École Polytechnique Fédérale de Lausanne

Master Thesis

Relaxed Radix Balanced Trees as Imutable 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

List of Figures

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

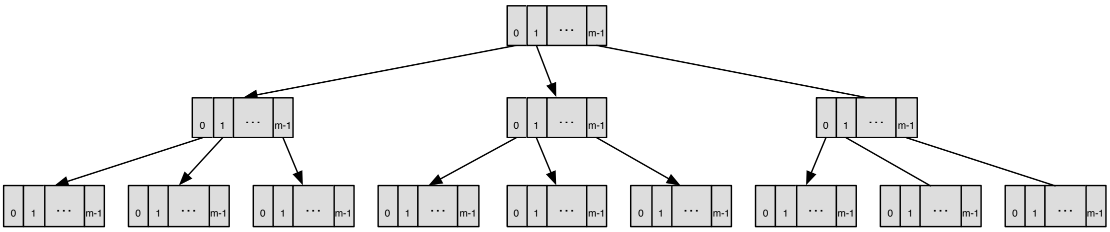


Figure 2.1: Radix Balanced Tree Structure

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

2.1.2.4 Splits

2.2 Parallel Vectors

2.2.1 Splitter Iterator

To divide the work into tasks for thread pool, a splitter is used to iterate over all elements of the collection. Splitters are a special kind of iterator that can be split at any time into some partition of the remaining elements. In the case of sequences the splitter should retain the original order. The most common implementation consists in dividing the remaining elements into two half.

The current implementation of the immutable parallel vector [1] uses the common division into 2 parts for it splitter. The drop and take operations are used divide the vector for the two new splitters.

2.2.2 Combiner Builder

Combiners are used to merge the results from different tasks (in methods like map, filter, collect, ...) into the new collection. Combiners are a special kind of builder that is able to merge to partial results efficiently. When it's impossible to implement efficient combination operation, usually a lazy combiner is used. The lazy combiner is one keeps all the it's sub-combiners in an array buffer and only when the end result is needed they are combined. This is a fairly efficient implementation but does not take advantage of parallelism.

The current implementation of the immutable parallel vector [1] use the lazy approach because of it's inefficient concatenation operation. One of the consequences of this is that the parallel operations will always be bounded by this sequential combination of elements, which can be beaten by the sequential version in many cases.

2.3 Relaxed Radix Balanced Vectors

2.3.1 Relaxed Tree structure

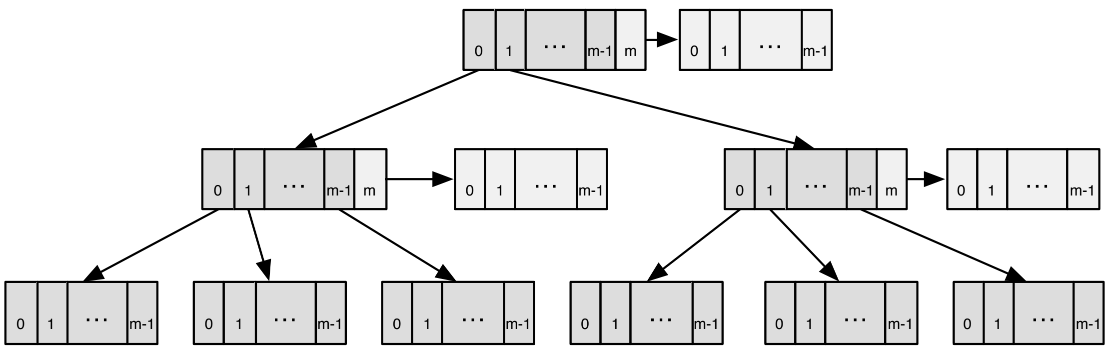


FIGURE 2.2: Radix Balanced Tree

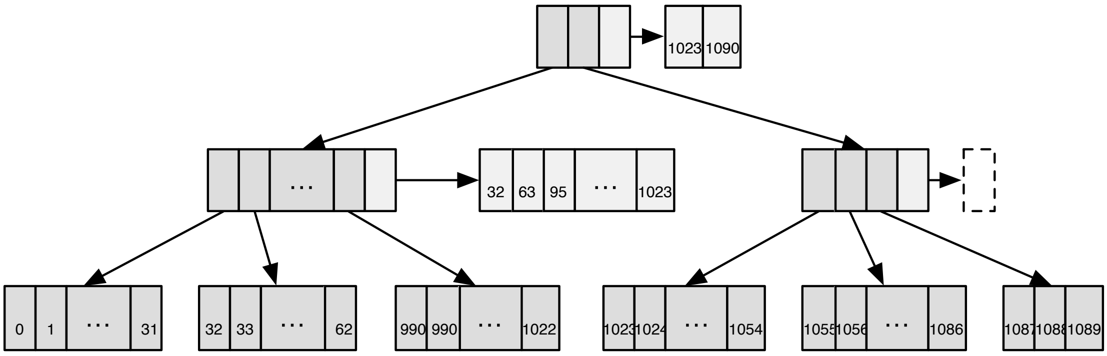


FIGURE 2.3: Relaxed radix example

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

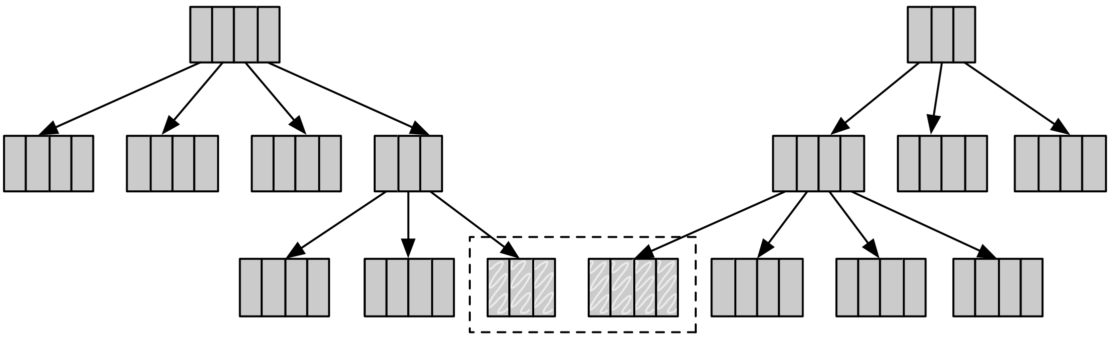


FIGURE 2.4: Concatenation example with blocks of size 4: Rebalancing level 0

Concatenation

Insert

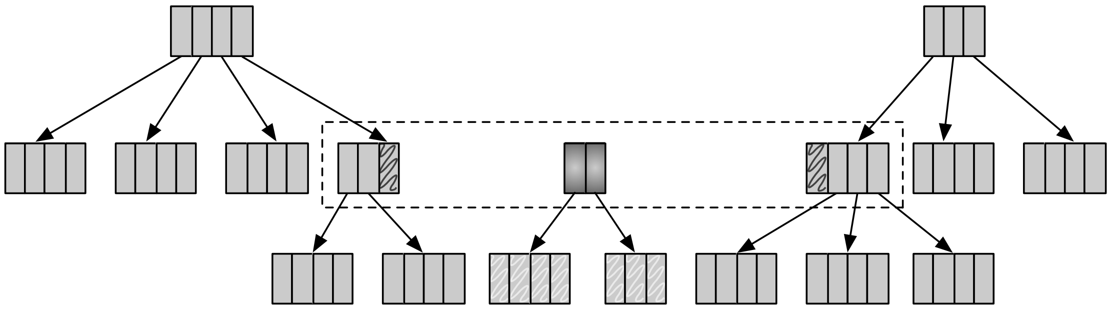


FIGURE 2.5: Concatenation example with blocks of size 4: Rebalancing level 1

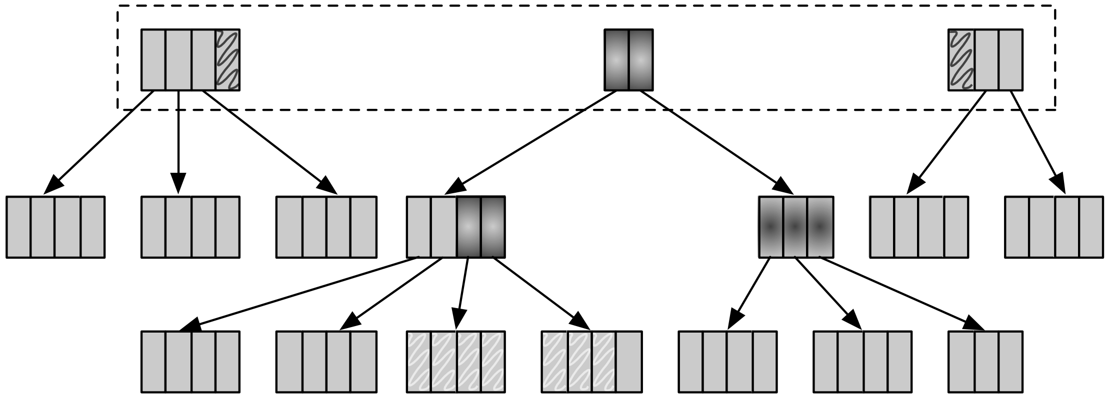


FIGURE 2.6: Concatenation example with blocks of size 4: Rebalancing level 2

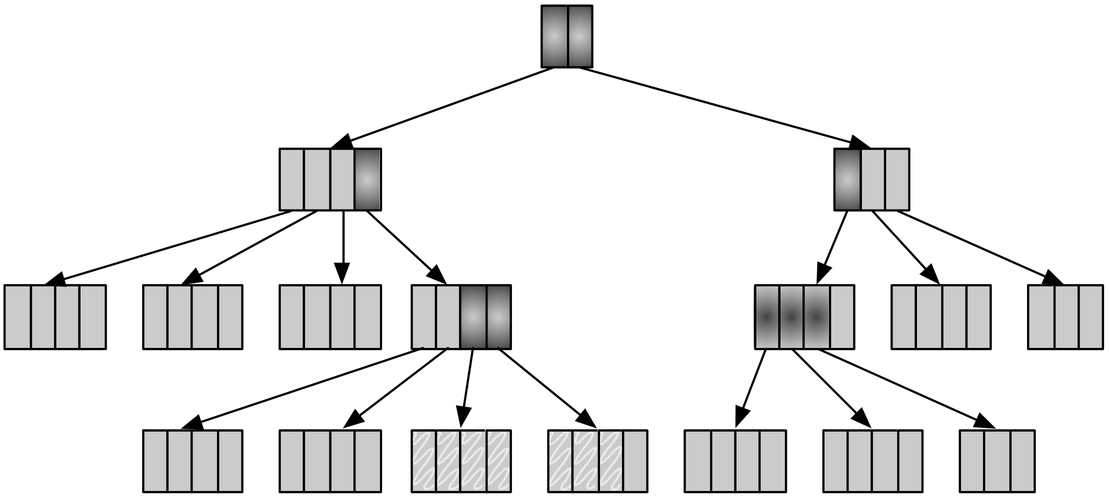


FIGURE 2.7: Concatenation example with blocks of size 4: Rebalancing level 3

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

$$526843 = 00\,\underbrace{00000}_{0}\,\underbrace{00000}_{0}\,\underbrace{10000}_{16}\,\underbrace{00010}_{2}\,\underbrace{01111}_{15}\,\underbrace{11011}_{27}$$

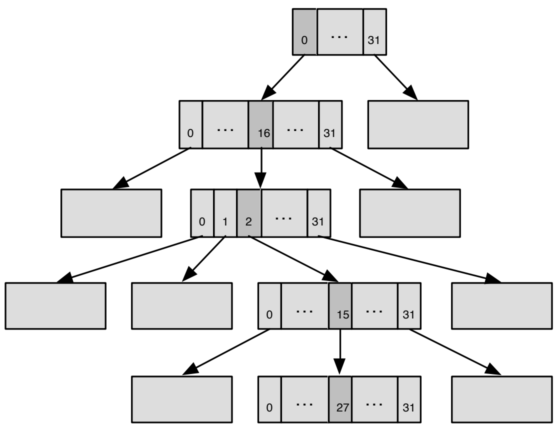


Figure 3.1: Accessing element at index 526843 in a tree of depth 5. Empty nodes represent collapses subtrees.

3.1.3 Abstractions

3.2 Displays

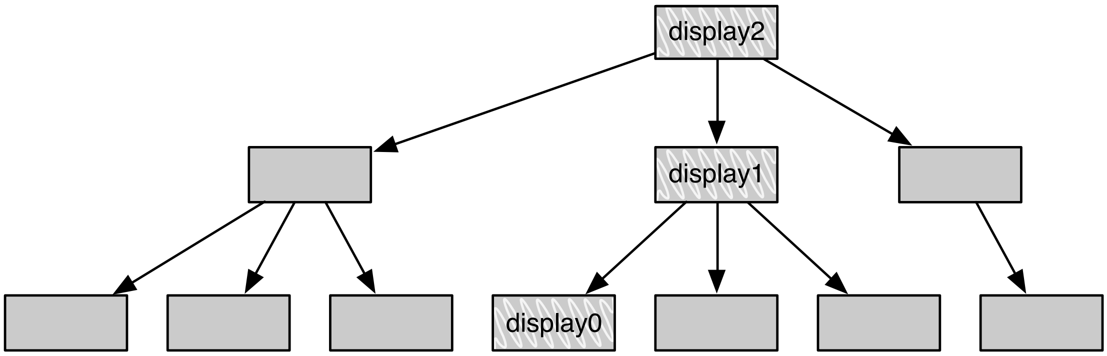


FIGURE 3.2: Displays

3.2.1 As cache

3.2.2 For transient states

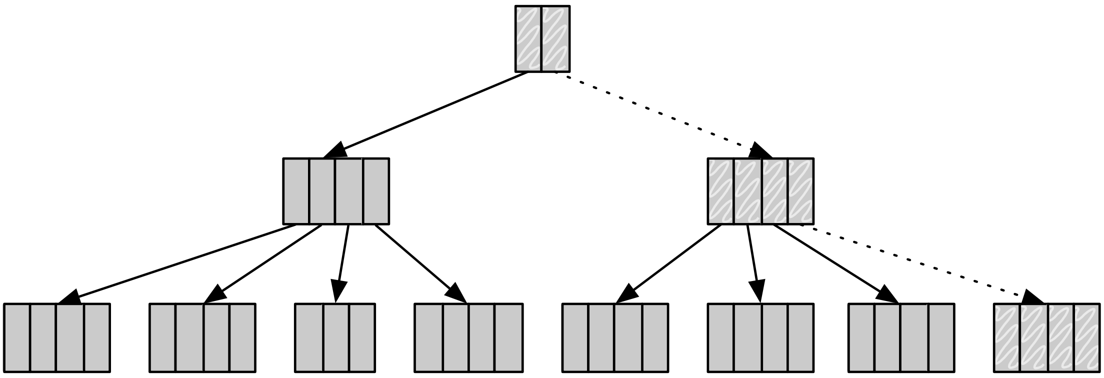


FIGURE 3.3: Radix Balanced Tree Transient state

3.3 Builder

3.4 Iterator

3.5 Relaxing the Radix

3.5.1 Relaxing Displays

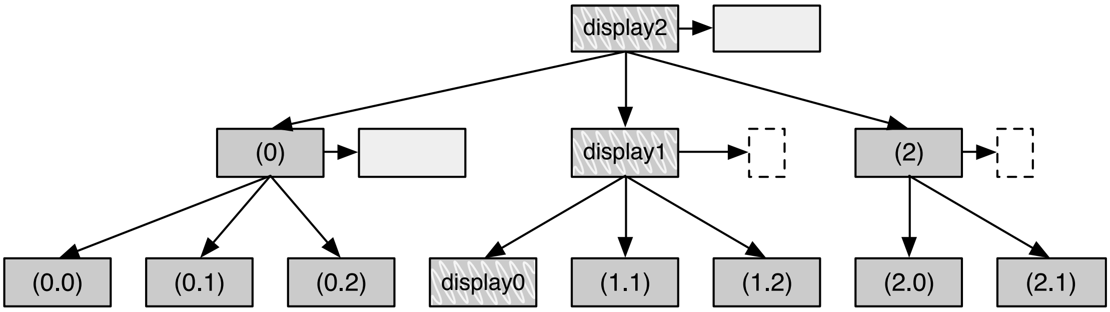
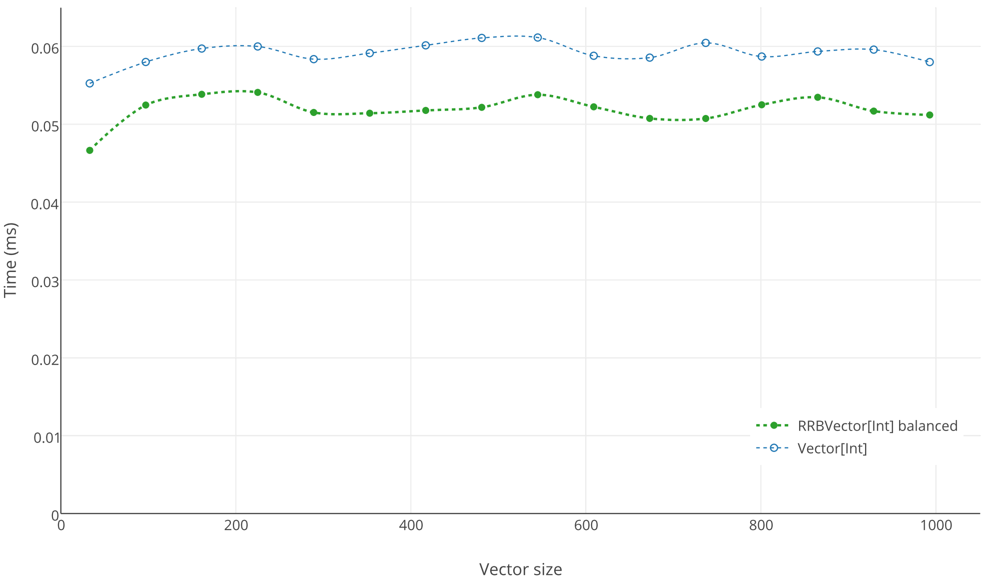


Figure 3.4: Radix Balanced Tree

3.5.2 Relaxing the Builder

3.5.3 Relaxing Iterator

I



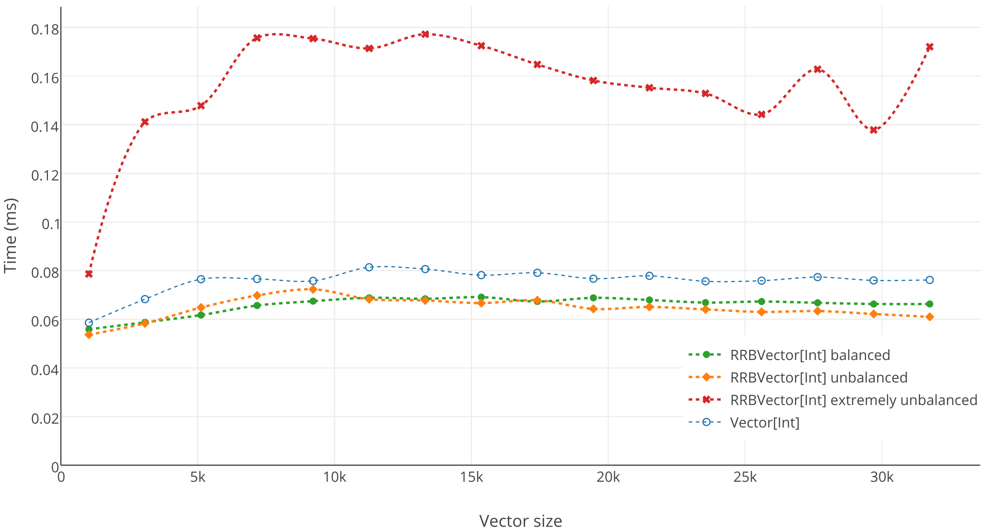


FIGURE 4.1: Time to execute 10k apply operations on sequential indices.

Chapter 4

Performance

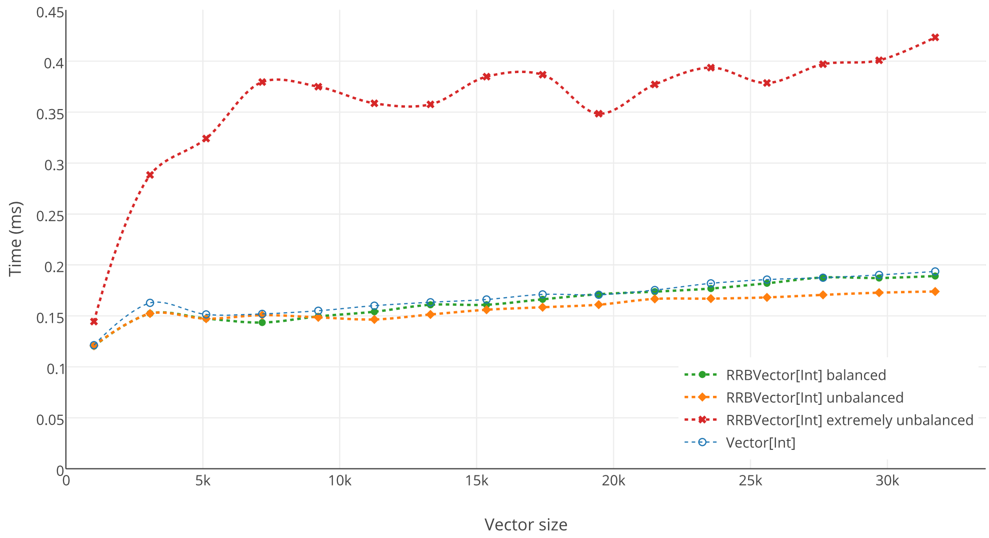


FIGURE 4.2: Time to execute 10k apply operations on random indices.

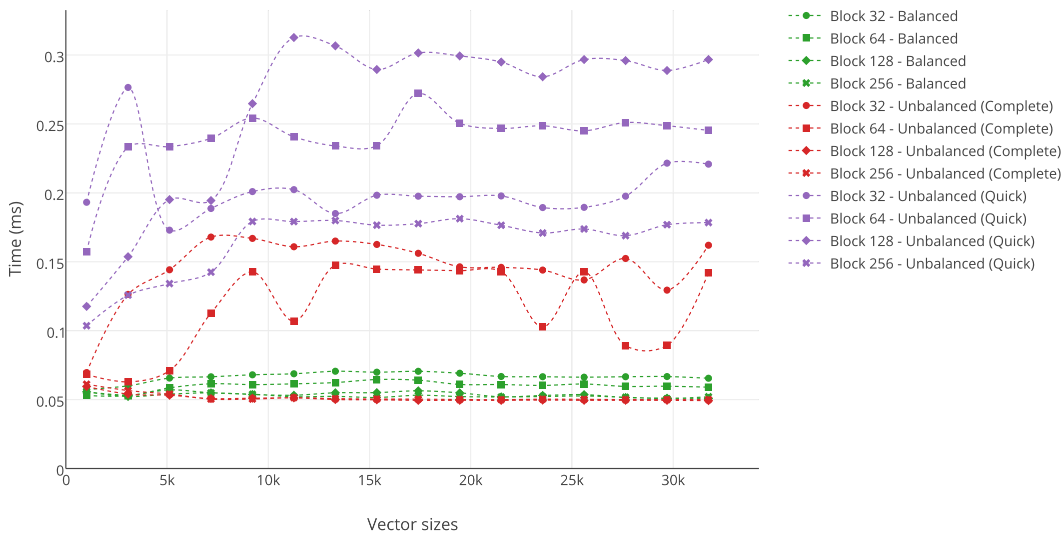


FIGURE 4.3: Time to execute 10k apply operations on sequential indices. Comparing performances for different block sizes and different implementation of the concatenation inner branch rebalancing (Complete/Quick).

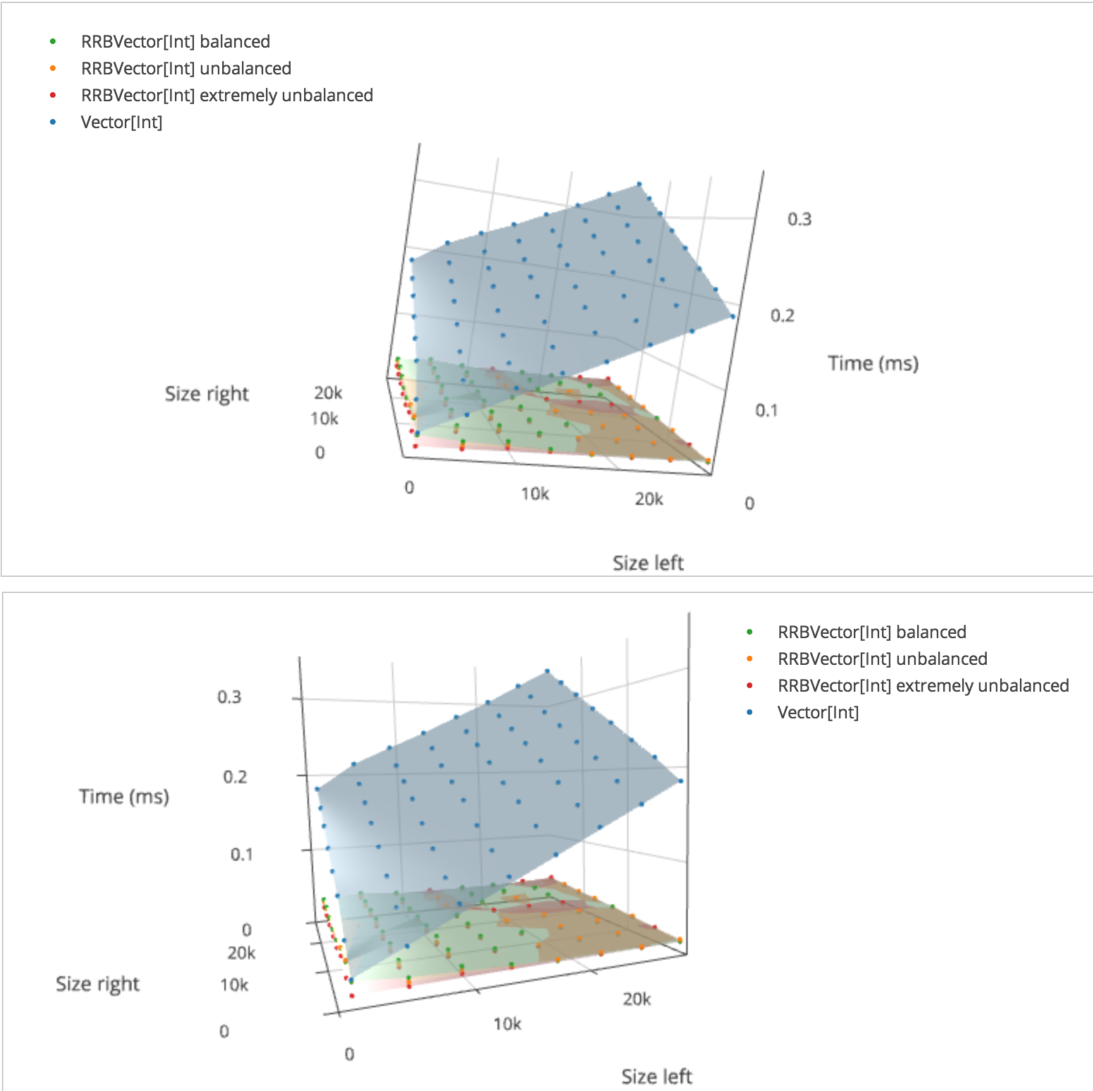
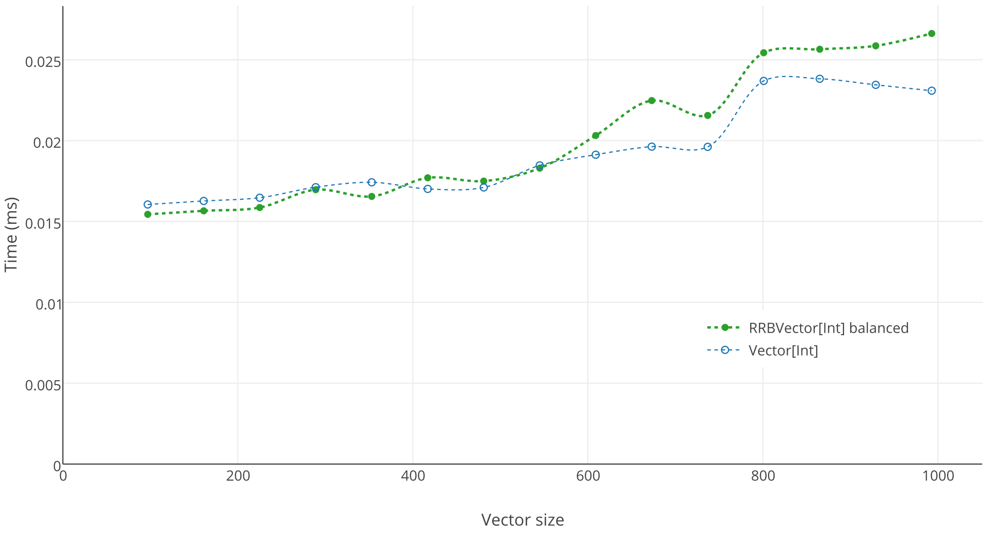


FIGURE 4.4: Execution time for a concatenation operation on two vectors. In theory (and in practice) Vector concatenation is $O(left+right)$ and the rrbVector concatenation operation is $O(log_{32}(left+right))$.



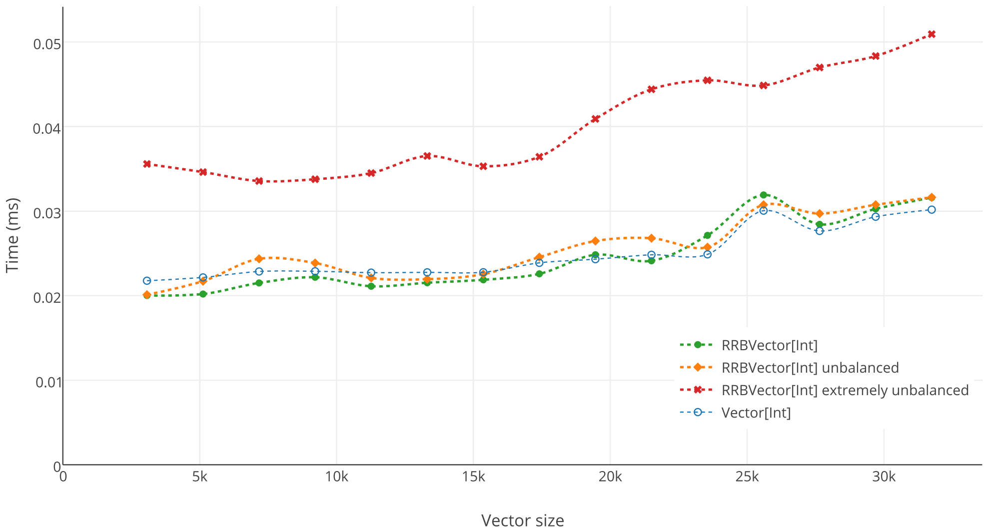


FIGURE 4.5: Time to execute 256 append operations. This shows the amortized cost of the append operation.

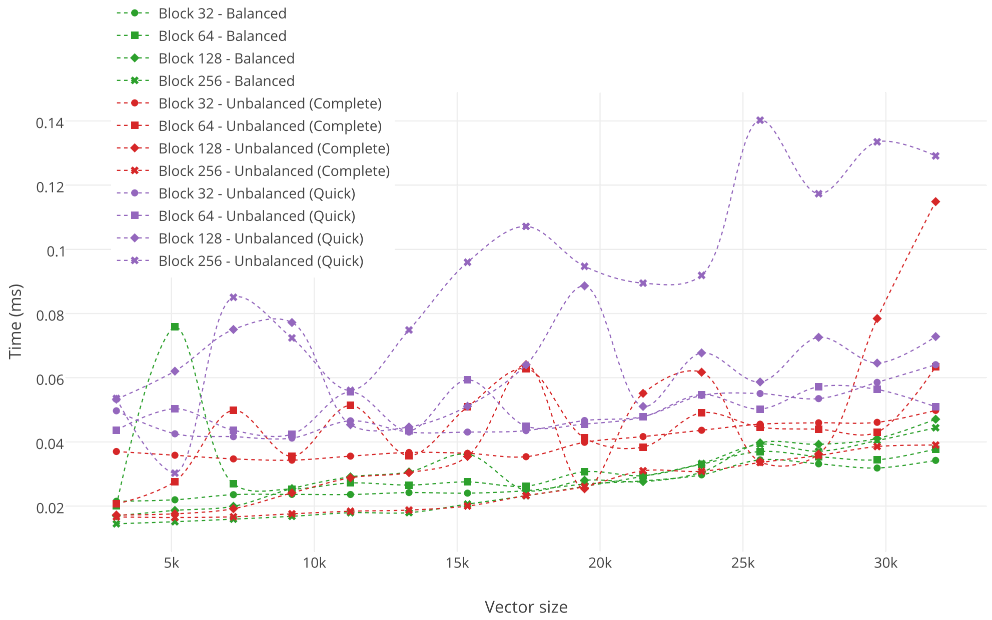


FIGURE 4.6: Time to execute 256 append operations. This shows the amortized cost of the append operation. Comparing performances for different block sizes and different implementation of the concatenation inner branch rebalancing (Complete/Quick).

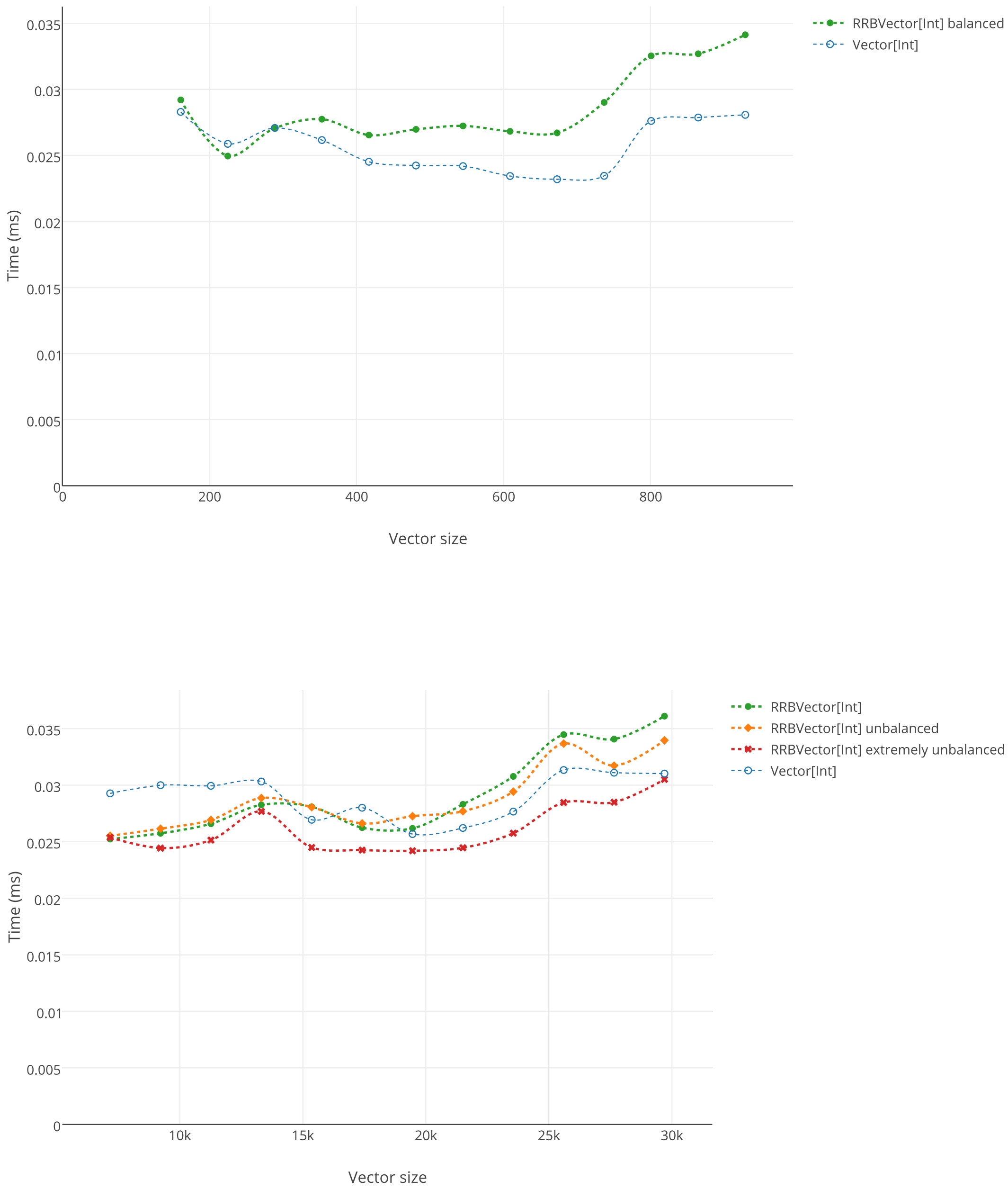


Figure 4.7: Time to execute 256 prepend operations. This shows the amortized cost of the prepend operation.

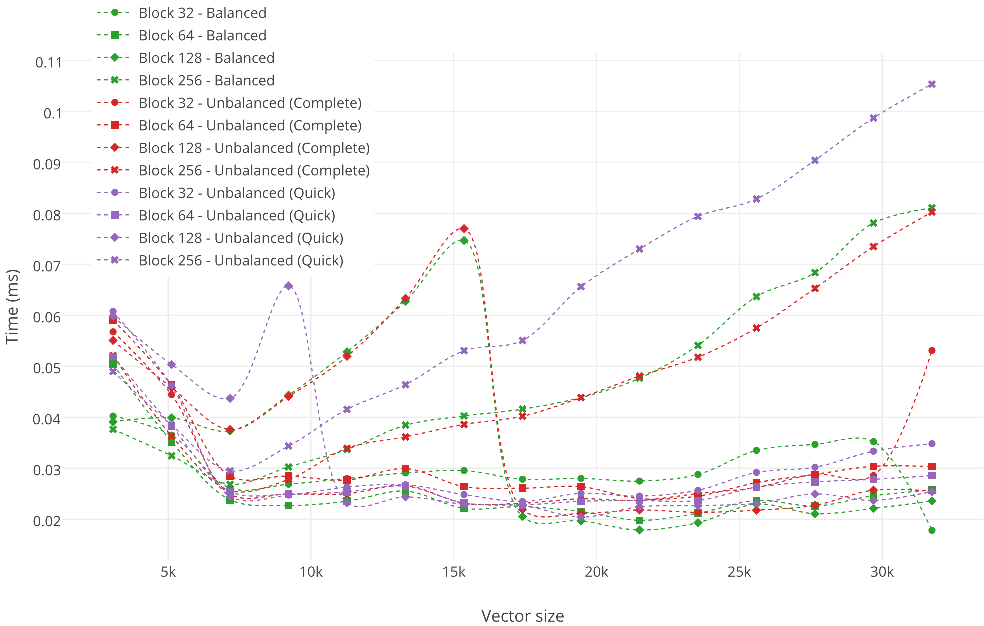
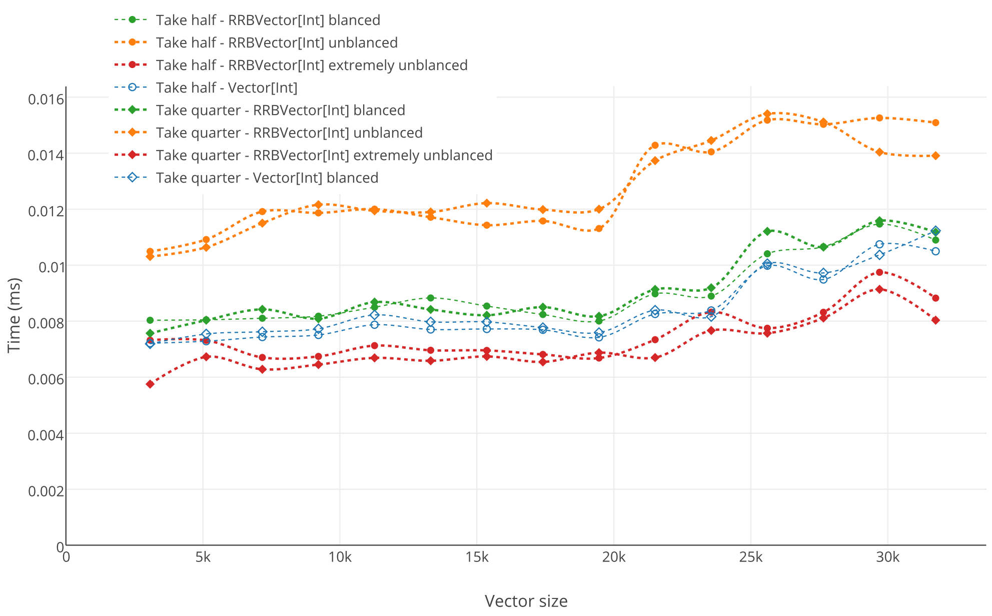


FIGURE 4.8: Time to execute 256 prepend operations. This shows the amortized cost of the append operation. Comparing performances for different block sizes and different implementation of the concatenation inner branch rebalancing (Complete/Quick).



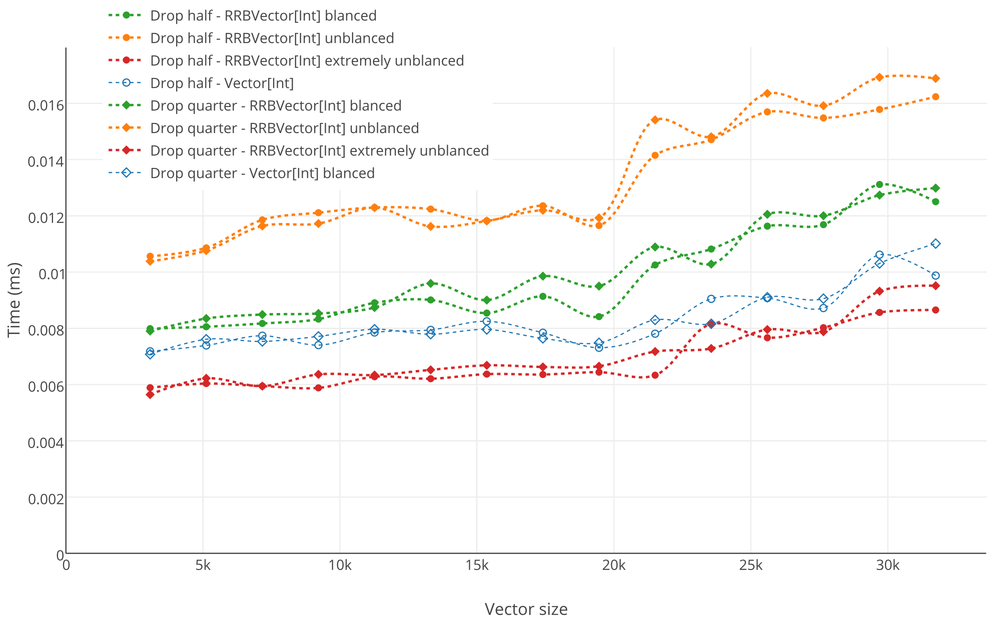
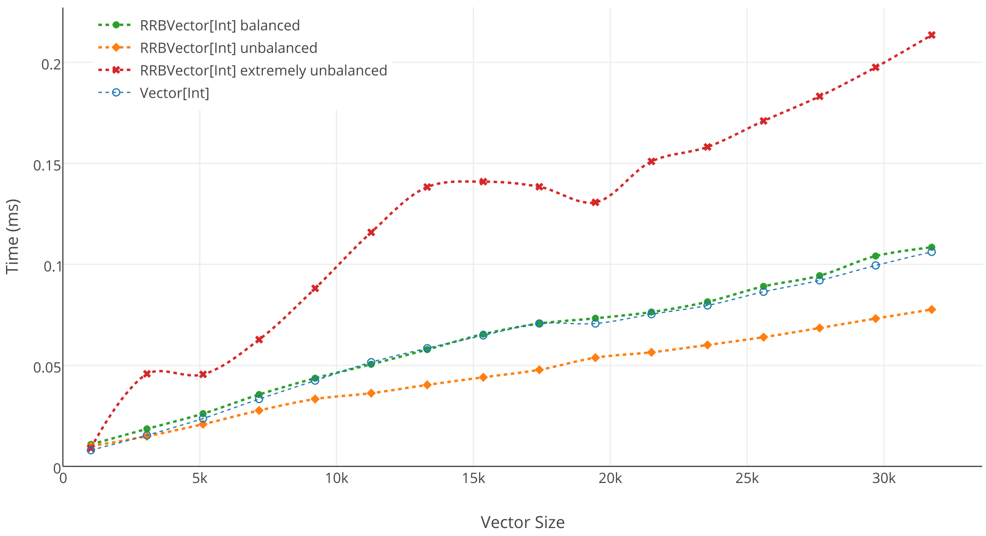


FIGURE 4.9: Execution time of take and drop.



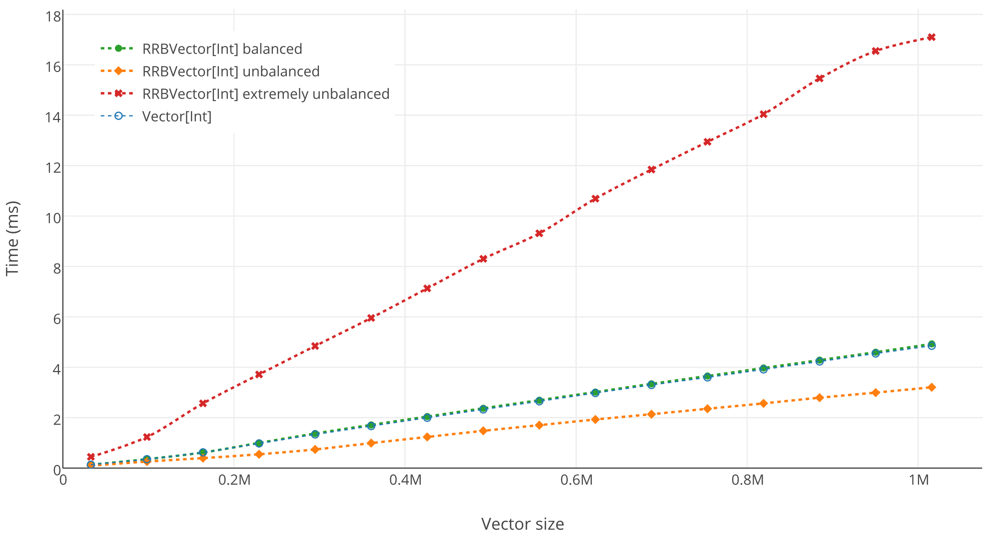


FIGURE 4.10: Excecution time to iterate through all the elements of the vector.

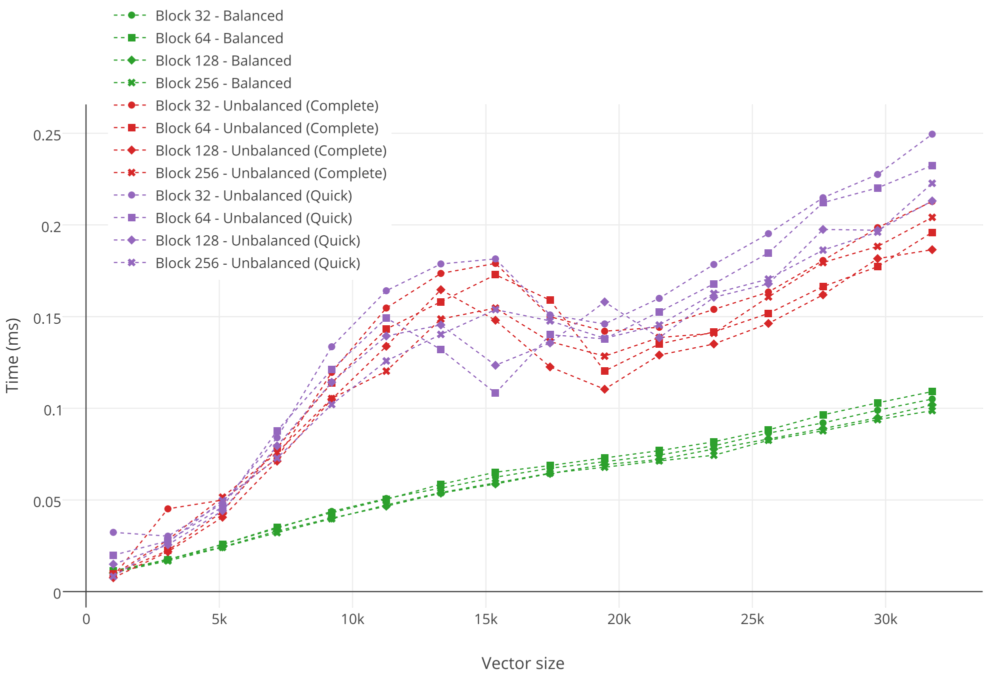
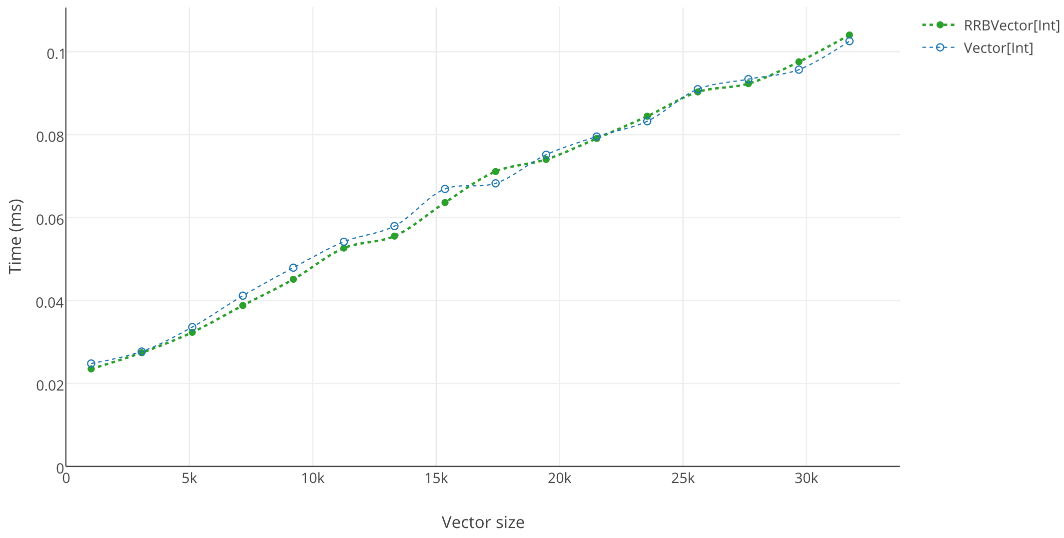


FIGURE 4.11: Excecution time to iterate through all the elements of the vector. Comparing performances for different block sizes and different implementation of the concatenation inner branch rebalancing (Complete/Quick).



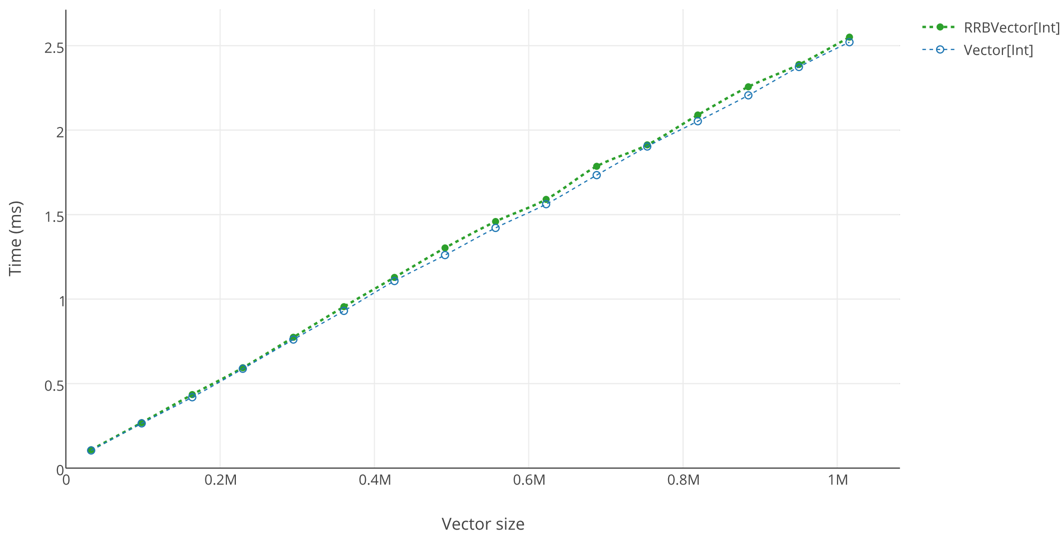


FIGURE 4.12: Execution time to build a vector of a given size.

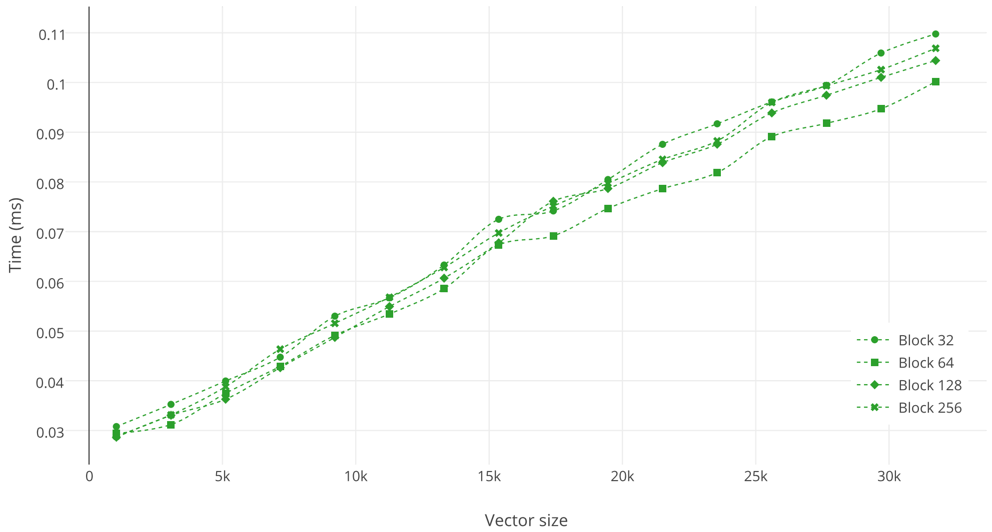


FIGURE 4.13: Execution time to build a vector of a given size. Comparing performances for different block sizes.

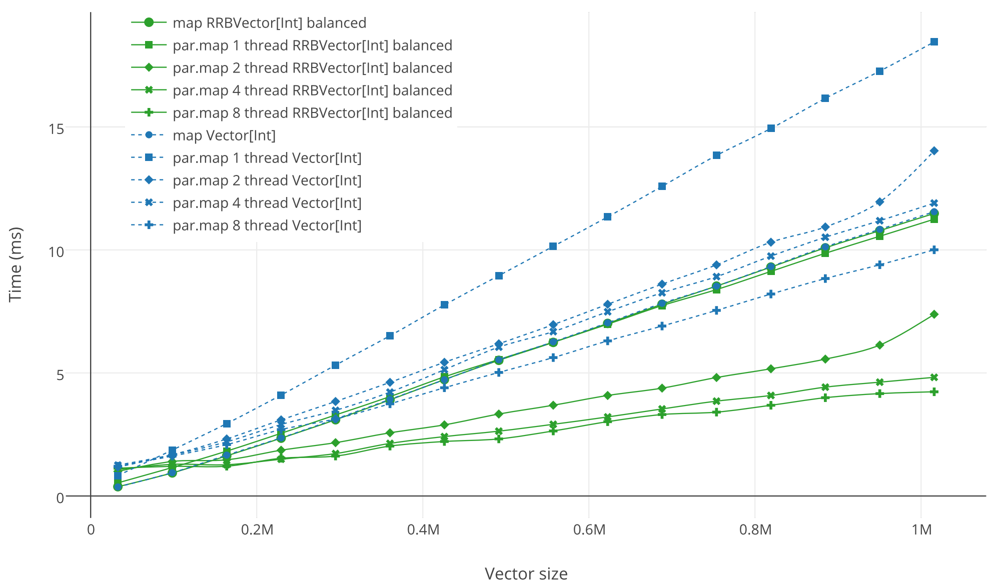


FIGURE 4.14: Benchmark on map and parallel map using the function (x=>x) to show the difference time used in the framework. This time represents the time spent in the splitters and combiners of the parallel collection (iterator and builder for the sequential version).

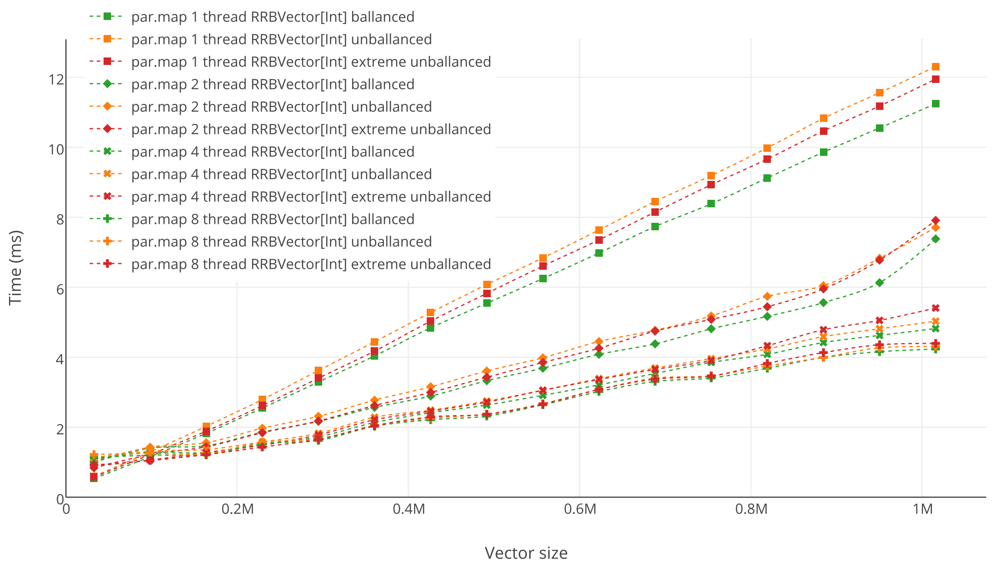


FIGURE 4.15: Benchmark on map and parallel map using the function (X=>X) to show the difference time used in the framework. This time represents the time spent in the splitters and combiners of the parallel collection.

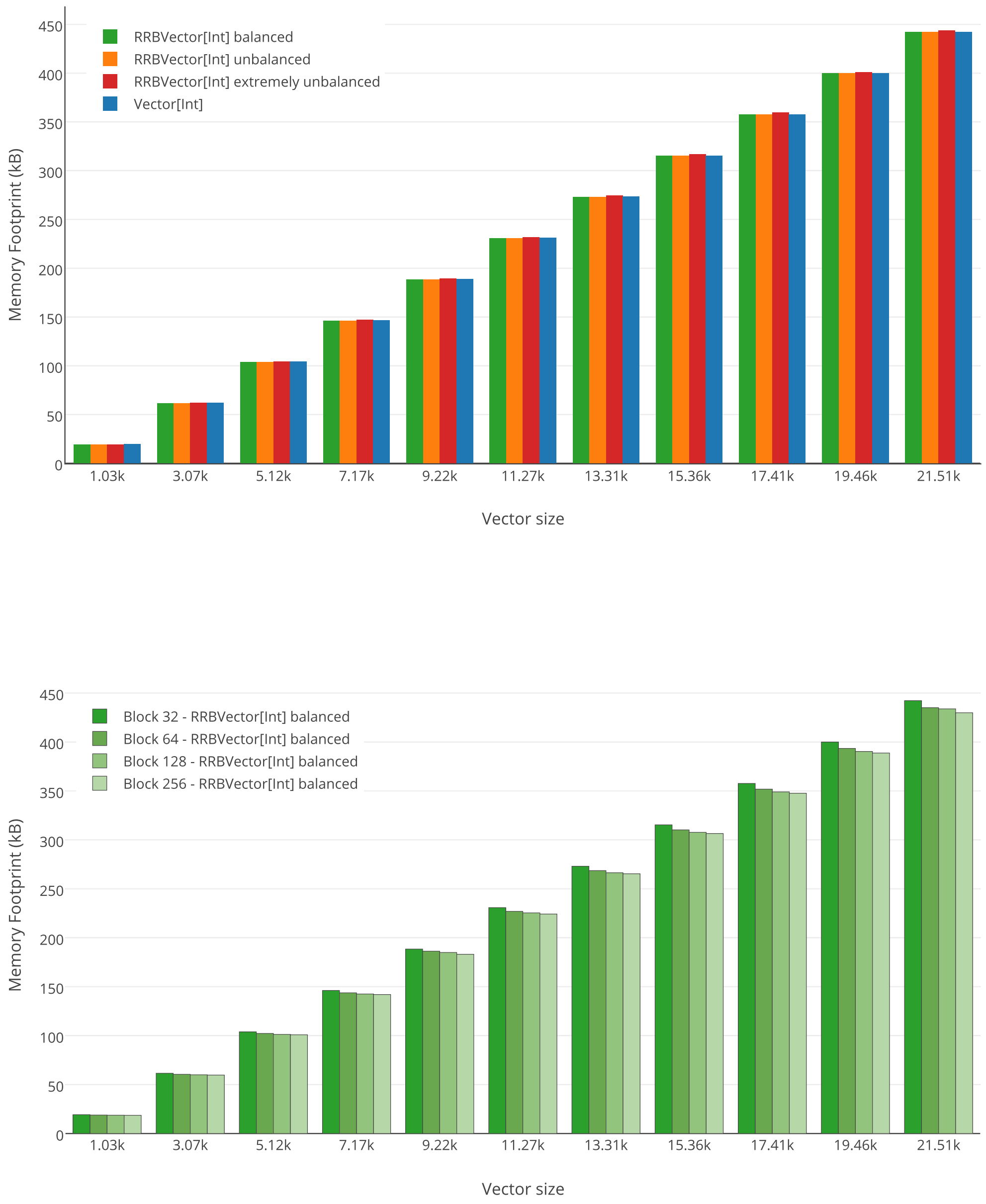


Figure 4.16: Memory Footprint

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

Bibliography

[1] GitHub - Scala 2.11 - ParVector.scala. https://github.com/scala/scala/blob/f4267ccd96a9143c910c66a5b0436aaa64b7c9dc/src/library/scala/collection/parallel/immutable/ParVector.scala.