École Polytechnique Fédérale de Lausanne

Master Thesis

Turning Relaxed Radix Balanced Vector from Theory into Practice for Scala Collections

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

Turning Relaxed Radix Balanced Vector
from Theory into Practice
for Scala Collections

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

JVM	**J**ava **V**irtual **M**achine
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 and Operations

2.1 Radix Balanced Vectors

2.1.1 Tree structure

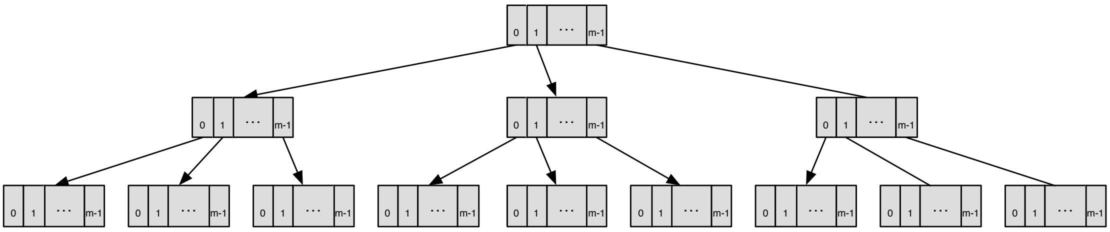


Figure 2.1: Radix Balanced Tree Structure

2.1.2 Operations

The immutable `Vector` [2] is a subtype of `IndexedSeq` in the current Scala collections, as such it implements all operations defined on it. Most operations are implemented using a set of core operations that can be implemented efficiently on RB-Trees, those operations are: `apply`, `updated`, `append`, `prepend`, `drop` and `take`.

The performances of most operations on RB-Trees have a computational complexity of $O(log_{32}(n))$, equivalent to the height of the tree. This is usually referred as effective constant time because for 32 bit signed indices the height of the tree will bounded by a small constant (6.2 in the case of books of size 32). This bound is reasonable to ensure that in practice the operations behave like constant time operations.

In this section the operations will be presented on a high level and without optimisation. These reflect the worst case scenario or the base implementation that is used before adding optimizations (see chapter 3). To improve performance of some operations, displays are used on branches of the RB-Tree (see section 3.2). To improve the complexity of operations to amortized constant time (instead of effective constant time[1]) for some operation sequences the vectors are augmented with transient states.

2.1.2.1 Apply

The `apply` operation in indexed sequences is defined as the operation that gets the element located at some index. This is the main element access method and it is used in other methods such as `head` and `last`.

```
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)
}
```

The base implementation on RB-Trees of this operation requires a simple traversal from the root to the leaf containing the element. Where the path taken is defined by the index of the element and extracted using some efficient bitwise operations (see section 3.1.2). With this traversal of the tree, the complexity of the `apply` operation is $O(log_{32}(n))$. This complexity can be reduced for some subset of elements by adding displays.

2.1.2.2 Updated

The `updated` method returns a new immutable `Vector` with one updated element at a given index. This is the core operation to modify the contents of the vector.

[1] In this context, effective constant time has a large but bounded constant time complexity, while amortized constant time tends to have a bound closer to a one level tree update.

On immutable RB-Trees the updated operation has to recreate the whole branch from the root to the element being updated. The update of the leaf creates a fresh copy of the leaf with the updated element. Then the parent of the leaf also need to create a new node with the updated reference to the new leaf, and so on up to the root.

```
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), ...)
}
```

Therefore the complexity of this operation is $O(log_{32}(n))$, for the traversal and creation of each node in the brach. This operation can be improved to have amortized constant time updates for local updates using displays with transient states. This way the parent nodes are only updated lazily when absolutely necessary. For example, if some leaf has all it's elements updated from left to right, the leaf will be copied as many times as there are updates, but the parent of that leaf does not change during those operations.

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 full 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.

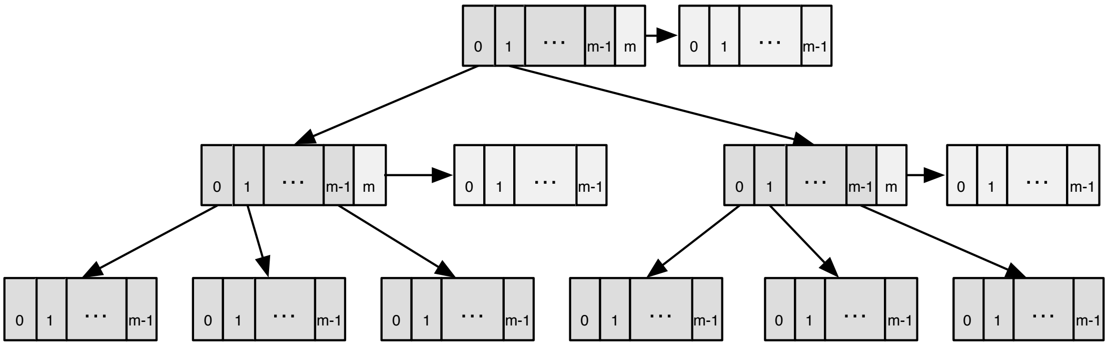


Figure 2.2: Radix Balanced Tree

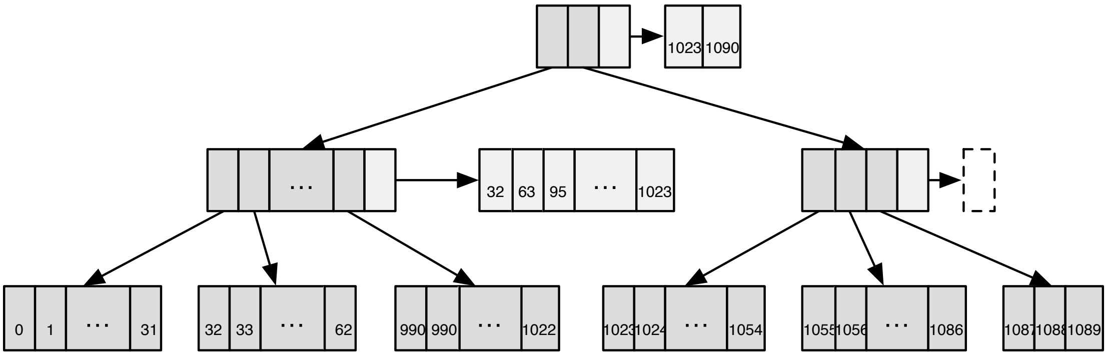


Figure 2.3: Relaxed radix example

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

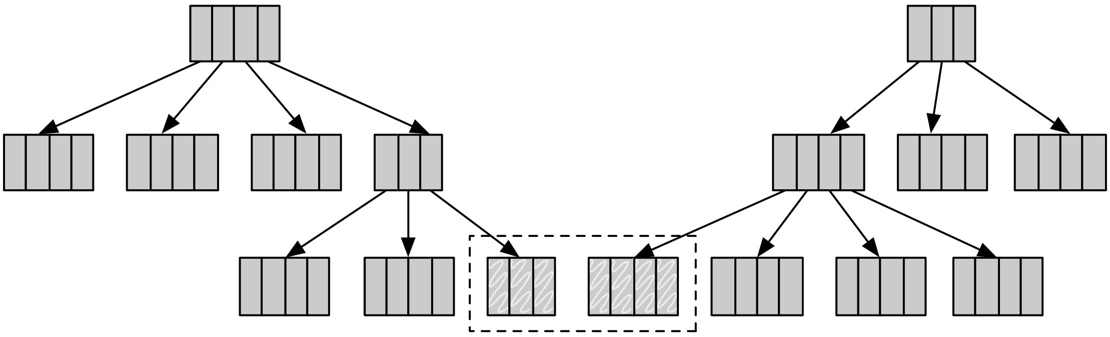


Figure 2.4: Concatenation example with blocks of size 4: Rebalancing level 0

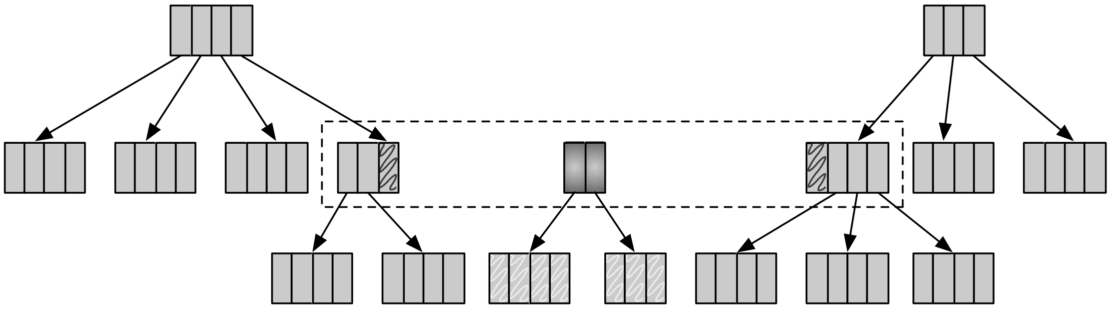


Figure 2.5: Concatenation example with blocks of size 4: Rebalancing level 1

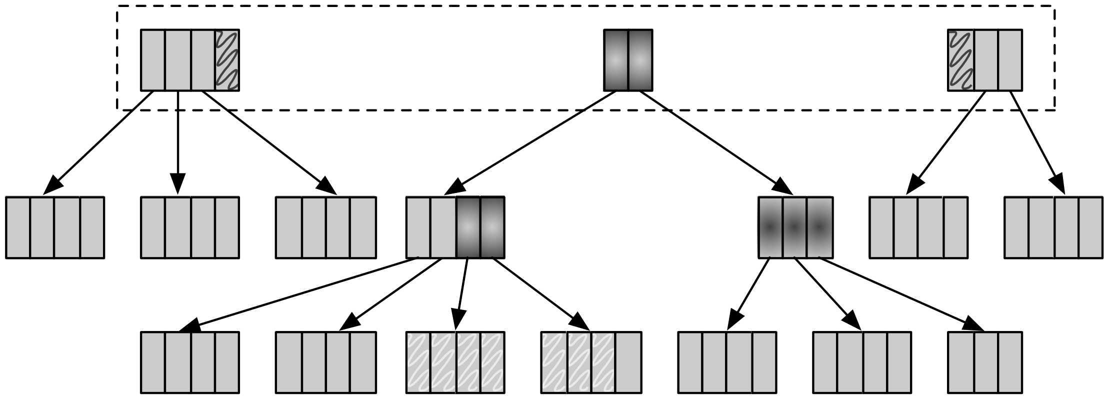


Figure 2.6: Concatenation example with blocks of size 4: Rebalancing level 2

Concatenation

Insert

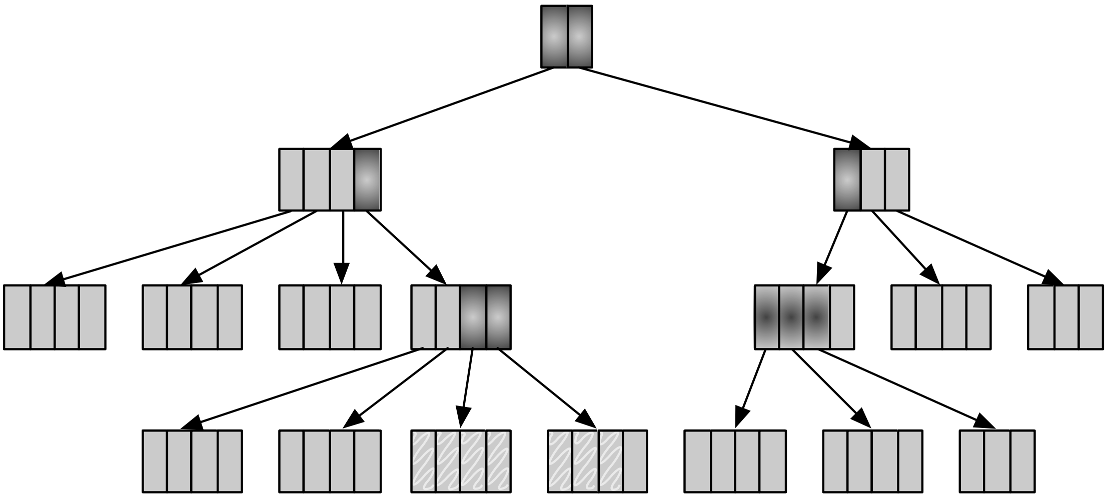


Figure 2.7: Concatenation example with blocks of size 4: Rebalancing level 3

2.3.2.4 Splits

2.3.2.5 Parallel Vector

I

Chapter 3

Optimizations

3.1 Where is time spent?

3.1.1 Arrays

Most of the memory used in the vector data structure is composed of arrays. The three key operations used on these arrays: array creation, array update and array access. The arrays are used as immutable arrays, as such the update operations are only allowed when the array is initialised. This also implies that each time there is a modification on some part of an array, a new array must be created and all the old elements copied.

The size of the array will affect the performance of the vector. With larger blocks the access times will be reduced because the depth of the tree will decrease. But, on the other hand, increasing the size of the block will make slow down the update operations. This is a direct consequence of the need to copy the entire array for a single update.

3.1.2 Computing indices

Computing the indices in each node while traversing or modifying the vector is key in performance. This performances is gained by using low level binary computations on the indices in the case where the tree is balanced. And, using precomputed sizes in the case where the balance is relaxed.

Radix Assuming that the tree is full, elements are fetched from the tree using radix search on the index. As each node has a branching of 32, the index can be split bitwise in blocks of 5 ($2^5 = 32$) and used to know the path that must be taken from the root down to the element. The indices at each level L can be computed with $(index >> (5 \cdot L))\&31$. For example the index 526843 would be:

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

```
def getSubIndex(indexInTree: Int, level: Int): Int =
  (index >> (5*level)) & 31
```

This scheme can be generalised to any block size m where $m = 2^i$ for $0 < i \leq 31$. The formula would be $(index >> (m \cdot L))\&((1 << m) - 1)$. It is also possible to generalise for other values of m using the modulo, division and power operations. In that case the formula would become $(index/(m^L))\%m$. This last generalisation is not used because it reduces sightly the performance and it complicates other index manipulations.

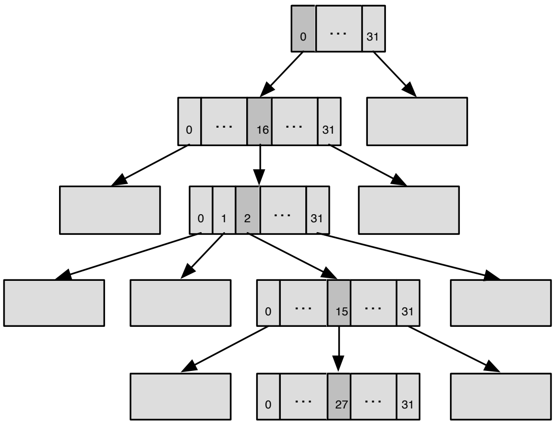


Figure 3.1: Accessing element at index 526843 in a tree of depth 5. Empty nodes represent collapses subtrees.

Relaxing the Radix When the tree is relaxed it is not possible to know the subindices from index. That is why we keep the sizes array in the unbalanced nodes. This array keeps the accumulated sizes to make the computation of subindices as trivial as possible. The subindex is the same as the first index in the sizes array where $index < sizes[subindex]$. The simplest way to find this subindex is by a linearly scanning the array.

```
def getSubIndex(sizes: Array[Int], indexInTree: Int): Int = {
    var is = 0
    while (sizes(is) <= indexInTree)
      is += 1
    is
}
```

For small arrays (like blocks of size 32) this will take be faster than a binary search because it takes advantage of the cache lines. If we would consider using bigger block sizes it would be better to use a hybrid between binary and linear search.

To traverse the tree down to the leaf where the index is, the subindices are computed from the sizes as long as the tree node is unbalanced. If the node is balanced, then the more efficient radix based method is used from there to the leaf. To avoid the need of accessing and scanning an additional array in each level.

3.1.3 Abstractions

3.2 Displays

As base for optimizations, the vector object keeps a set of fields to track one branch of the tree. They are named with using the level number from 0 up to the maximum possible level. In the case of blocks of size 32 the maximum level used is 5 [1], they are allocated by default and nulled if the tree if shallower. The highest non null display is and replaces the root field. All displays bellow the root are never null. This implies that the vector will always be focused on some branch.

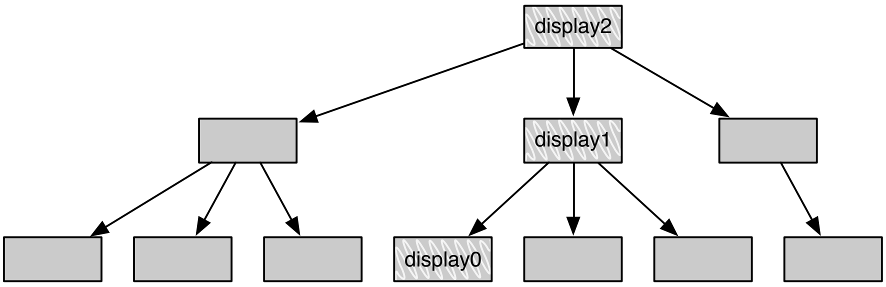


Figure 3.2: Displays

[1] As in practice, only the 30 bits of the index are used.

To know on which branch the vector is focused there is also a `focus` field with an index. This index is the index of any element in the current `display0`. This index represents the radix indexing scheme of node subindices described in 3.1.2.

To follow the simple implementations scheme of immutable objects in concurrent contexts, the focus is also immutable. Therefore each vector object will have a single focused branch during its existence[2]. Each method that creates a new vector must decide which focus to set.

3.2.1 As cache

One of the uses of the displays is as a cached branch. If the same leaf node is used in the following operation, there is no need for vertical tree traversal which is key to amortize operation to constant time. In the case another branch in needed, then it can be fetched from the lowest common node of the two branches.

To know the which is the level of the lowest common node in a vector of block size 2^m (for some consistent m), only the `focus` index and the index being fetched are needed. The operation $index \veebar focus$ will return a number is bounded to the maximum number of elements in a tree of that level. The actual level can be extracted with some if statements. This operation bounded by the same number of operations that will be needed to traverse the tree back down through the new branch.

```
def getLowestCommonLevel(index: Int, focus: Int): Int = {
  val xor = index ^ focus
  if (xor < 32 /*(1<<5)*/ ) 0
  else if (xor < 1024 /*(1<<10)*/ ) 1
  else if (xor < 32768 /*(1<<15)*/ ) 2
  ...
  else 5
}
```

When deciding which will be the focused branch of a new vector two heuristics are used for this: If there was an update operation on some branch where that operations could be used again, that branch is used as focus. If the first one cant be applied, the display is set to the first element as this helps key collection operations such as `iterator`.

[2] The display focus may change during the initialisation of the object as optimisation of some methods

3.2.2 For transient states

Transient states is the key optimisation to get append, prepend and update to amortized constant time. It consists in decoupling the tree by creating an equivalent tree that does not contain the edges on the current focused branch. The information missing in the edges of the tree is represented and can be reconstructed from the displays. In the current version of the collections vector [2] this state is identified by the `dirty` flag.

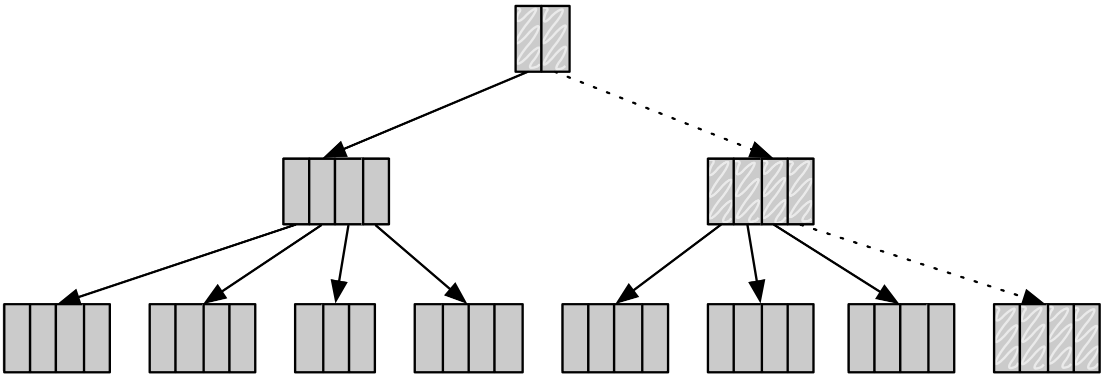


FIGURE 3.3: Transient Tree with current focus displays marked in white and striped nulled edges.

Without transient states when some update is done on a leaf, all the branch must be updated. On the other hand, if the state is transient, it is possible to update only the subtree affected by the change. In the case of updates on the same leaf, only the leaf must be updated. When appending or prepending, $\frac{31}{32}$ operations must only update the leaf, then $\frac{31}{1024}$ need to update two levels of the tree and so on. These operations will thus be amortized to constant ($\sum_{k=1}^{\infty} \frac{k*31}{32^k} = \frac{32}{31}$ block updates per operation) time if they are executed in succession.

There is a cost associated to the transformation from normal state to transient state and back. This cost is equivalent to one update of the focused branch. The transient state operations only start gaining performance on the normal ones after 3 consecutive operations. With 2 consecutive operations they are matched and with 1 there is a loss of performance.

3.2.3 Relaxing the Displays

When relaxing the tree balance it is also necessary to relax the displays. This is mainly due to the loss of a simple way to compute the lowest common node on unbalanced trees. Computing the node requires now the additional sizes information located in each unbalanced node. As such it is necessary to access the nodes to be abel to compute the lowest common node, and there is a loss in performance due to increased memory accesses.

To still take advantage of efficient operations on balanced trees, the display is relaxed to be focused on a branch of some balanced subtree[3]. To keep track of this subtree there are three additional fields: `focusStart` that represents start index of the current focused subtree, `focusEnd` that represents the end index of the subtree and `focusDepth` that sets height of the focused subtree[4]. The operations that can take advantage of the the efficient display operations will check if the index is in the subtree index range and invoke the efficient operation. If not, it will invoke the relaxed version of the operation, that starts from the root of the tree.

For example, the code for `getElement` would become:

```
def getElement(index: Int): A = {
  if (focusStart <= index && index < focusEnd)
    getElementFromDisplay(index - focusStart)
  else if (0 <= index && index < endIndex)
    getElementFromRoot(index)
  else
    throw new IndexOutOfBoundsException(index)
}
```

This `getElement` on the unbalanced subtrees of figure 3.4 would use the `getElementFromDisplay` to fetch elements in `display0` directly from it and fetch elements from nodes (1.1) and (1.2) from `display1`. The rest is fetched from the root using `getElementFromRoot`.

[3] A fully balanced tree will be itself the balanced subtree, as such it will always use the more performant operations.

[4] As an optimisation, the `focus` field is split into the part corresponding to the subtree and the part that represents the indices of the displays that are unbalanced.

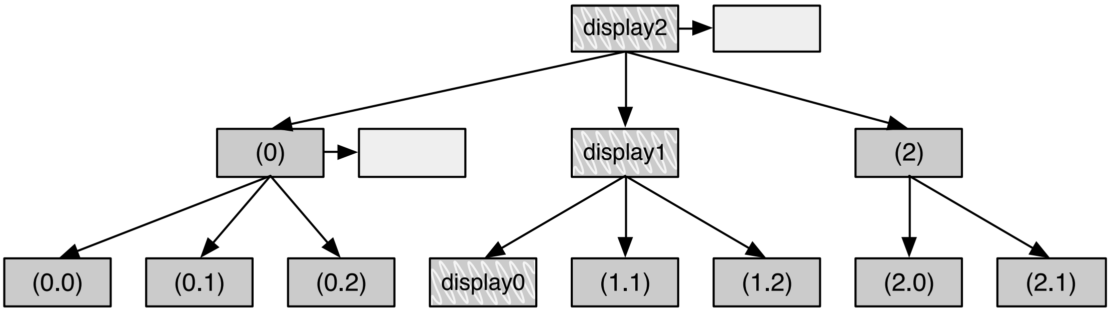


FIGURE 3.4: Relaxed Radix Balanced Tree with a focus on a balanced subtree rooted of `display1`. Light grey boxes represent unbalanced nodes sizes.

3.3 Builder

Relaxing the Builder

3.4 Iterator

Relaxing the Iterator I

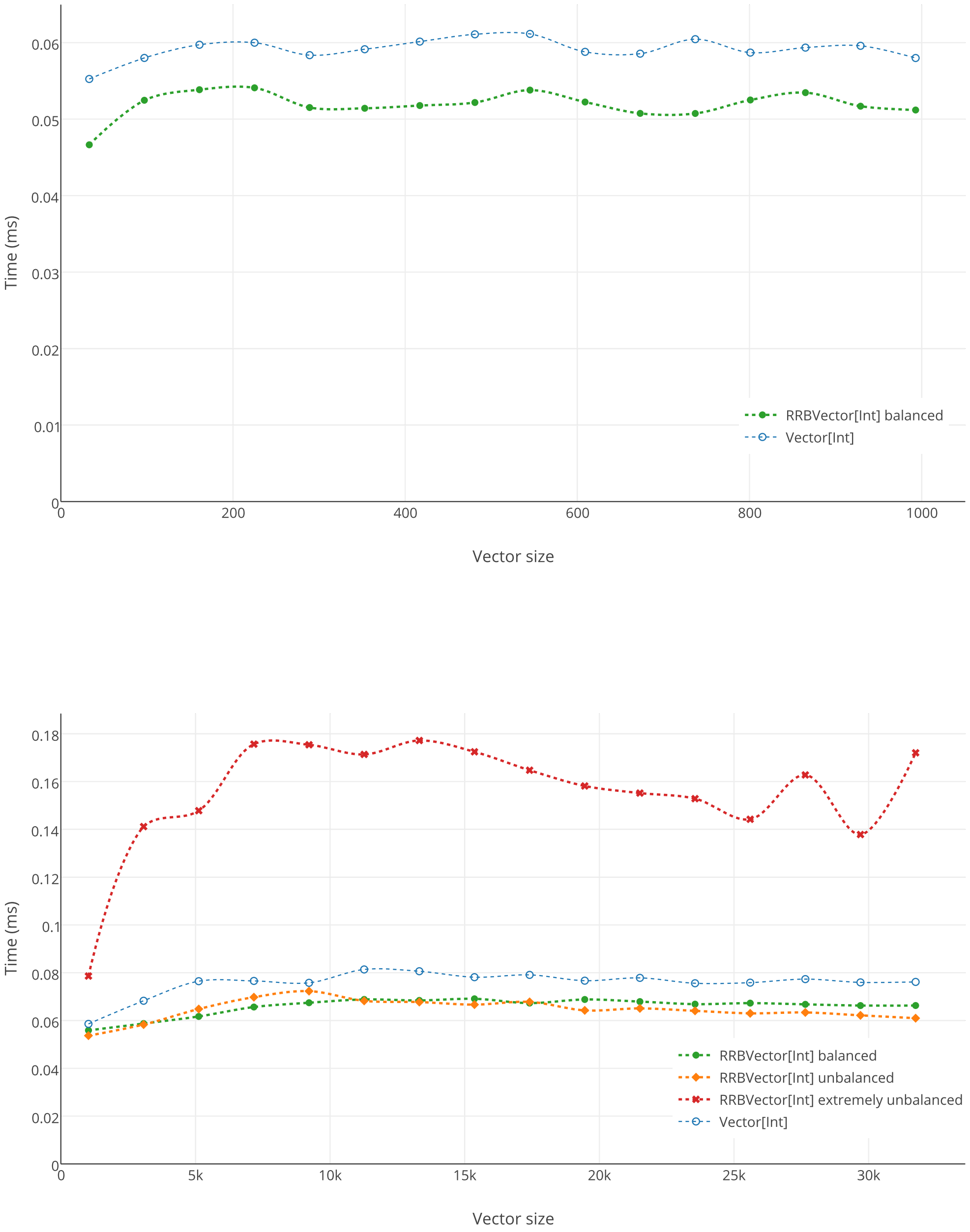


FIGURE 4.1: Time to execute 10k apply operations on sequential indices.

Chapter 4

Performance

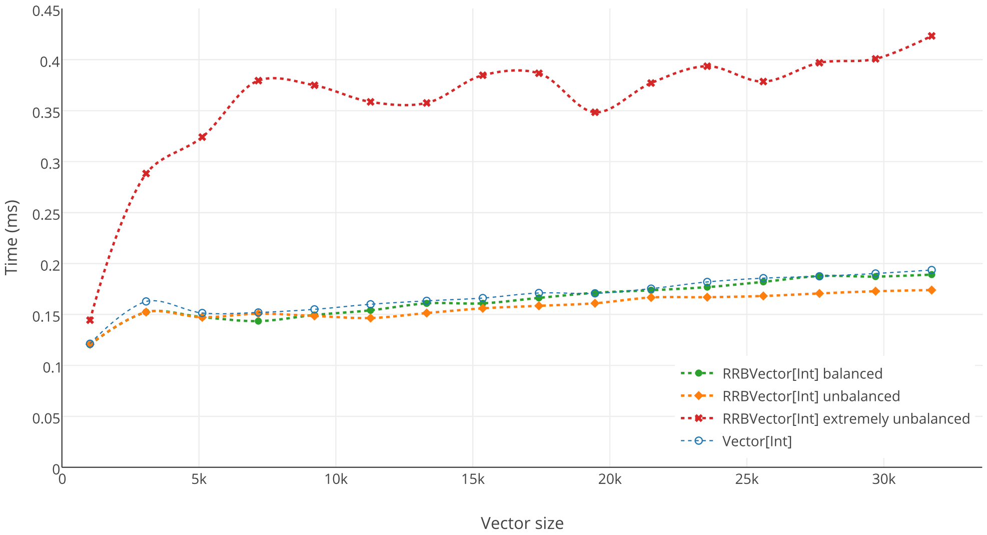


Figure 4.2: Time to execute 10k apply operations on random indices.

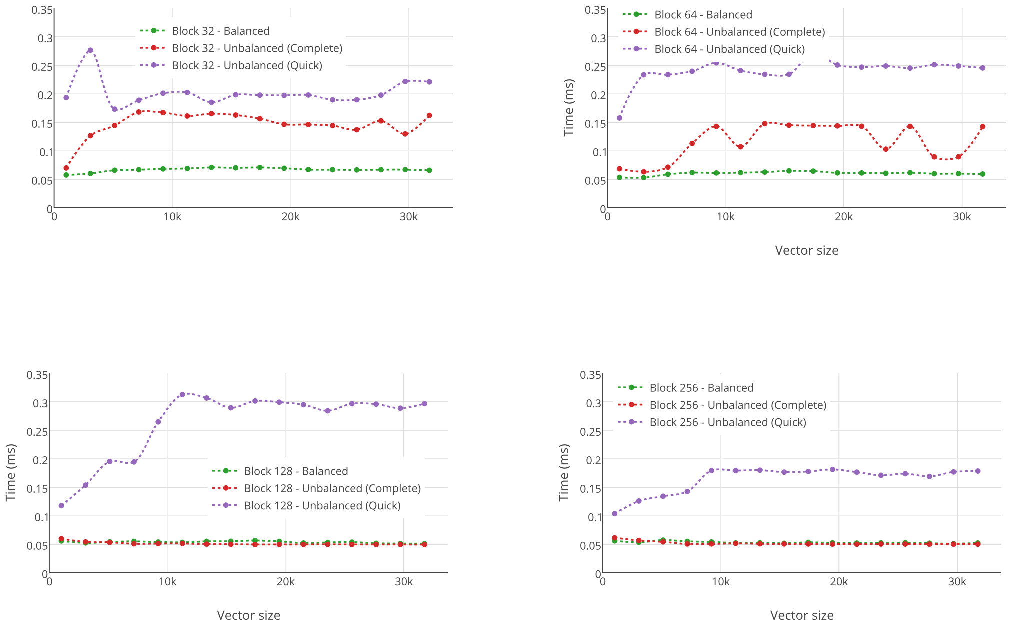


Figure 4.3: Time to execute 10k apply operations on sequential indices. Comparing performances for different block sizes and different implementation of the concatenation inner branch rebalancing (Complete/Quick).

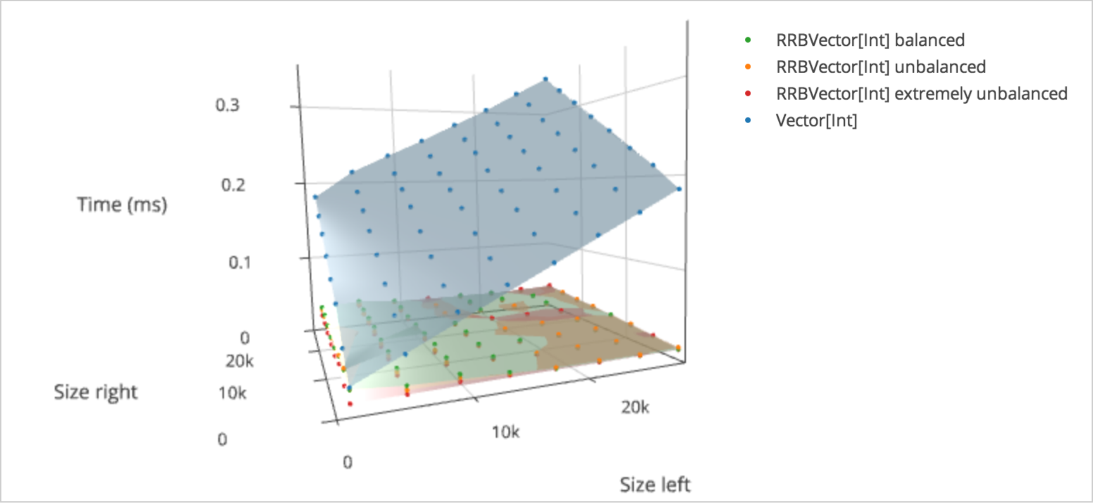


FIGURE 4.4: Execution time for a concatenation operation on two vectors. In theory (and in practice) Vector concatenation is $O(left + right)$ and the rrbVector concatenation operation is $O(log_{32}(left + right))$.

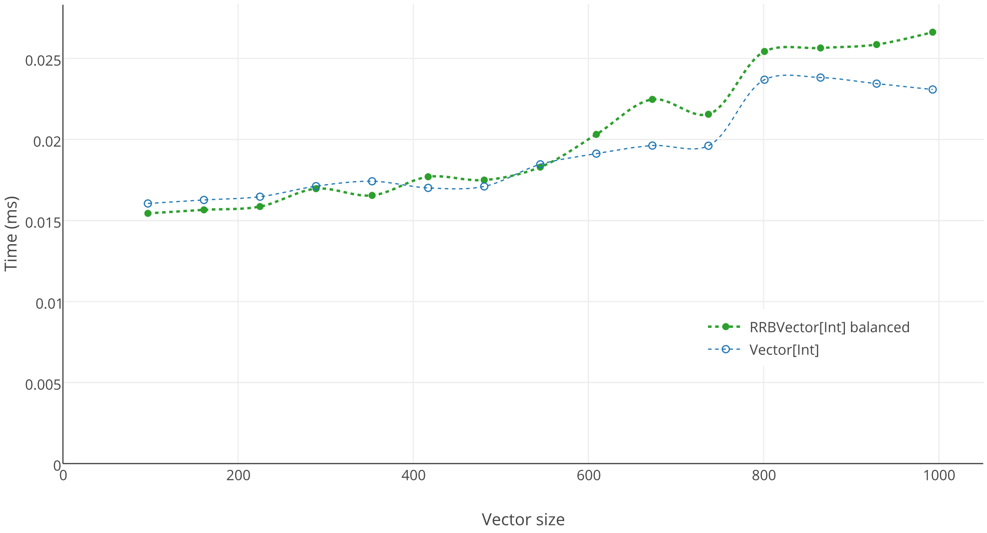


FIGURE 4.5: Time to execute 256 append operations. This shows the amortized cost of the append operation.

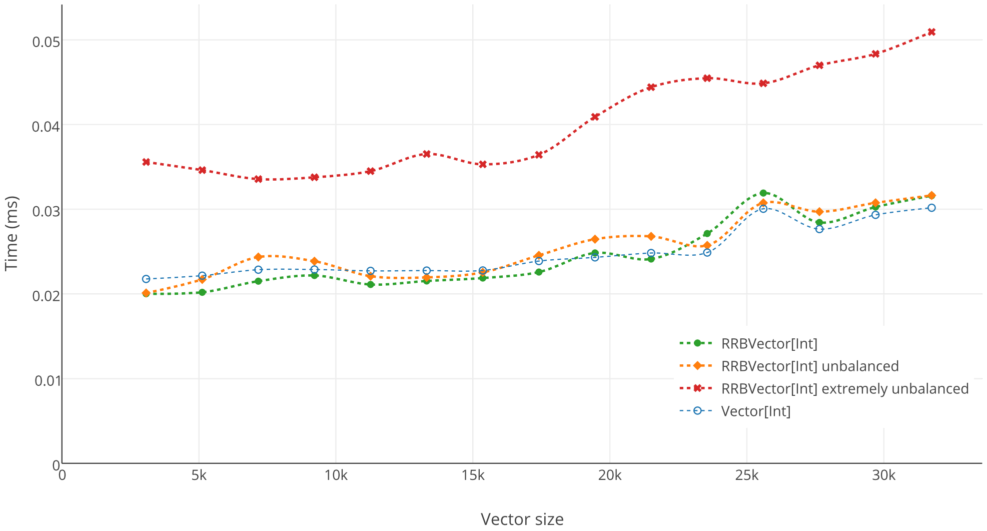


FIGURE 4.6: Time to execute 256 append operations. This shows the amortized cost of the append operation.

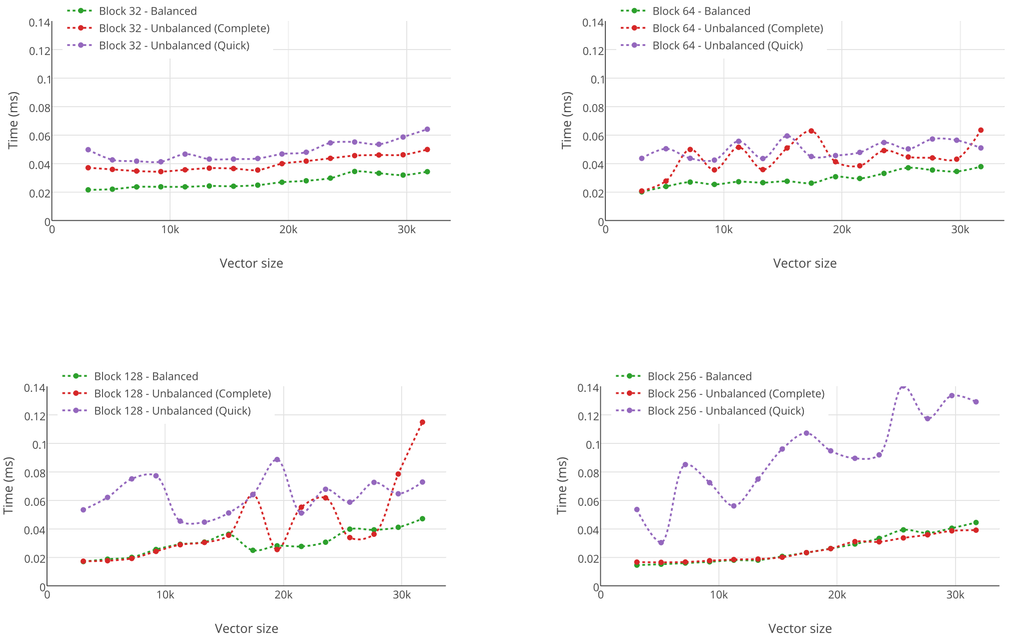
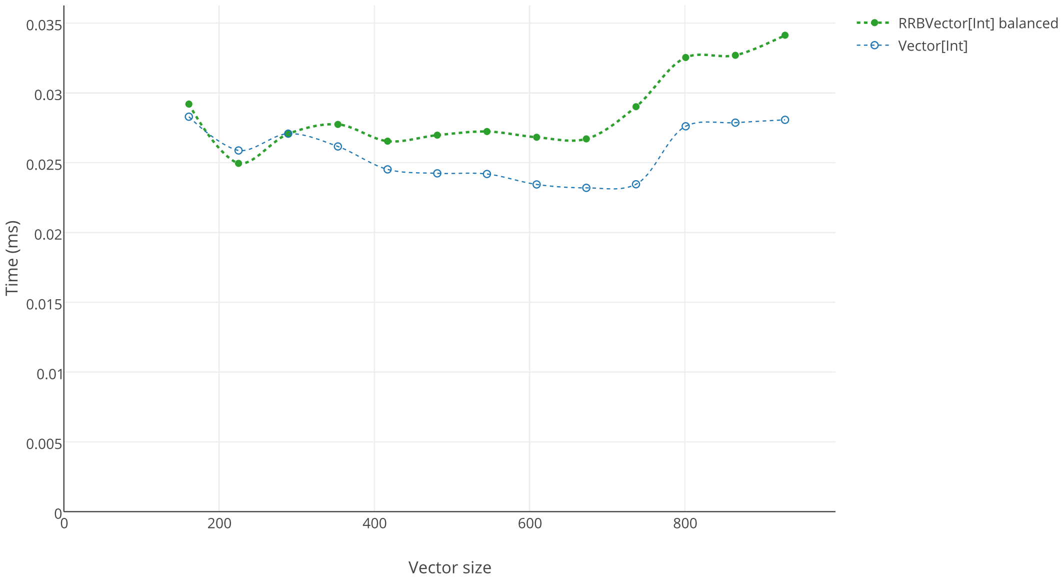


FIGURE 4.7: Time to execute 256 append operations. This shows the amortized cost of the append operation. Comparing performances for different block sizes and different implementation of the concatenation inner branch rebalancing (Complete/Quick).



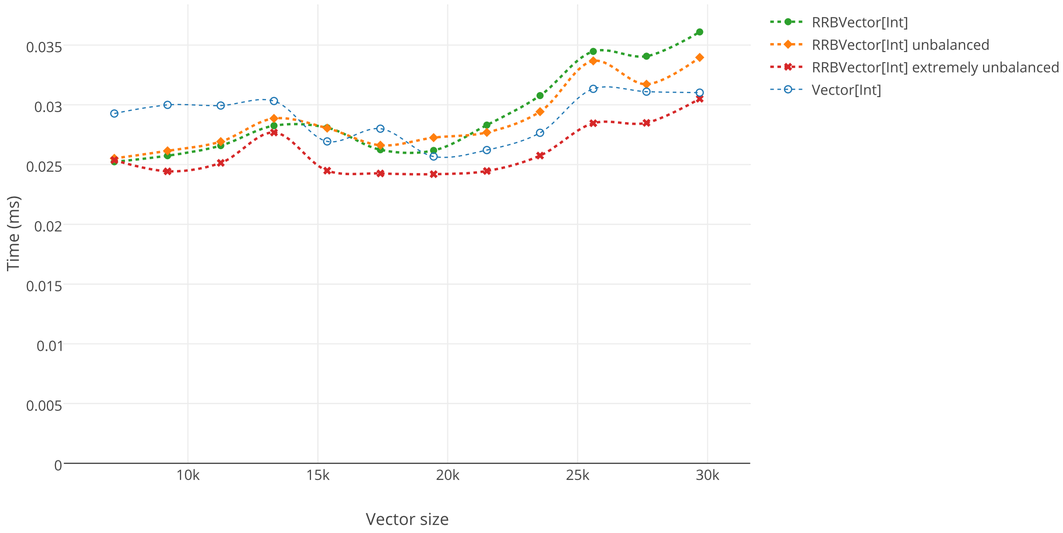


FIGURE 4.8: Time to execute 256 prepend operations. This shows the amortized cost of the prepend operation.

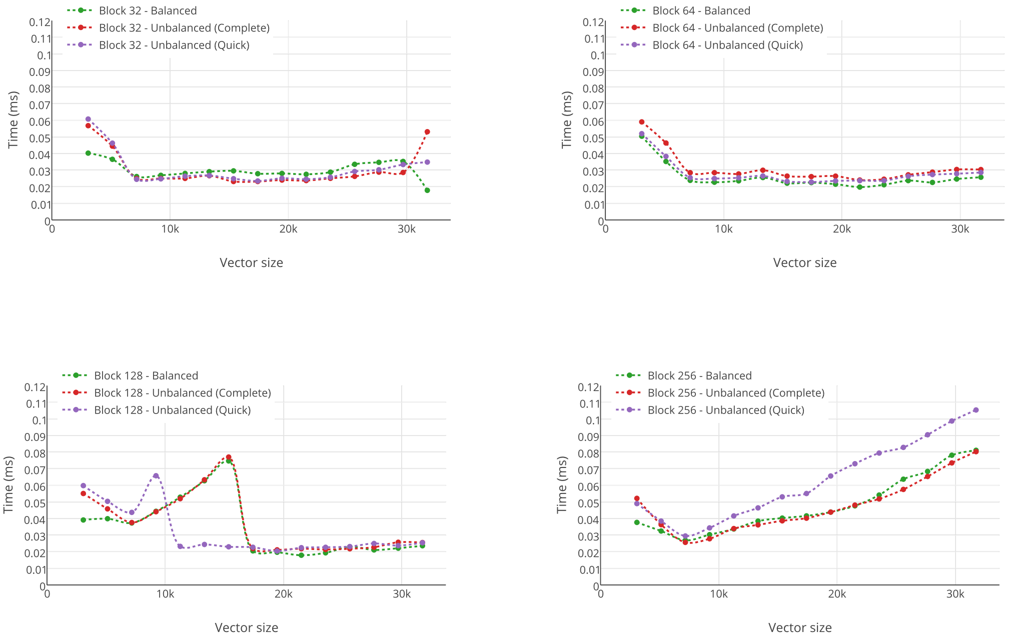


FIGURE 4.9: Time to execute 256 prepend operations. This shows the amortized cost of the append operation. Comparing performances for different block sizes and different implementation of the concatenation inner branch rebalancing (Complete/Quick).

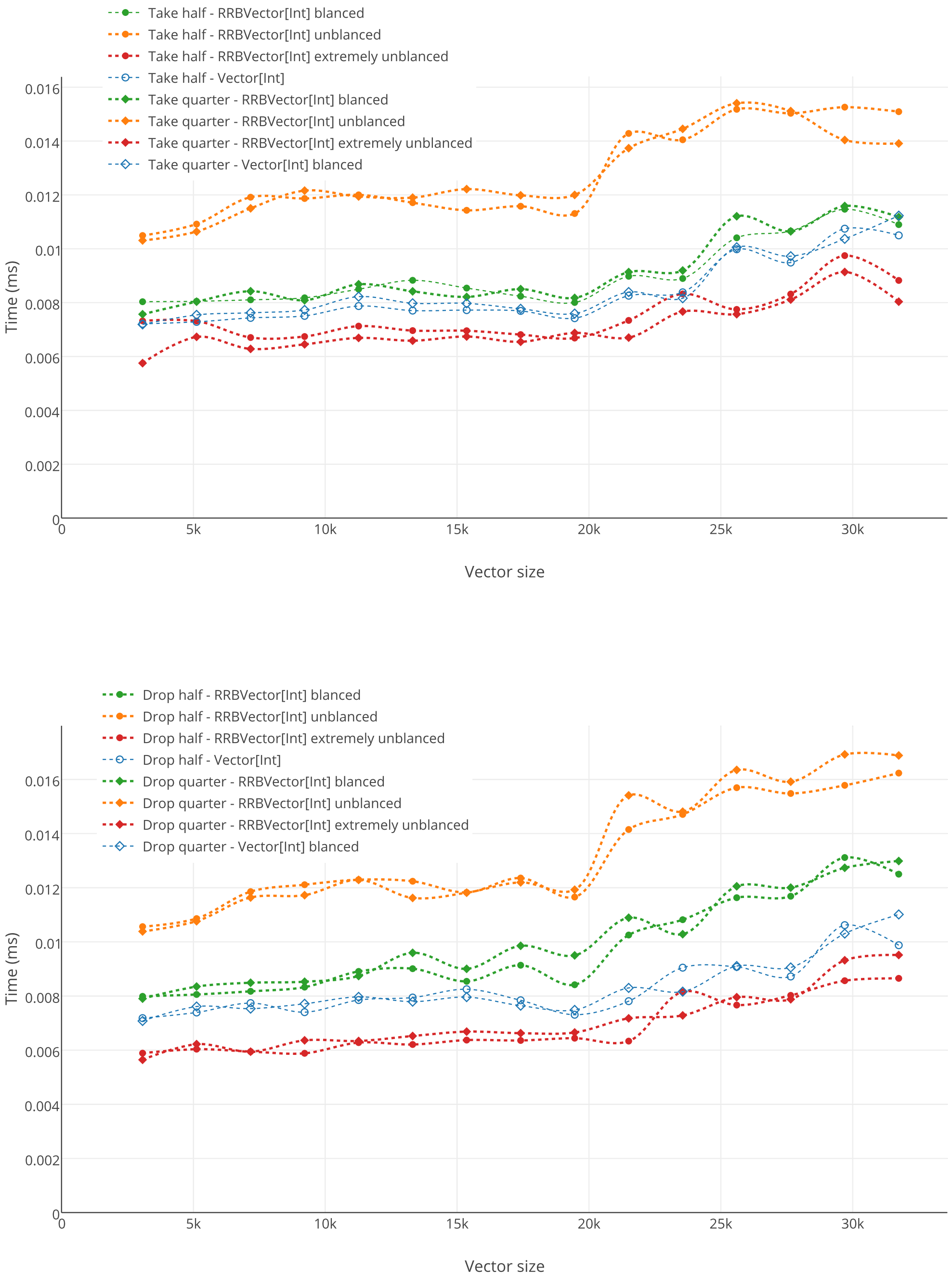
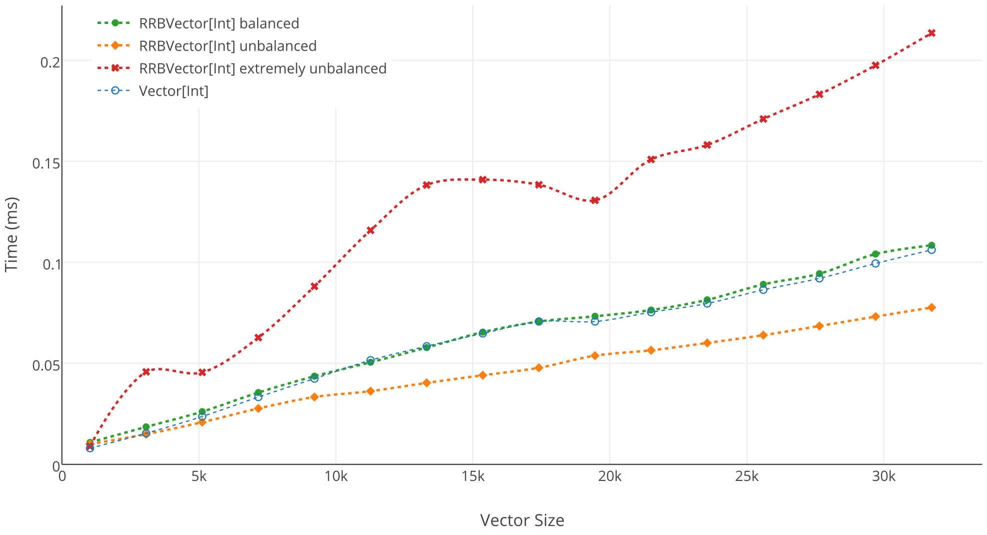


FIGURE 4.10: Execution time of take and drop.



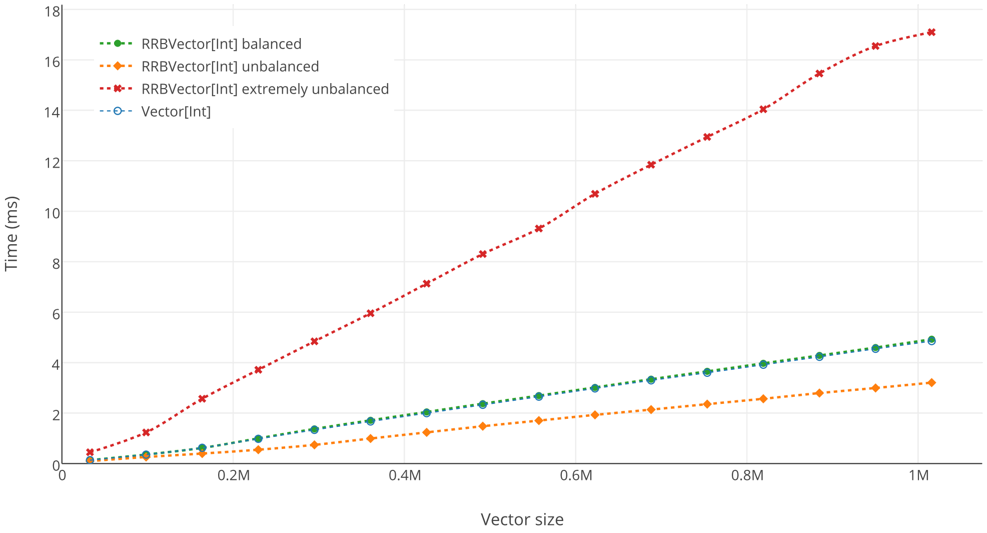


FIGURE 4.11: Excecution time to iterate through all the elements of the vector.

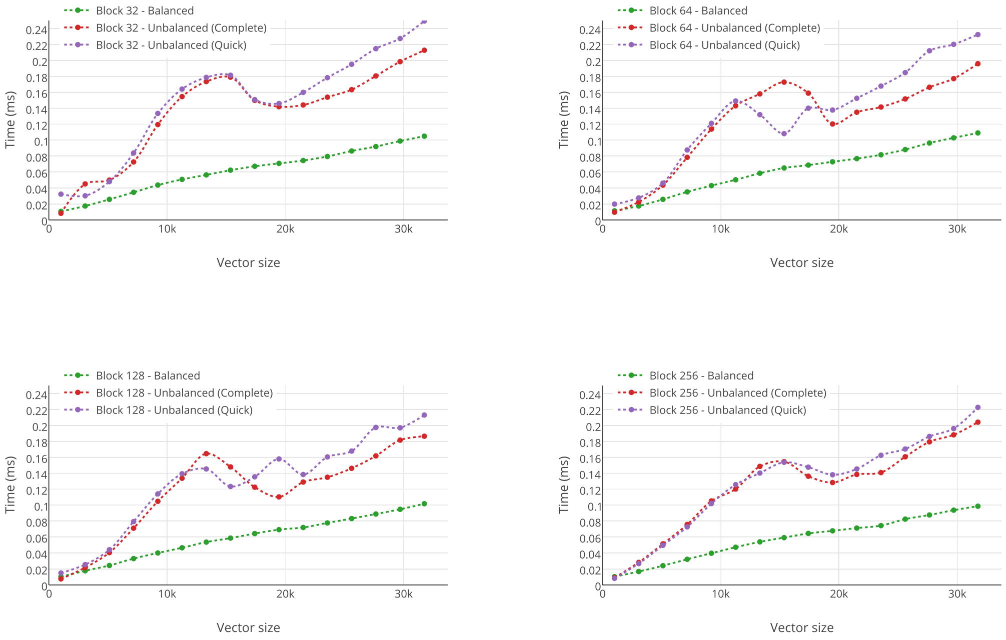
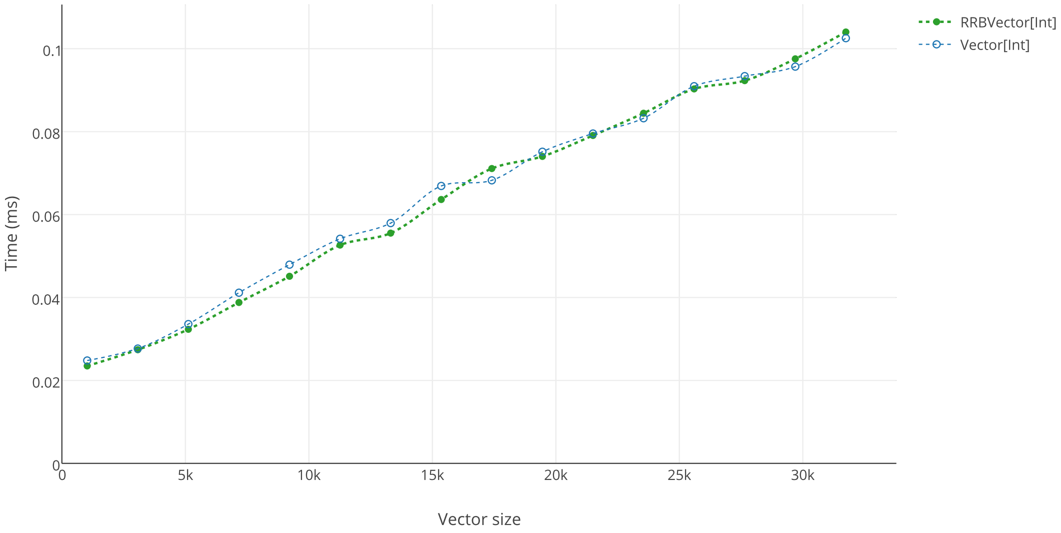


Figure 4.12: Excecution time to iterate through all the elements of the vector. Comparing performances for different block sizes and different implementation of the concatenation inner branch rebalancing (Complete/Quick).



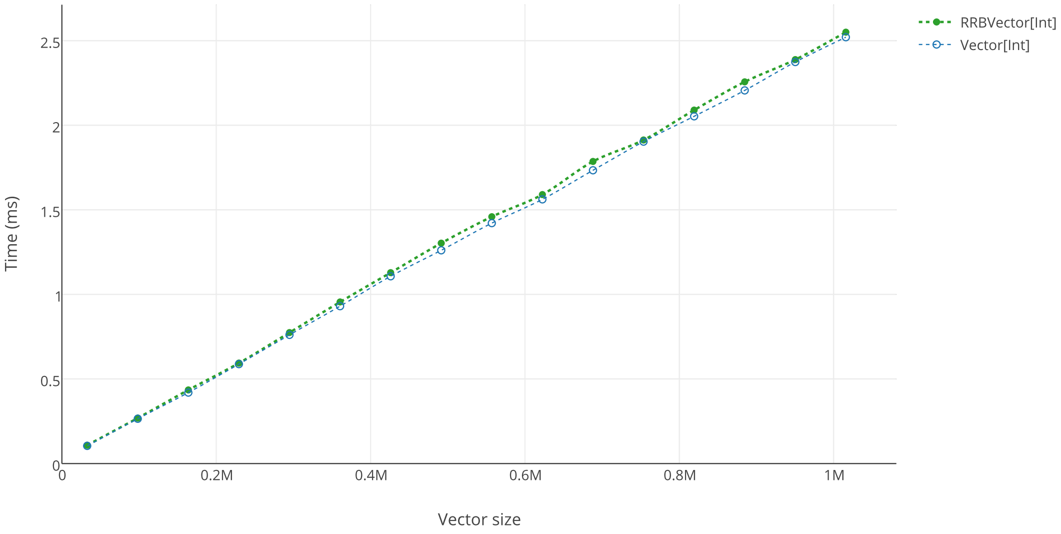


Figure 4.13: Execution time to build a vector of a given size.

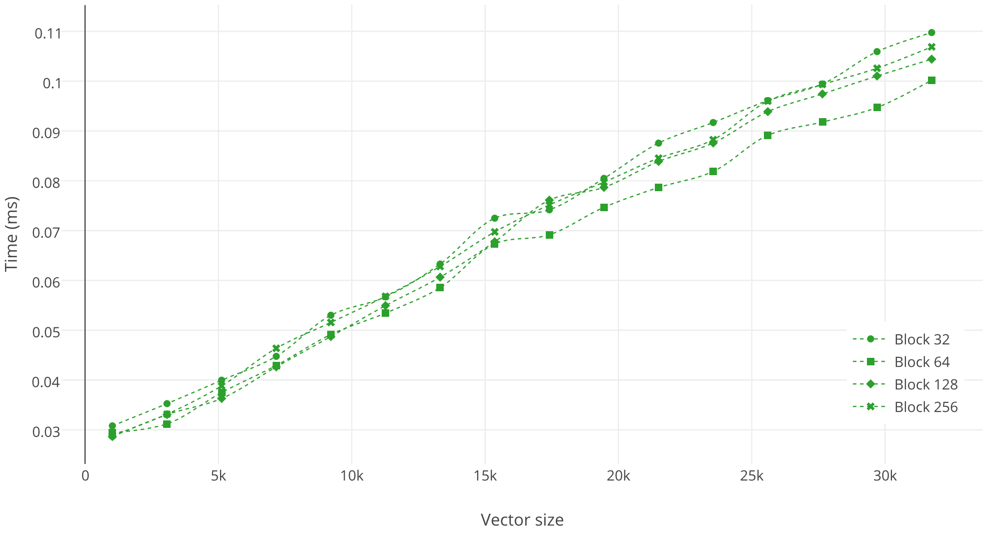


FIGURE 4.14: Execution time to build a vector of a given size. Comparing performances for different block sizes.

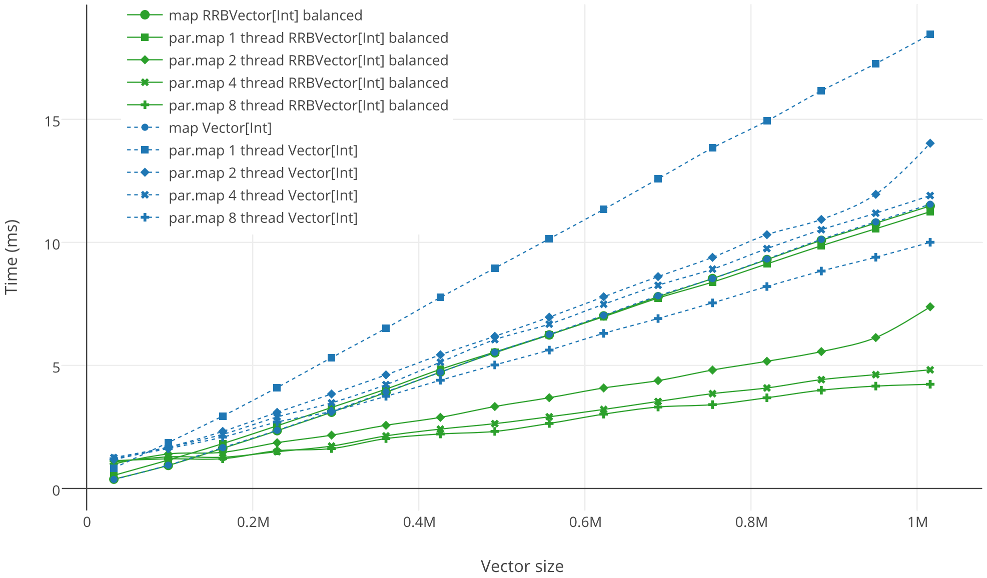


FIGURE 4.15: Benchmark on map and parallel map using the function (x=>x) to show the difference time used in the framework. This time represents the time spent in the splitters and combiners of the parallel collection (iterator and builder for the sequential version).

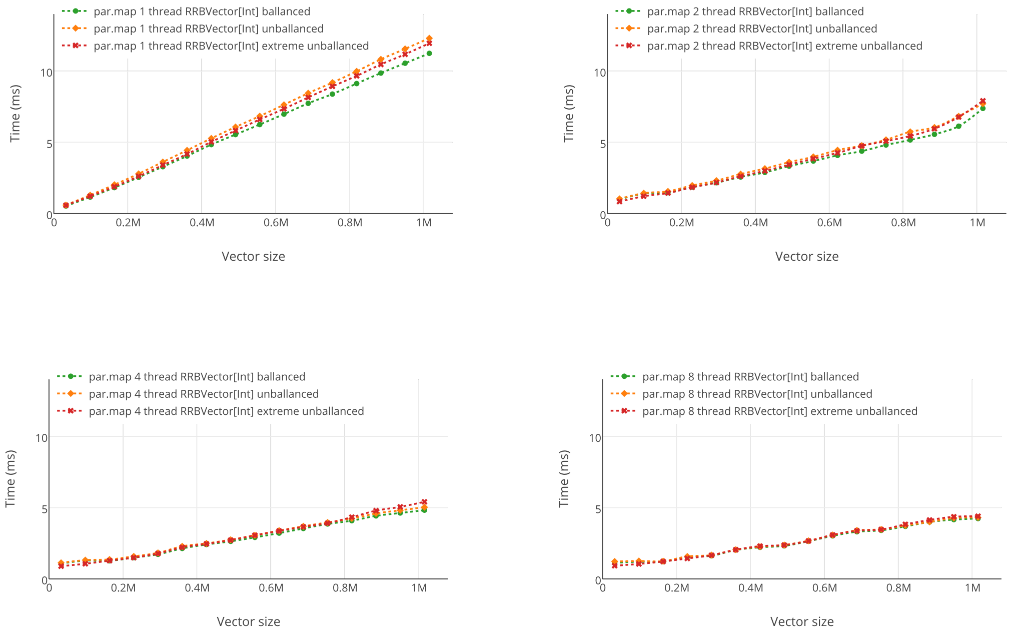
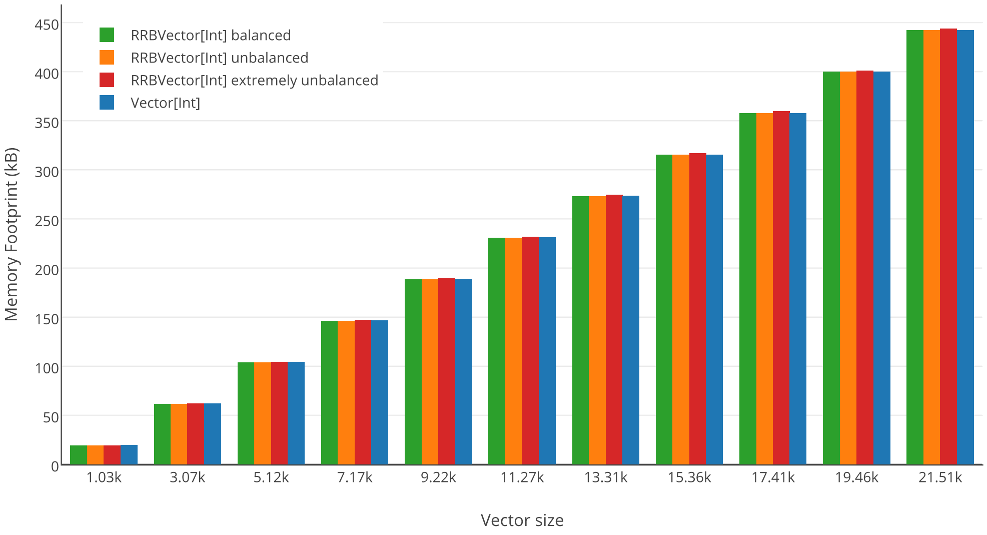


FIGURE 4.16: Benchmark on map and parallel map using the function (x=>x) to show the difference time used in the framework. This time represents the time spent in the splitters and combiners of the parallel collection.



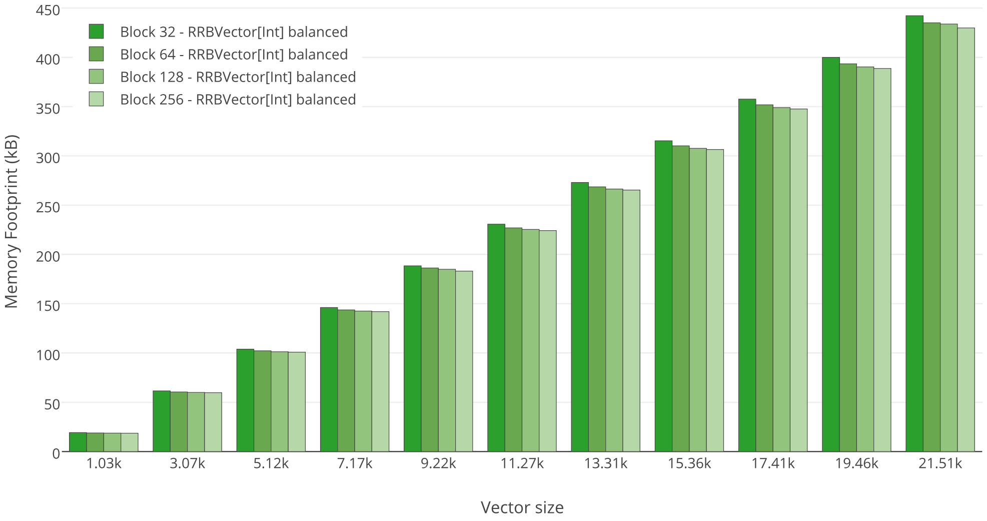


Figure 4.17: Memory Footprint

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, .

[2] GitHub - Scala 2.11 - Vector.scala. https://github.com/scala/scala/blob/394da59828b830f639d2418960052655d9dd040a/src/library/scala/collection/immutable/Vector.scala, .