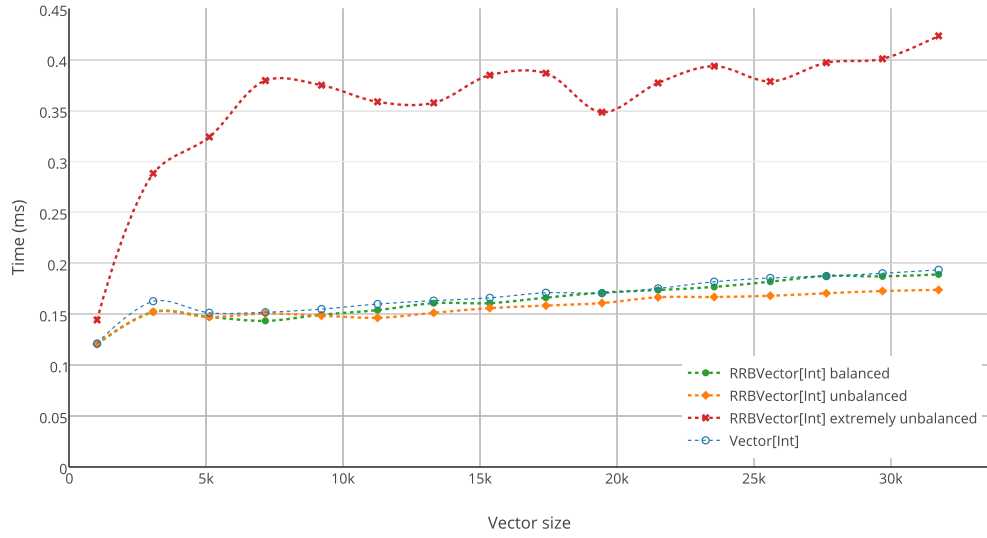


FIGURE 4.2: Time to execute 10k apply operations on random indices.

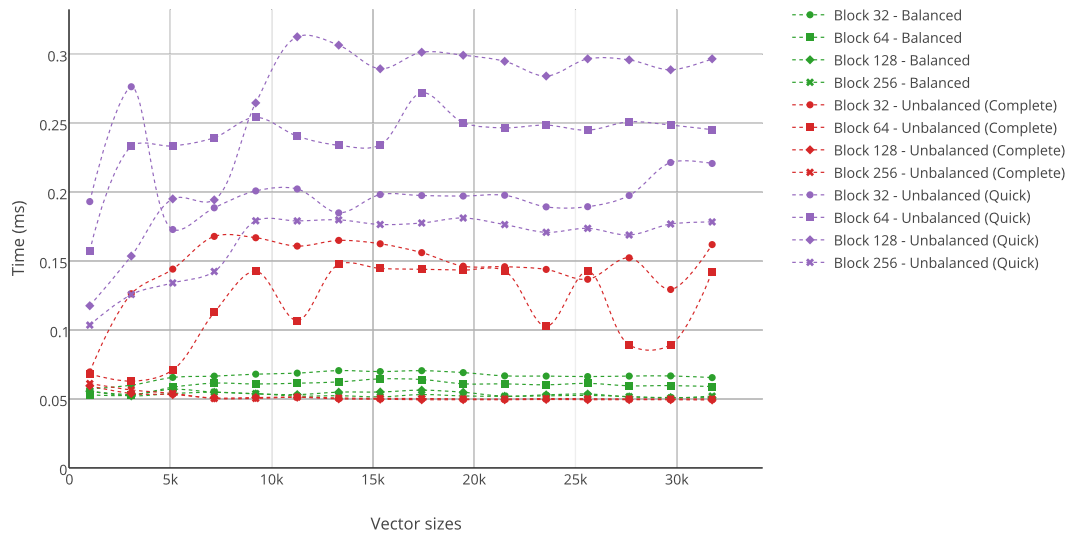


FIGURE 4.3: Time to execute 10k apply operations on sequential indices. Comparing performances for different block sizes and different implementation of the concatenation inner branch rebalancing (Complete/Quick).

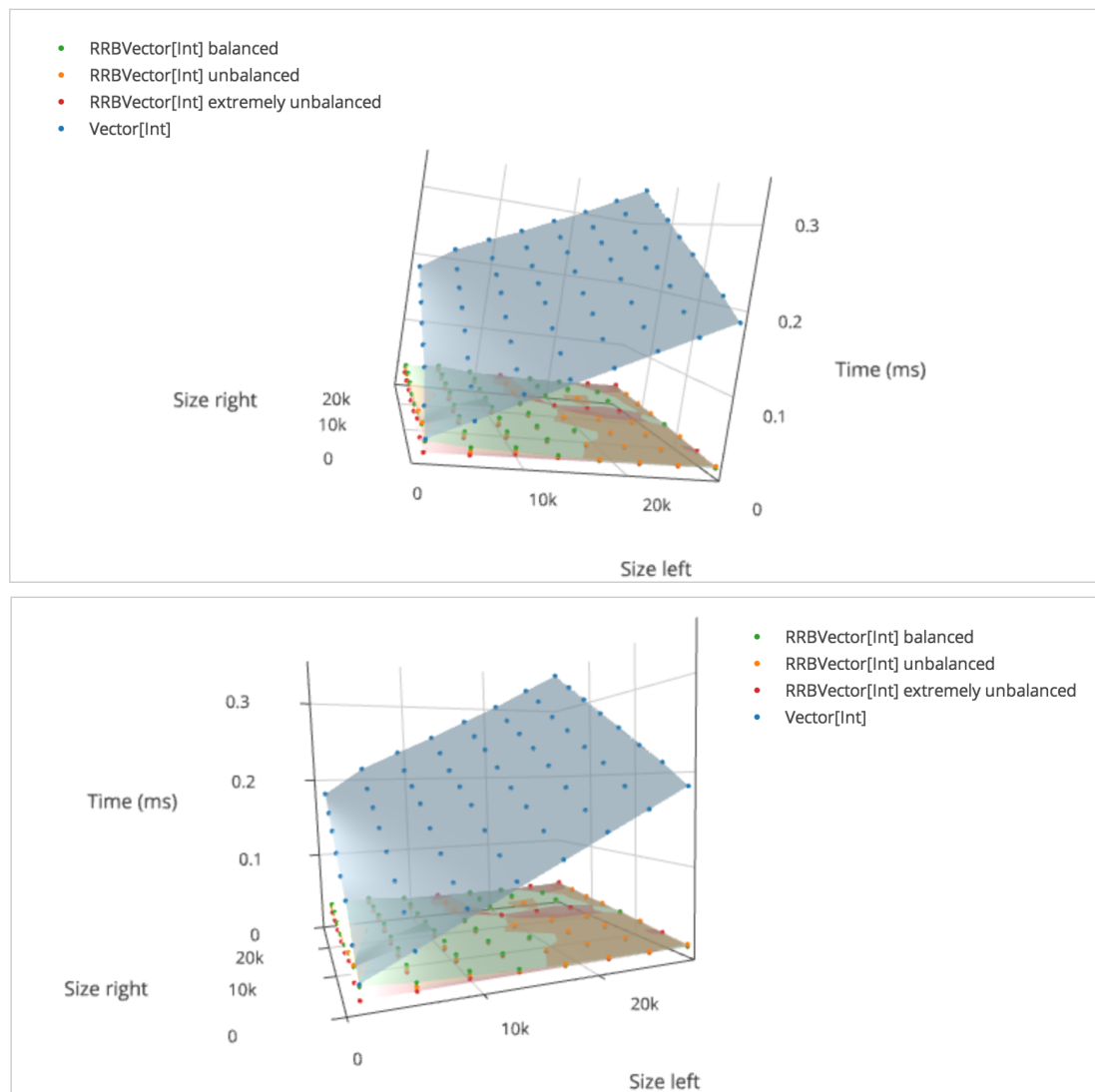


FIGURE 4.4: Execution time for a concatenation operation on two vectors. In theory (and in practice) Vector concatenation is $O(left + right)$ and the rrbVector concatenation operation is $O(\log_{32}(left + right))$.

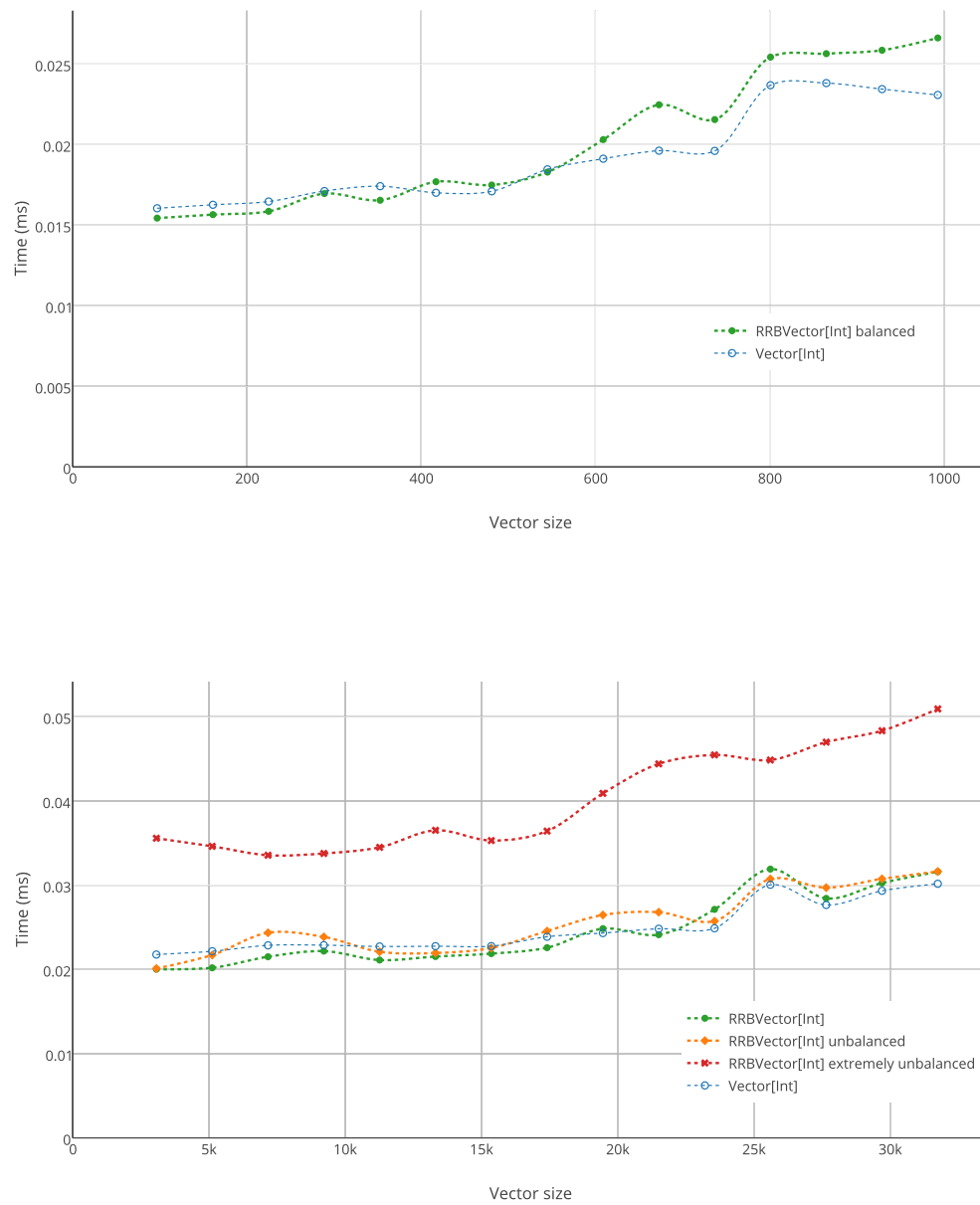


FIGURE 4.5: Time to execute 256 append operations. This shows the amortized cost of the append operation.

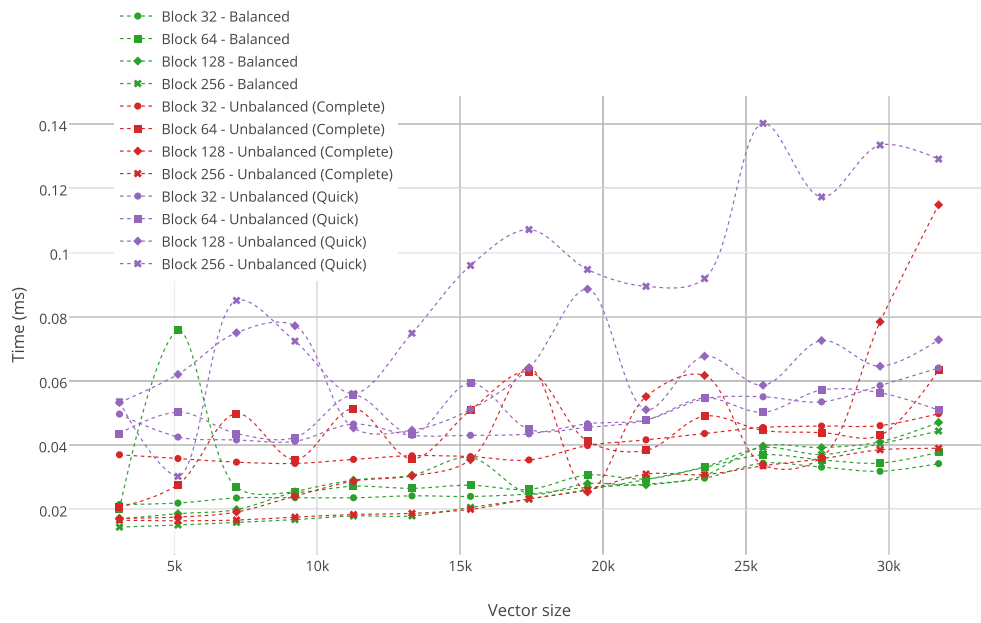


FIGURE 4.6: Time to execute 256 append operations. This shows the amortized cost of the append operation. Comparing performances for different block sizes and different implementation of the concatenation inner branch rebalancing (Complete/Quick).

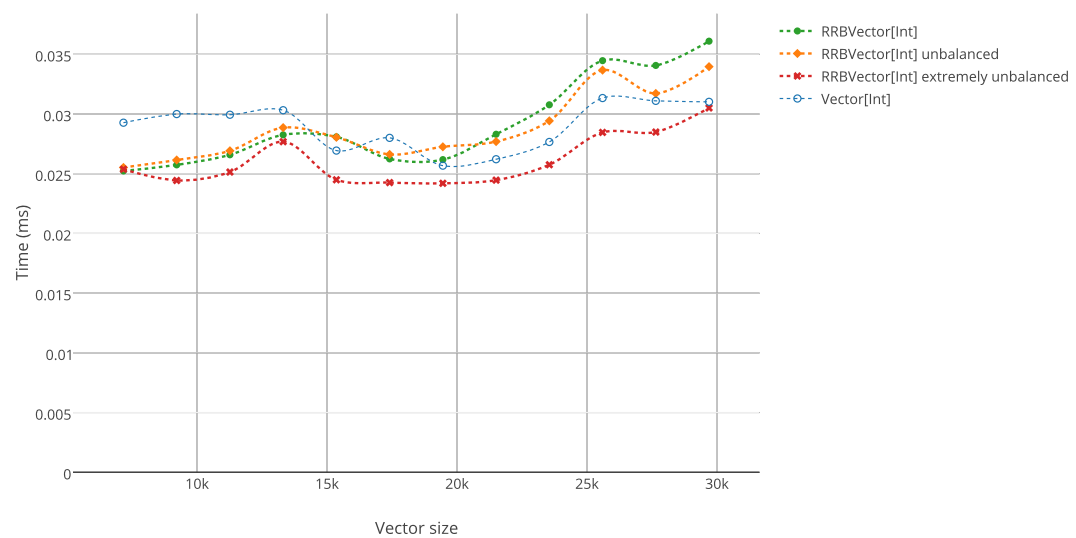
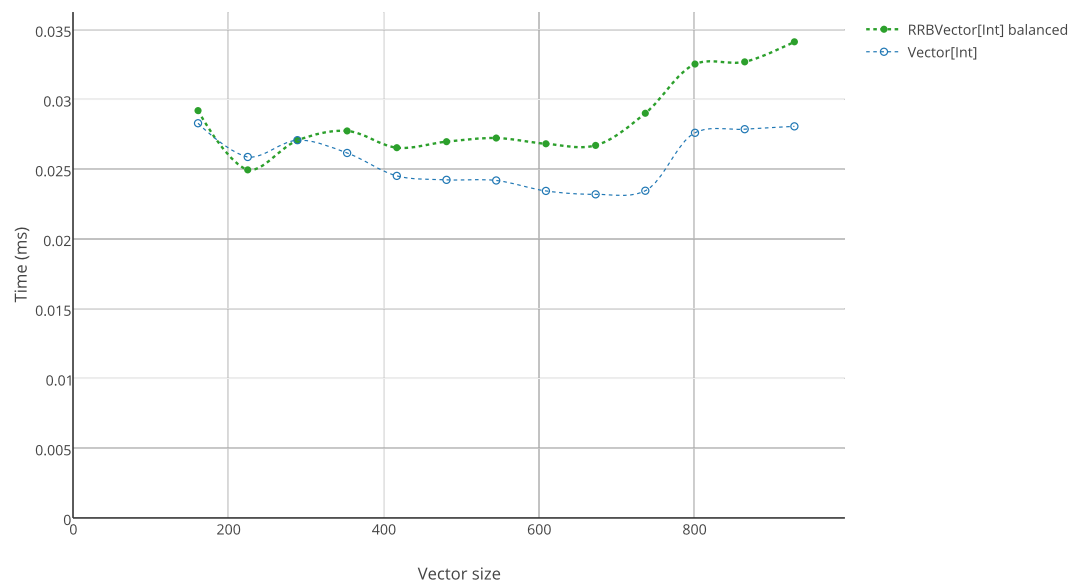


FIGURE 4.7: Time to execute 256 prepend operations. This shows the amortized cost of the prepend operation.

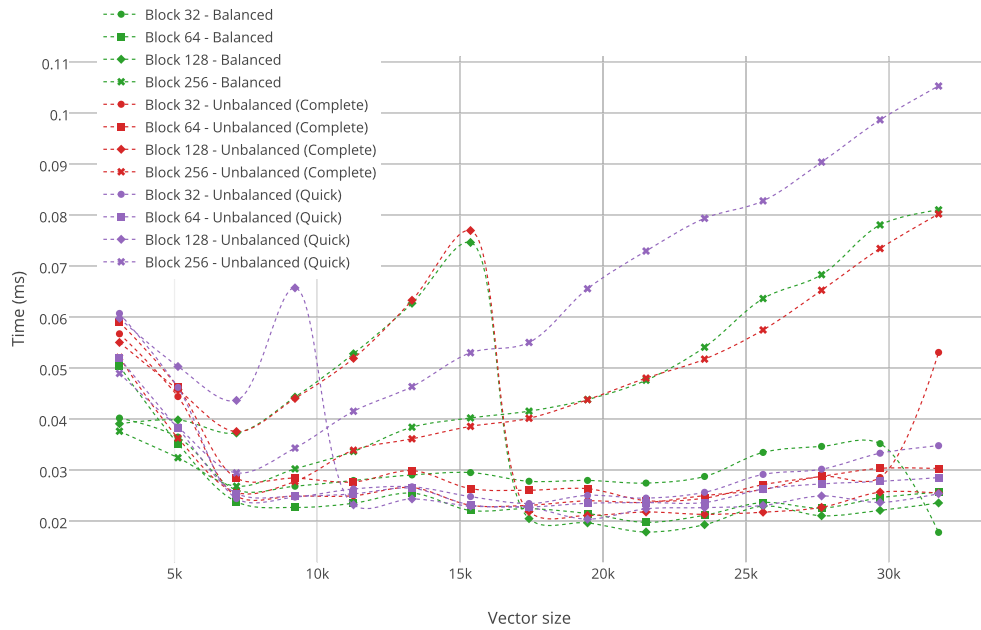


FIGURE 4.8: Time to execute 256 prepend operations. This shows the amortized cost of the append operation. Comparing performances for different block sizes and different implementation of the concatenation inner branch rebalancing (Complete/Quick).

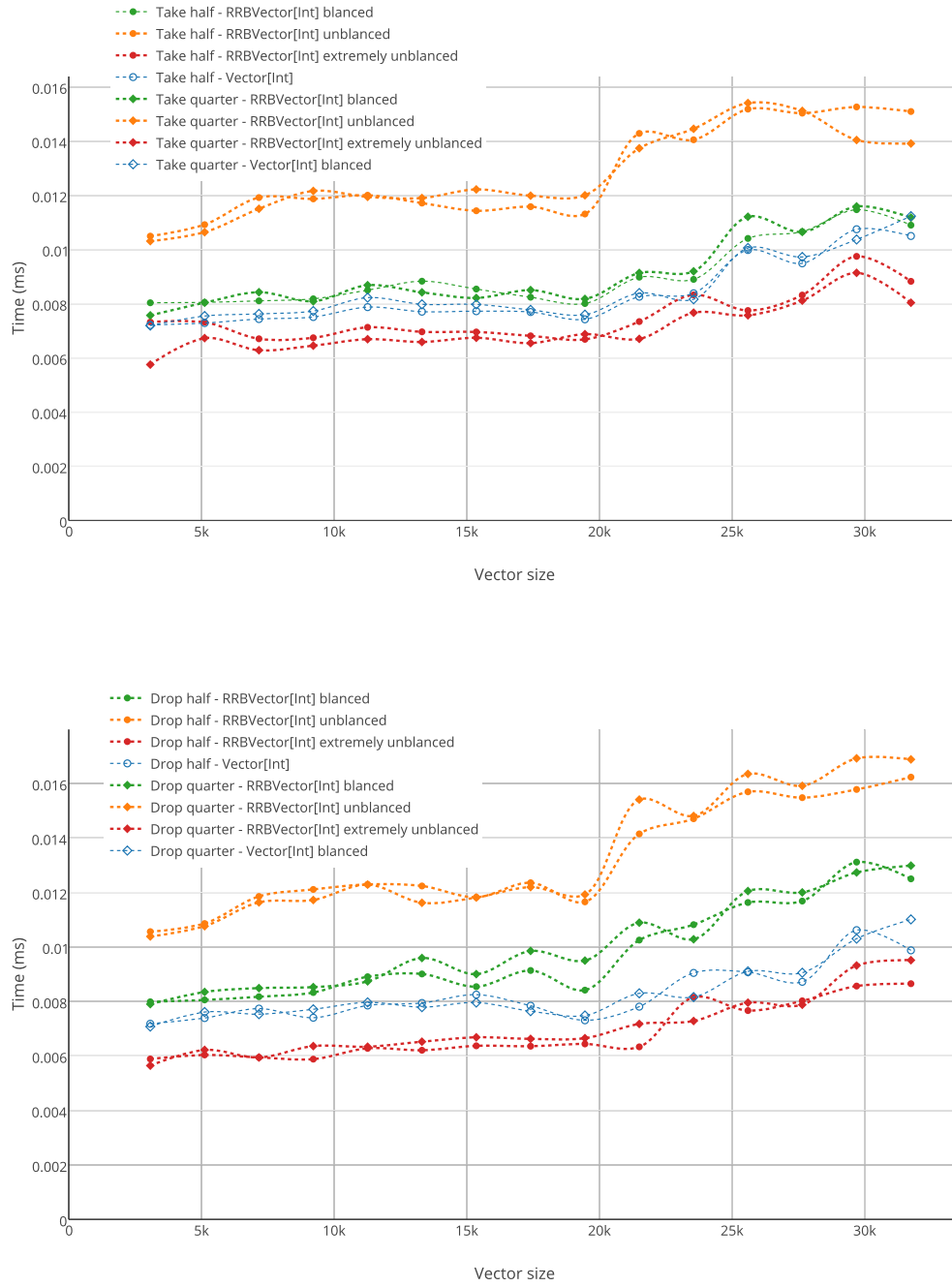


FIGURE 4.9: Execution time of take and drop.

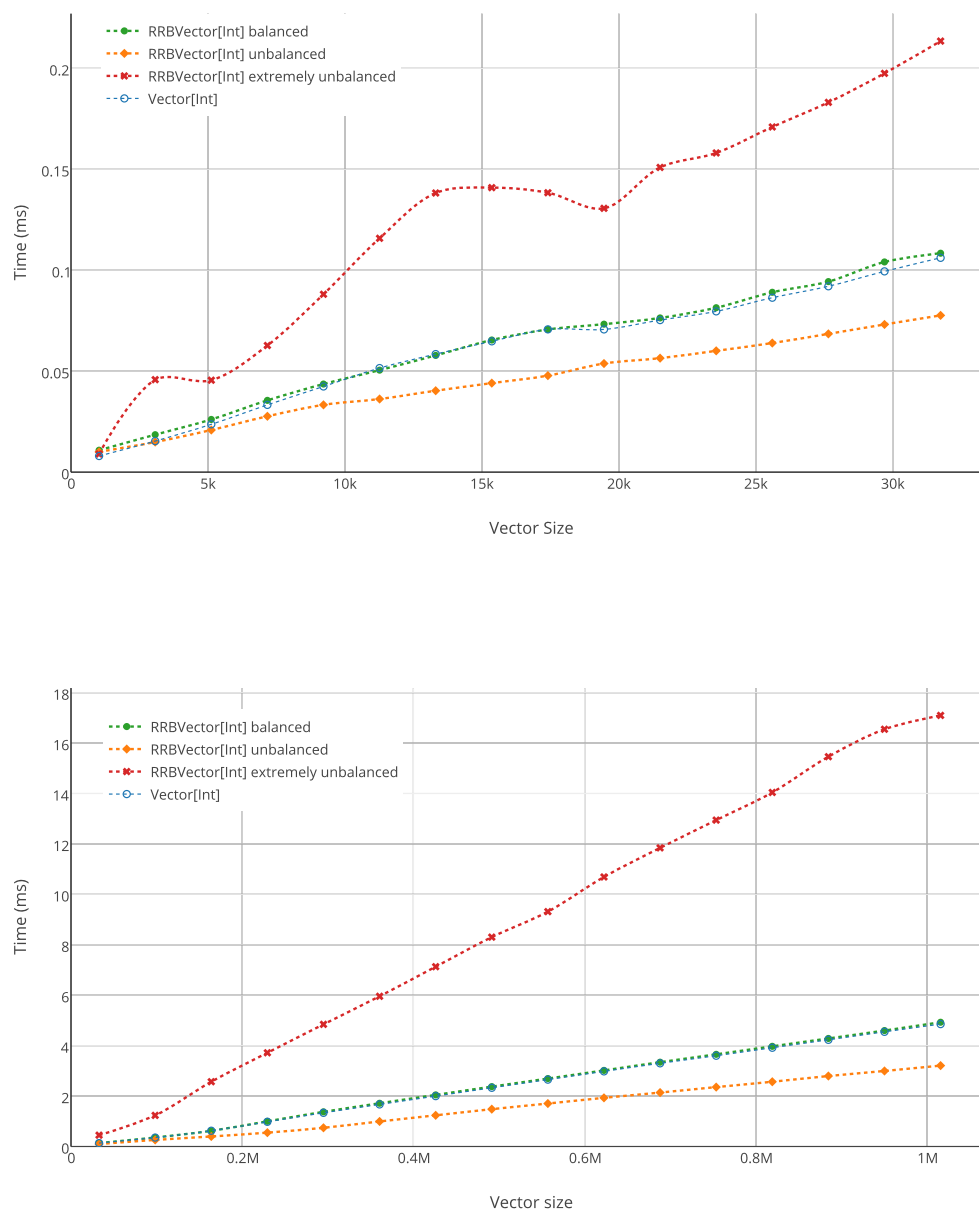


FIGURE 4.10: Execution time to iterate through all the elements of the vector.

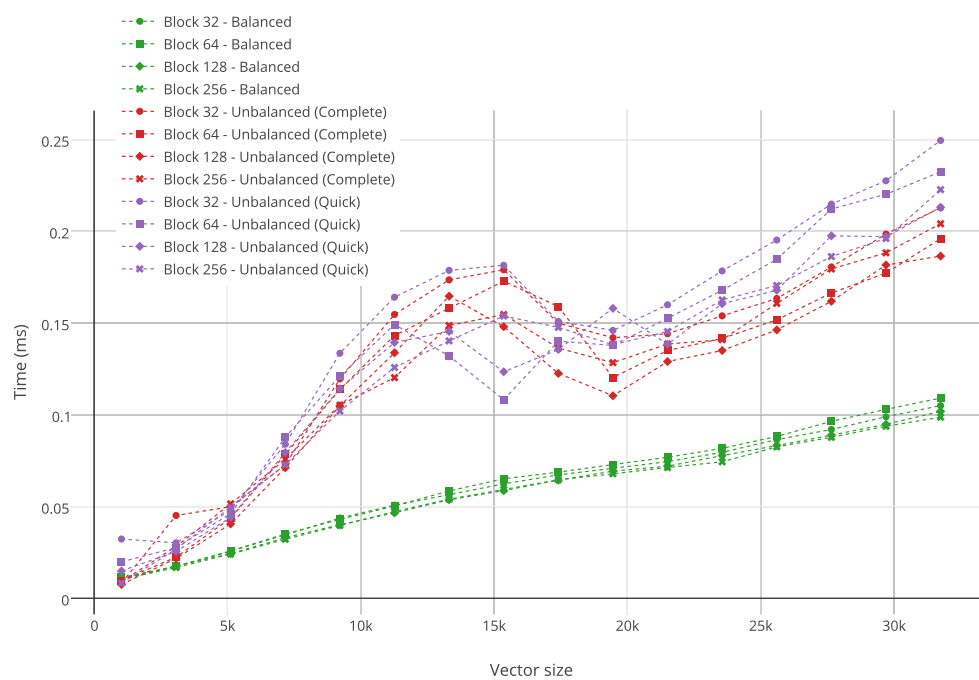


FIGURE 4.11: Execution time to iterate through all the elements of the vector. Comparing performances for different block sizes and different implementation of the concatenation inner branch rebalancing (Complete/Quick).

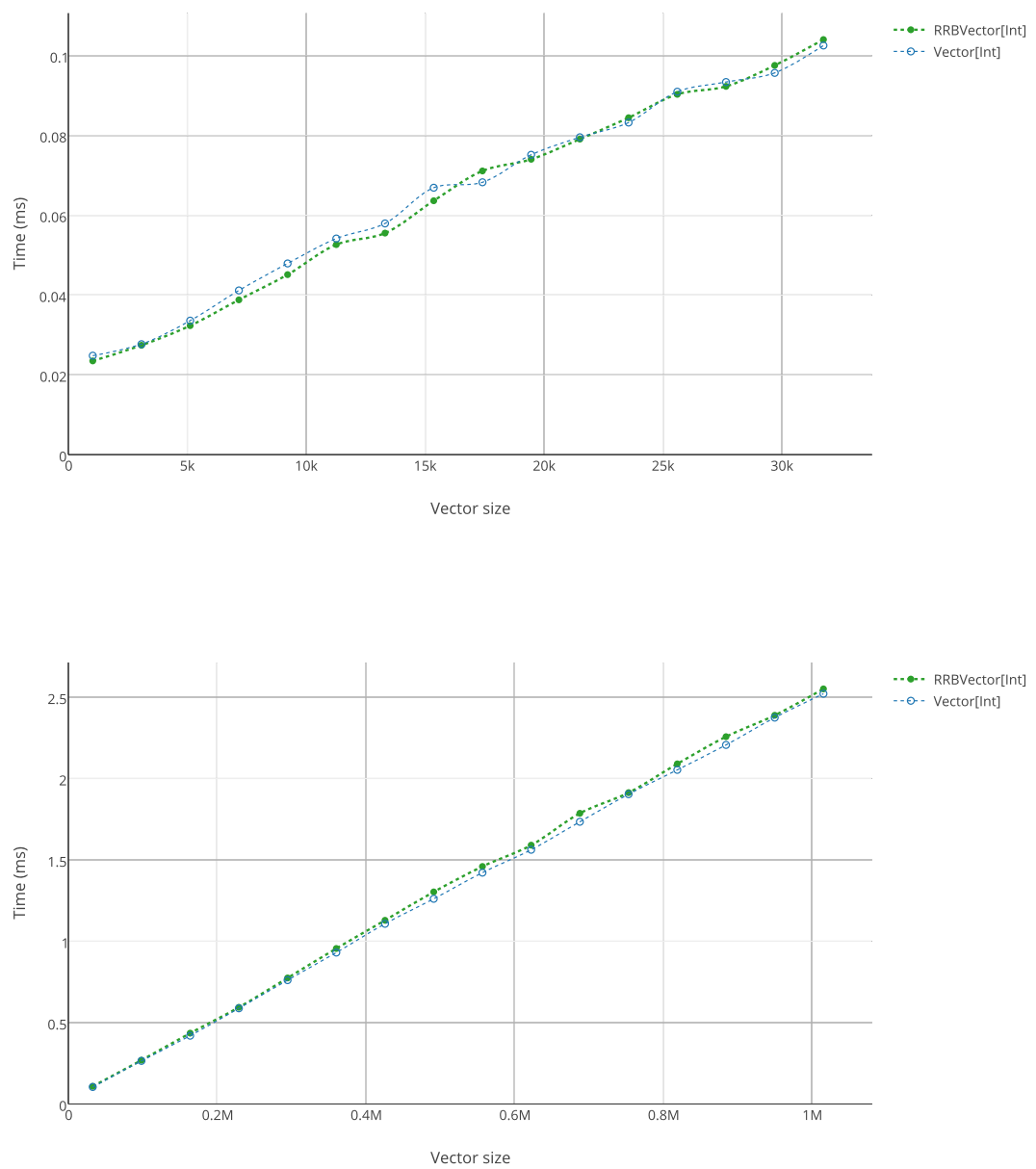


FIGURE 4.12: Execution time to build a vector of a given size.

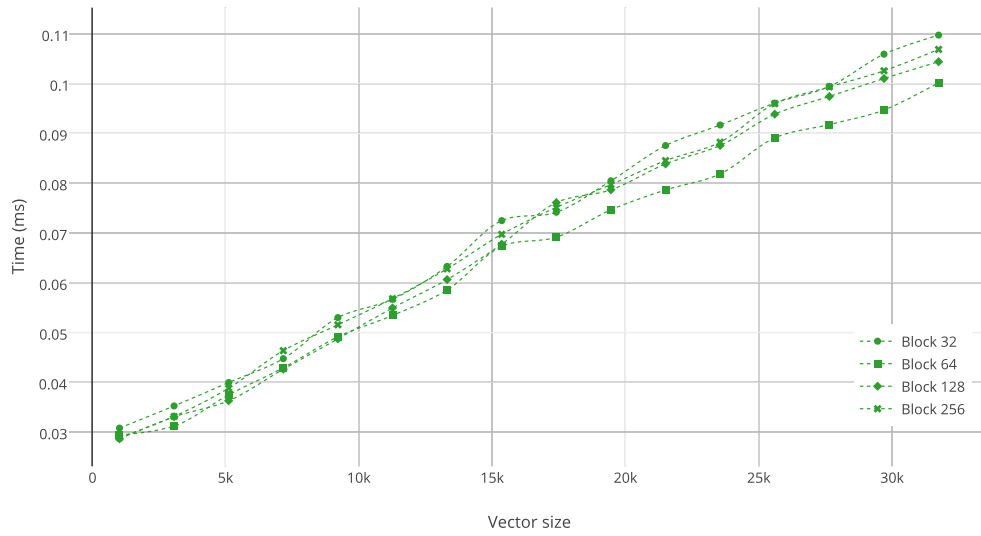


FIGURE 4.13: Execution time to build a vector of a given size. Comparing performances for different block sizes.

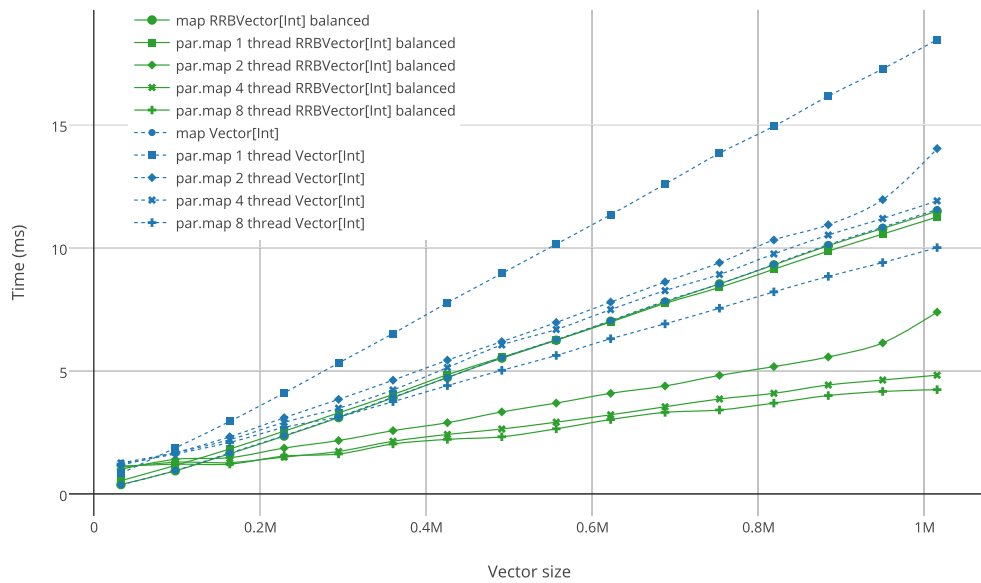


FIGURE 4.14: Benchmark on map and parallel map using the function $(x \Rightarrow x)$ to show the difference time used in the framework. This time represents the time spent in the splitters and combiners of the parallel collection (iterator and builder for the sequential version).

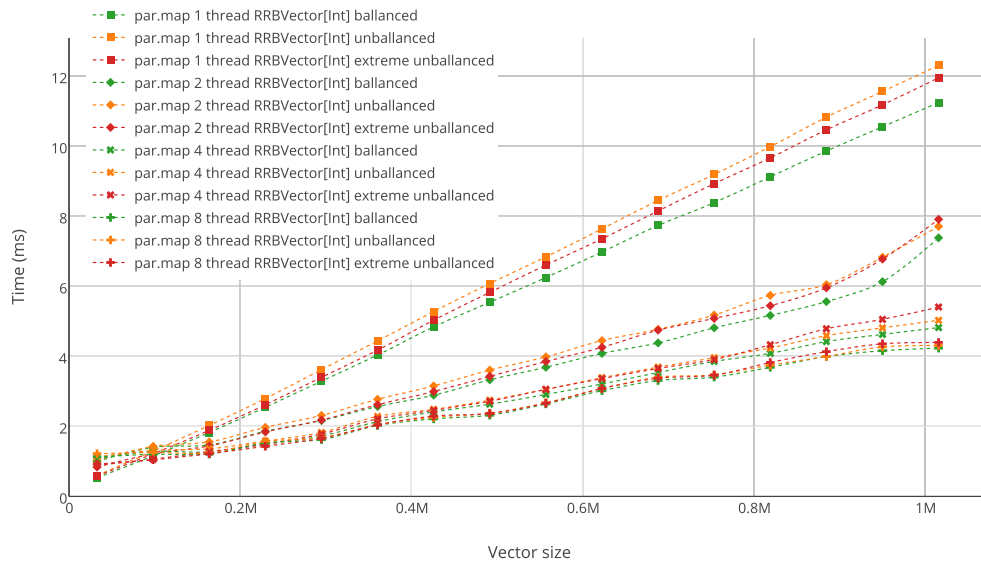


FIGURE 4.15: Benchmark on map and parallel map using the function $(x \Rightarrow x)$ to show the difference time used in the framework. This time represents the time spent in the splitters and combiners of the parallel collection.

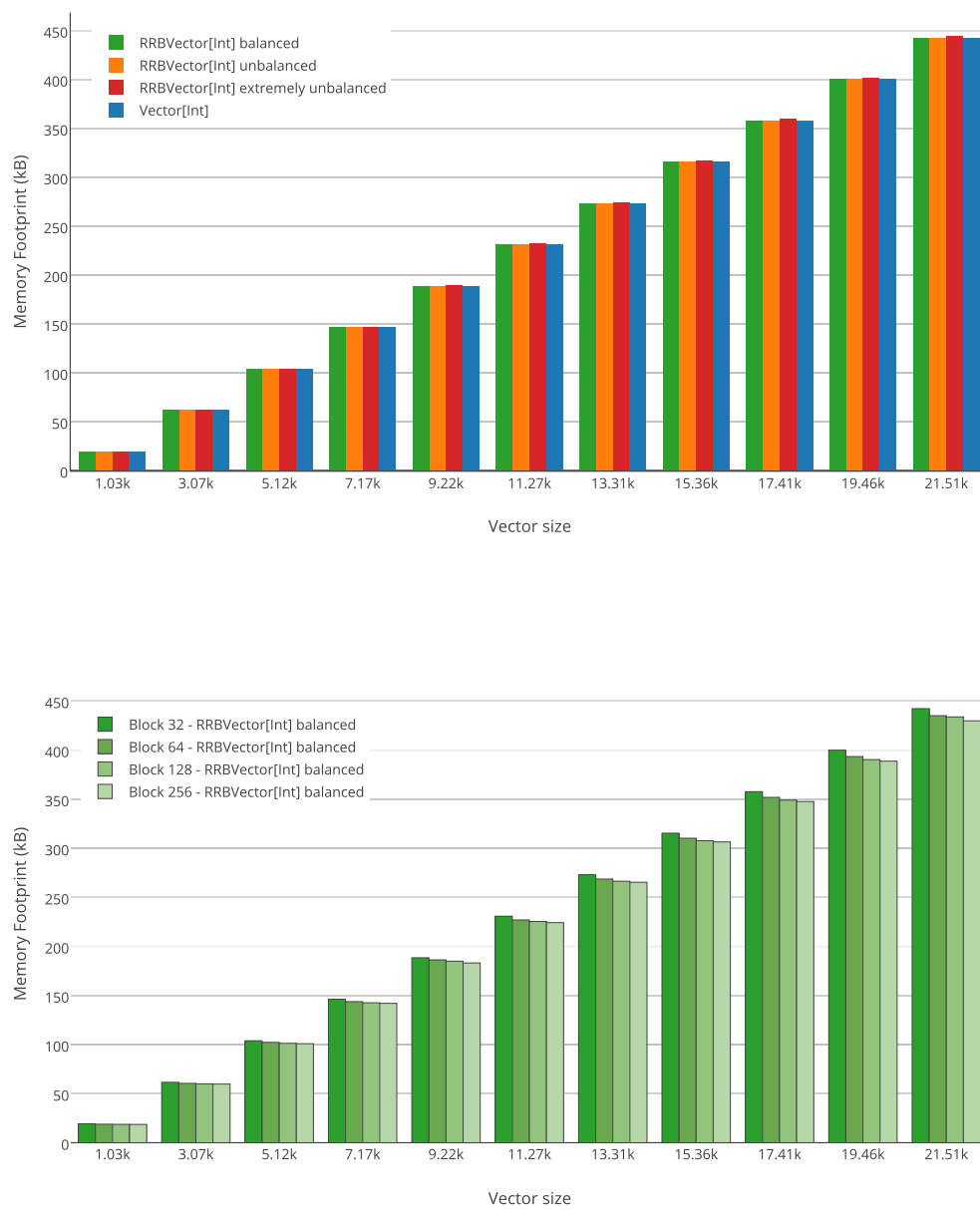


FIGURE 4.16: Memory Footprint

Chapter 5

Testing

5.1 Teststing correctness

5.1.1 Unit tests

5.1.2 Invariant Assertions

I

Chapter 6

Related Work

6.1 RRB-Vectors in Clojure

I

Chapter 7

Conclusions