ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Master Thesis

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


*Author:*
Nicolas Stucki

*Supervisor:*
Martin Odersky


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

*in the*

LAMP
Computer Science

January 2015

# *Abstract*

*The immutable* `Vector` *collection in the Scala library offers nearly constant-time random access reads thanks to its underlying wide tree data structure. Furthermore, it provides amortized constant time sequential* `read`*,* `update`*,* `append` *and* `prepend` *operations by efficiently sharing parts of the tree between different immutable vectors. However, the performance of parallel operations is hindered by the overhead of re-combining the results obtained from parallel workers.*

*Recent research has shown that parallel performance can be improved by relaxing the wide tree invariants and thus improving the performance of the combine operation. Although tree sharing is still used, sequential* `read`*,* `update`*,* `append` *and* `prepend` *are no longer amortized constant time. This prevents the new approach from making its way into the Scala Collections library.*

*The main insight of this thesis is that relaxed-invariant vector trees can be seen as a composition of strict and irregular parts. Therefore earlier optimizations can still be applied to strict subtrees, while irregular subtrees bring their own new optimization opportunities. This allows our implementation to provide both amortized constant-time sequential operations and efficient parallel execution at the same time.*

*Our implementation, which is fully compatible with Scala Collections, matches the sequential performance of standard vectors in most cases. At the same time benchmarks show parallel operations execute up to 2.3X faster on 4 threads on a 4 core machine thanks to the relaxed invariants allowing fast re-combination of results from different parallel executions.*

I

# Chapter 1

# Introduction

## 1.1 Motivation

With the increase in multicore machines, programs tend to use more parallelism and concurrency. In these contexts, managing mutable data structures introduces more complexity. Thread safety can become a burden on the programmer and the machine. A simpler approach is using immutable data structures, where elements can be accessed directly without any risk of corruption due to race conditions.

Scala has a rich collection of immutable data structures as well as parallel versions that divide their work transparently on thread pools. Where the vector is used as the default general purpose parallel sequence. The immutable vector is based on balanced wide trees, called RB-Trees, to allow effective constant time in practice on its key operations: get element at index, update element at index, append, prepend and splitting the vector. But, one of the operations required for parallelism, the concatenation operation is suboptimal.

To improve parallel performance, P. Bagwell and T. Rompf proposed a relaxed structure for the vector [1]. The Relaxed Radix Balanced Tree allows the tree to be partially filled, in turn allowing amortized constant-time concatenation. The drawback is in the loss of some optimizations opportunities due to the relaxation, which leads to the loss of amortized constant-time sequential operations due to the relaxation.

## 1.2 Objective

The main objective of this project was to implement[1] version of the Scala immutable Vector using RRB-Trees that could potentially replace the old one. As such the new implementation is functionally equivalent to the old one.

To provide the same performance and amortized constant-time operations of RB-Tree, we base the implementation on a composition of generic RRB-Tree operations with more performant RB-Tree operations. Using the most efficient operations on any tree or subtree where the stricter structure invariant holds. When possible, generalizations of optimizations are devised in the RRB operations and in other cases new optimizations opportunities are exploited. To enhance these benefits, the tree transformations favour the creation of RB-Trees or subtrees.

## 1.3 Results

To show and analyze the trade-offs between the implementations extensive benchmarks where realized on all core operations of the vectors. These compare the current vector with an array of different RRB-Trees including: different tree branching sizes, different implementation of concatenation algorithm and differently unbalanced RRB-Trees.

The implementation successfully integrated the the RRB-Trees with the radix based optimizations. As such the new Vector implementation has constant time for all operation on which it had before and on the additional concatenation and insertion operations. Form this it was possible to create a parallel vector that takes full advantage of the fork-join pool parallelization. Benchmarks showed a 2.3X improvement on the data split and combination on a pool with 4 threads.

The benchmarks showed that for almost all pre-existing constant time operations the performance achieved was almost identical. The few cases that do not have the same performance are still well bounded by constant time. It was shown that even with an apparently less compact tree structure the change in memory footprint negligible.

## 1.4 Document Structure

In chapter 2 we discuss the data structures and operations: Section 2.1 describes the current version of the vector structure and operations, section 2.2 discusses related data

[1] Implementation is located a https://github.com/nicolasstucki/scala-rrb-vector [2]

structure that use the same trees in other ways (like the iterator of a vector), section 2.3 describes the RRB-Tree and how to relax the operations based on the non relaxed versions.

In chapter 3 we describe the optimization done on the current version of the vector and how they are affected by the relaxation. In section 3.1 we describe where most of the processing time is spent and how it optimizations use this information. In section 3.2 we describe the optimizations that are used to reduce the algorithmic complexities of some operations. In section 3.3 we describe the implementation of the related structures and optimizations on them.

In chapter 4 we discuss the performance of the implementation in practice. In section 4.1 we describe characteristics of the JVM and how to take advantage of them. In sections 4.2 and 4.3 we describe the benchmarks used to measure performance. In section 4.4 we show the results of the benchmarks and analyze it.

In chapter 5 we present the testing methodology and in section 6 we review the related work. We conclude in in section 7.

I

# Chapter 2

# Immutable Vector Structure and Operations



An immutable vector is an representation for an indexed sequence[1] of elements with efficient random access and update[2]. It also provides operations that allow efficient additions and removal of elements.

## 2.1 Radix Balanced Vectors

The current immutable `Vector` implementation[3] of the Scala Collections is a wrapper around an immutable tree. The vector implements its methods based on a set of core efficient operations on the trees.

### 2.1.1 Tree structure

The RB-Tree (Relaxed Balanced Tree) structure is a shallow and densely-filled tree where elements are located only in the leafs and on the left side. The nodes in the tree have a fixed size, whether they're internal nodes, linking to subtrees, or leafs, containing elements. Benchmarks have shown the optimal node size is 32, but as we will later see[3], the node size can be any power of 2, allowing efficient radix-based implementations (see 3.1.2). Figure 2.1 shows this structure for $m$ children on each node.

The RB-Vector also keeps the height of the tree in the `depth` field. This height will always be upper bounded by $log_{32}(size - 1) + 1$ for nodes of 32 branches. Therefore

[1] Ordered sequence of elements where each one is identified by its index or position.

[2] Update in the scene of immutable data structures refers to the creation of a structure with an element changed.

[3] Nodes of size 64 could also be consider, with some trade-offs.

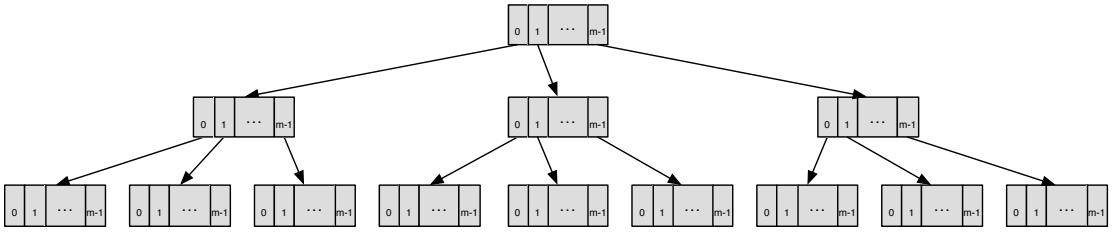


FIGURE 2.1: Radix Balanced Tree Structure

the tree is quite shallow and the complexity to traverse it is considered as effectively constant when taking into account that the size will never be larger than the maximum index representable with `Int`, which corresponds to a maximum depth of 7 levels[4].

Yet, the tree may have numbers of elements that are not powers of 32. To allow this, the RB-tree fills in the elements in the leafs from the left. The branches on the right that do not contain any element in any of their leafs are not allocated (left as empty references or truncated nodes). For example the tree that would hold 1056 elements in figure 2.2 would have empty subtrees on the right of the nodes on the rightmost branch. To prevent out-of-bounds access, the vector has an `endIndex` field that point to that last index. With this it is possible to have efficient implementations of `take` and `append` (see 2.1.2), as they make changes on this index rather that the structure of the tree.

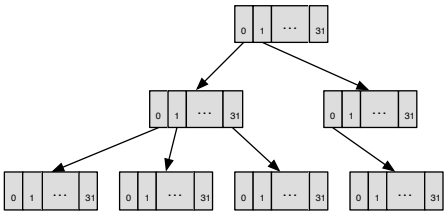


FIGURE 2.2: Radix Balanced Tree Structure with nodes of size 32 filled with 1056 elements.

Similarly, to have efficient implementation of `drop` and `prepend` operations (see 2.1.2), the vector defines a `startIndex` field that points to the first element in the tree that is actually used in the vector. In this case, although the leafs are empty on the left of the first element, the nodes are still allocated with the full size. As such, the tree structure is logically still full on the left but elements are never accessed if below the `startIndex`. An element at some `index` in the vector will be at `startIndex + index` in the tree. This allows the balanced radix structure to be maintained regardless of the missing elements on the left. As such, the implementations for `drop` and `prepend` will take advantage of the index to avoid restructuring the whole tree.

[4]Maximum depth of $log_{32}(2^{31}) + 1$, which is 6.2.

### 2.1.2 Operations

The immutable vector in the current Scala Collections is an indexed sequence, as such it offers a large range of operations. Most of these 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 (see 2.1.1). 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[5] and without optimization. These reflect the base implementation that is used before adding optimizations, which, in many cases lower the cost of the operations (see Chapter 3). This makes the operations in this section act as worst-case scenarios, when none of the optimizations apply and thus the basic operations have to be performed.

For simplicity of code in this section we define:

```
type Node = Array[AnyRef]
val Node = Array
```

#### 2.1.2.1 Apply

The `apply` operation returns the element located at the given 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: Node, depth: Int): A = {
    val indexInNode = // compute index
    if(depth == 1) node(indexInNode)
    else getElem(node(indexInNode), depth-1)
  }
  getElem(vectorRoot, vectorDepth)
}
```

The base implementation of the `apply` operation requires a simple traversal from the root to the leaf containing the element, 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 $O(log_{32}(n))$.

[5] Ignoring some pieces of code like casts, return types, type covariance, among others.

#### 2.1.2.2 Updated

This is the primary operation for transforming the vector. However, since the vector is an immutable data structure, instead of modifying the element in place, this operation returns a new immutable vector and does not modify the existing one. This is the reason it is called `updated` instead of the more common `update`.

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: Node, depth: Int) = {
    val indexInNode = // compute index
    val newNode = copy(node)
    if(depth == 1) {
      newNode(indexInNode) = elem
    } else {
      newNode(indexInNode) = updatedNode(node(indexInNode), depth-1)
    }
    newNode
  }
  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 branch. For example, if some leaf has all it's elements updated from left to right, the branch will be copied as many times as there are updates. We will later explain how this can be optimized by allowing transient states that avoid re-creating the path to the root tree node with each update (see 3.2.2).

#### 2.1.2.3 Append

The `appended` operation (or `:+`) adds an element to the end of the vector. Its implementation has two cases, depending on the current state of the RB-Tree: If the last leaf is not full the element is inserted at the end of it and all nodes of the last branch are copied. If the leaf is full we must find the lowest node in the last branch where there is sill room left for a new branch. Then a new branch is appended to it, down to the new leaf with the element being appended.

In both cases the new `Vector` object will have the end index increased by one. When the root is full, the depth of the vector will also increase by one.

```
def :+(elem: A): Vector[A] = {
  def appended(node: Node, depth: Int): Node = {
    if (depth == 1)
      copyLeafAndAppend(node, elem)
    else if (!isTreeFull(node.last, depth-1))
      copyAndUpdateLast(node, appended(node.last, depth-1))
    else
      copyBranchAndAppend(node, newBranch(depth-1))
  }
  def createNewBranch(depth: Int): Node = {
    if (depth == 1) Node(elem)
    else Node(newBranch(depth-1))
  }
  if(!isTreeFull(root, depth)))
    new Vector(appended(root, depth), depth, ...)
  else
    new Vector(Node(root, newBranch(depth)), depth+1, ...)
}
```

In the code above, the `isTreeFull` operation computes the answer using efficient bitwise operations on the end index of the vector.

Due to the update of all nodes in the last branch, the complexity of this operation is $O(log_{32}(n))$. Like the updated operation, append can be optimized by keeping transient states of the immutable vector (see Section 3.2.2).

#### 2.1.2.4 Prepend

The key to be able to prepend elements to the tree is the `startIndex` that the vector keeps in its fields. Without this it would be impossible to prepend an element to the vector without a full copy of the vector to rebalance the RB-Tree. The operation can be split into to cases depending on the value of the start index.

The first case is to prepend an element on a vector that start at index 0. If the root is full, create a new root on top with the old root as branch at index 1. If there is still space in the root, shift branches in the root by one to the right. In both subcases, create a new branch containing nodes of size 32, but with only the last branch assigned. Put the element in the last position of this newly created branch and set the start index of the vector to that index in the tree.

The second case is to prepend an element on a vector that has a non zero start index. To do so, we follow the branch of the start index minus one from the root of the tree. If we reaches the a leaf, prepend the element to that leaf. If we encounters an inexistent branch, create it by putting the element in its rightmost position and leaving the rest empty. In both subcases update the parent nodes up to the root.

The new `Vector` object will have an updated start index and end index, to account for the changes in the structure of the tree. It may also have to increase the depth of the vector if the start index was previously zero.

```
def +:(elem: A): Vector[A] = {
  def prepended(node: Node, depth: Int) = {
    val indexInNode = // compute index
    if (depth == 1)
      copyAndUpdate(node, indexInNode, elem)
    else
      copyAndUpdate(node, indexInNode,
                prepended(node(indexInNode), depth-1))
  }
  def newBranch(depth: Int): Node = {
    val newNode = Node.ofDim(32)
    newNode(31) = if (depth == 1) elem else newBranch(depth-1)
    newNode
  }
  if (startIndex==0) {
    new Vector(Node(newBranch(depth), root), depth+1, ...)
  } else {
    new Vector(prepended(root, depth), depth, ...)
  }
}
```

Since the algorithm traverses and creates new nodes from the root to a leaf, the complexity of the prepend operation is $O(log_{32}(n))$. Like the updated and appended operations, `prepended` can be optimized by keeping transient states of the immutable vector (see Section 3.2.2).

#### 2.1.2.5 Concatenation

Given that elements in an RB-Tree are densely packed, implementations for concatenation and insert will need to rebalance elements to make them again densely packed. In the case of concatenation there are three options to join the two vectors: append all elements of the RHS vector to the LHS vector, prepend all elements from the LHS vector to the RHS vector or simply reconstruct a new vector using a builder. The append and prepend options are used when on of the vectors is small enough in relation to the other one. When vectors become large, the best option is building a new vector, since this avoids creating one instance of the vector each time an element is added.

The computational complexity of the concatenation operation on RB-Trees for vectors lengths $n1$ and $n2$ is $O(n1 + n2)$ in general. In the special case where $n1$ or $n2$ is small enough the complexity becomes $O(min(n1, n2) * log_{32}(n1 + n2))$, the complexity of repeatedly appending or prepending an element.

#### 2.1.2.6 Splits

The core operations to remove elements in a vector are the `take` and `drop` methods. They are used to implement may other operations like `splitAt`, `tail`, `init`, ...

Take and drop have a similar implementation. The first step is traversing the tree the leaf where the cut will be done. Then the branch is copied and cleared on one side. The tree may become shallower during this operation, in which case some of the nodes on the top will not be copied and just dropped. Finally, the `startIndex` and `endIndex` are adjusted according to the changes on the tree.

The computational complexity of any split operation is $O(log_{32}(n))$ due to the traversal and copy of nodes on the branch where the cut index is located. $O(log_{32}(n))$ for the traversal of the branch and then $O(log_{32}(m))$ for the creation of the new branch, where $m$ is the size of the new vector (with $0 \leq m \leq n$).

## 2.2 Related Structures

### 2.2.1 Vector Builder

A vector builder is a special wrapper for a RB-Tree that has an efficient `append` operation. It encapsulates mutable trees during the building of the vector and then freezes them on the creation of the result. Details of this structures are discussed in section 3.3.1.

### 2.2.2 Vector Iterator

A vector iterator is a structure that implements an efficient traversal of the vector. In the case of vectors, it is achieved by direct inorder traversal on trees. Details of this structures are discussed in section 3.3.2.

### 2.2.3 Parallel Vectors

The parallel vector (or `ParVector` [4]) is a wrapper around the normal `Vector` object that uses the parallel collections API instead of the normal collections API. With this, operations on collections get executed transparently on fork-join thread pools rather that on the main thread. In order to execute in parallel, the parallel vector needs to be split into chunks, that are processed in parallel and later combined. This is where the `Splitter` and `Combiner` objects come into play.

The use of a vector combiner produces additional overhead, since requires an concatenation of vector. To reduce this excessive overhead on concatenation the current implementation does it lazily by buffering vectors. Once all the vectors are computed a new one is build by traversing all the vectors in the buffer.

**Splitter (Iterator)** To divide the work into tasks for the 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 uses the common division into 2 parts of same size. The drop and take operations are used divide the vector for the two new splitters.

**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 keeps all the sub-combiners in an array buffer and only when the end result is needed they are combined by building a new vector with all elements in the vectors contained in the buffer. This alternative comes with a drawback on loss of parallelism of the vector building at the end of the combination as this is done in a single thread.

The current implementation of the immutable parallel vector uses the lazy approach because of its inefficient concatenation operation. One of the consequences of this is that the parallel operations will always be bounded by this sequential building of a vector, which is usually slower than the sequential version.

## 2.3 Relaxed Radix Balanced Vectors

The implementation for RRB vectors proposed consists in replacing the wrapped RB-Tree by an RRB-Tree and augmenting the operations accordingly while trying to maintain the performance of RB-Tree operations. The structure of the RRB-Trees doesn't ensure by itself that the tree height is bounded, this is something that the implementation of the operations must ensure.

### 2.3.1 Relaxed Tree structure

An RRB-Tree is a generalization of an RB-Tree where there is an additional kind of node. This new node is an unbalanced node that does not require the it's children to be full. Leaf nodes are never considered as unbalanced. For efficiency these nodes will have metadata on the sizes of it's children.

In an unbalanced node, the sizes are kept in an additional array structure and attached at the end of the array that contains the children. The sizes array has the same length as the number of children of the node as is shown in figure 2.3. This kind of nodes get truncated rather that keep empty branches as all this metadata can now be encoded into the sizes.

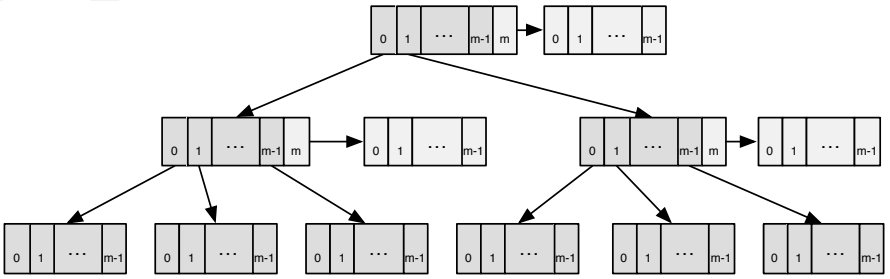


FIGURE 2.3: Radix Balanced Tree structure

In the RRB-Tree paper [1] the sizes are located in the front. They where moved to favour the performance on radix indexing on balanced node rather than on unbalanced nodes sizes. This is based on the assumption that most operation will be executed on balanced nodes.

To have an homogeneous representation for unbalanced nodes and balanced node, the balanced nodes arrays are extended with one extra empty position at the end. Therefore all the nodes have the same structure and the node is balanced if and only if the last position is empty[6]. The leafs do not need this additional data. Figure 2.4 shows an example of a RRB-Tree that has a combination of balanced an unbalanced nodes.

[6]Empty position is just a `null` reference.

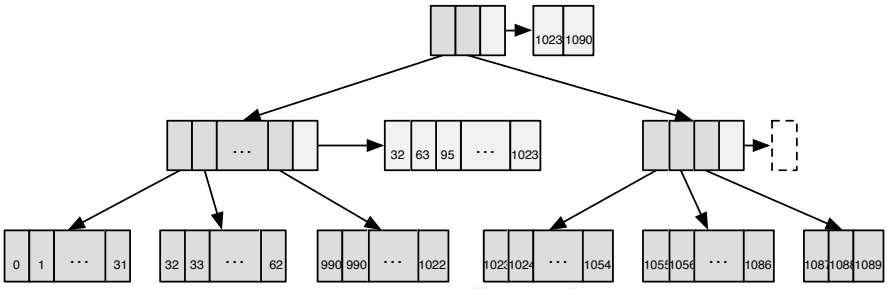


FIGURE 2.4: Concrete example of an RRB-Tree that contains 1090 elements.

As it is possible to represent trees that are not filled on the left, the `startIndex` is removed from the `Vector` object and assumed to be always zero. The `endIndex` is also not strictly necessary but is kept for performance of index bound checks.

### 2.3.2 Relaxing the Operations

For most operations the implementation is relaxed using the same technique. It consist in using a generalized version of the code for RB-Trees that take into account the unbalanced nodes that do not support radix operations. Mainly this consists in changing the way the `indicesInNode` are computed on unbalanced nodes and their `clone` operations. This adds an abstraction layer for some of the operations (see figure 2.5).

To take advantage of radix based code when possible the operations will still call the radix version if the node is balanced. This implies that from a balanced node down the radix based code is executed. Only the unbalanced nodes will have loss in performance due to generalization and therefore the code executed for a balanced RRB-Tree will tend to be the same as the one for RB-Trees.

As the RRB-Tree structure does not ensure the bound on the height unless the tree is completely balanced, the operation must ensure that tree respects the $log_{32}(n)$ height. Fortunately there are only three operations that can unbalance a tree: the `concatenated`, `prepend` and `drop` operations. The operations that also modify the structure of the tree like `append` and `take` do not add unbalanced nodes, but can affect the existing ones.

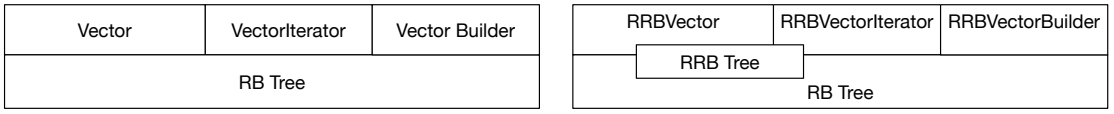


FIGURE 2.5: Abstractions layers for original and relaxed RB trees.

### 2.3.3 Core operations with minor changes

**Apply** The apply operation does not fundamentally change. The only difference is in the way the index of the next branch of a node is computed. If the node is unbalanced the sizes of the tree must be accessed (see section 3.1.2).

**Updated** For the `updated` the changes are simple because the structure of the RRB-Tree will not be affected. The computation of the `indexInNode` will change to take into account the possibility of unbalanced nodes while traversing down the tree. The `copy` operation will need to copy the sizes of the node if the node is unbalanced. Because the sizes are represented in an immutable array, this copy of sizes is in fact a reference to the same object.

**Append** To append an element on an RRB-Tree it is only necessary to change the `copyBranchAndAppend`, `copyAndUpdateLast` and `createNewBranch` helper functions. The `copyBranchAndAppend` will additionally copy the sizes and append to it a new size with value equal the previous last size plus one. The `copyAndUpdateLast` will also copy the sizes of the branch and increase the last size by one. The `createNewBranch` will have to allocate one empty position at the end of the new non leaf nodes of the branch. Note that any new branch created by append will be created as a balanced subtree.

**Take** This operation needs to take into account the RRB tree traversal scheme to go down to the index. The tree is cut in the same way the RB-Tree is cut with an additional step. When cutting an unbalanced node the sizes of that node are cut at the same index and the last size is adjusted to match the elements in the cut.

### 2.3.4 Concatenation

The concatenation algorithm used on RRB-Vectors is the one proposed in the RRB-Trees paper [1]. From a high level, the algorithm merges the rightmost branch of the vector on the LHS with the leftmost branch of the vector on the RHS. While merging the node, there is a rebalancing that is effectuated on each of them to ensure the logarithmic bound on the height of the vector. The RRB version of concatenation has a time complexity of $O(log_{32}(n))$, which is a clear improvement from the RB concatenation that was $O(n)$.

```
def concatenate(left: Vector[A], right: Vector[A]) = {
    val newTree = mergedTrees(left.root, right.root)
    val maxDepth = max(left.depth, right.depth)
    if (newTree.hasSingleBranch)
      new Vector(newTree.head, maxDepth)
    else
      new Vector(newTree, maxDepth+1)
}
def mergedTrees(left: Node, right: Node, depth: Int) {
  if (depth==1) {
    mergedLeafs(left, right)
  } else {
    val merged =
      if (depth==2) mergedLeafs(left.last, right.first)
      else mergedTrees(left.last, right.first, depth-1)
    mergeRebalance(left.init, merged, right.tail)
  }
}
def mergedLeafs(left: Node, right: Node} = {
  // create a balanced new tree of height 2
  // with all elements in the nodes
}
```

The concatenation operation starts at the bottom of the branches by merging the leafs into a balanced tree of height 2 using `mergedLeafs`. Then, for each level on top of it, the newly created merged subtree and the remaining branches on that level will be merged and rebalanced into a new subtree. This new subtree always adds a new level to the tree, even though it might be drop later on. New sizes of nodes are computed each time a node is created based on sizes of children nodes. The rebalancing algorithm has

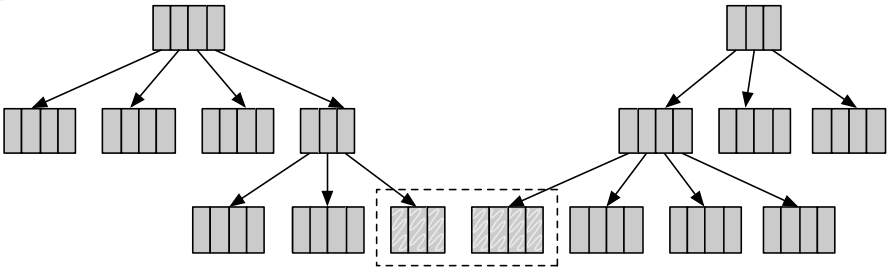

FIGURE 2.6: Concatenation example with nodes of size 4: Rebalancing level 0

two proposed variants. The first consists in completely rebalancing the nodes on the two top levels of the subtree. The second also rebalances the top two level of the subtree but it only rebalance the minimum amount of nodes that ensures the logarithmic bound. The first one leaves the tree better balanced, while the second is faster. More detail on the bounds and complexities can be bound on the RRB-Tree paper [1]. The following snippet of code shows a high level implementation of the first variant.

```
def mergeRebalance(left: Node, center: Node, right: Node) {
  val merged = left ++ centre ++ right // join all branches
  var newRoot = new ArrayBuilder
  var newSubtree = new ArrayBuilder
  var newNode = new ArrayBuilder
  def checkSubtree() = {
    if(newSubtree.length == 32) {
       newRoot += computeSizes(newSubtree.result())
       newSubtree.clear()
     }
  }
  for (subtree <- merged; node <-subree) {
    if(newNode.length == 32) {
      checkSubtree()
      newSubtree += computeSizes(newNode.result())
      newNode.clear()
    }
     newNode += node
  }
  checkSubtree()
  newSubtree += computeSizes(newNode.result())
  computeSizes(newRoot.result)
}
```

Figures 2.6, 2.7, 2.8 and 2.9 show a concrete step by step (level by level) example of the concatenation of two vectors. In the example, some of the subtrees where collapsed. This is not only to make it fit, but also to expose only the nodes that are referenced during the execution of the algorithm. Nodes with colours represent new nodes and changes, to help track them from figure to figure.

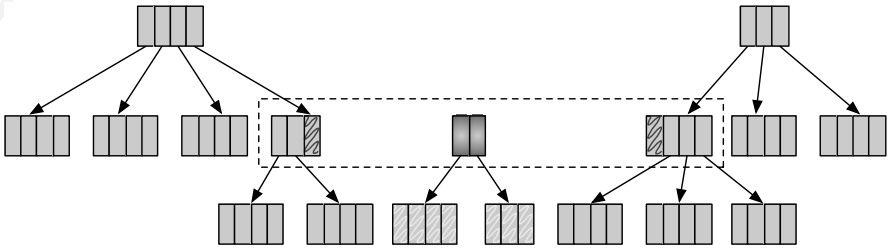

FIGURE 2.7: Concatenation example with nodes of size 4: Rebalancing level 1

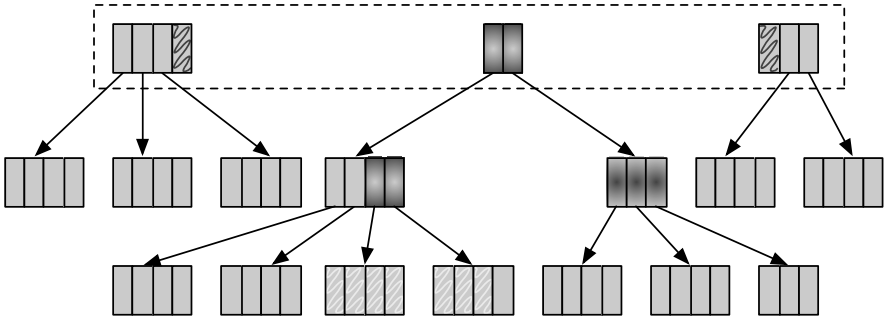

FIGURE 2.8: Concatenation example with nodes of size 4: Rebalancing level 2

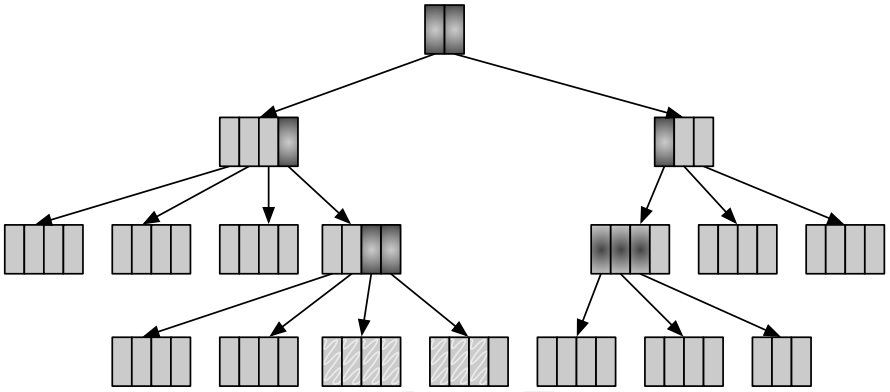

FIGURE 2.9: Concatenation example with nodes of size 4: Rebalancing level 3

The concatenation algorithm chosen as for the RRB-Vector is the one that is slower but that rebalances better the trees. The reason behind this decision is that with better balanced trees all other operations on the trees are more performant. In fact choosing the least performant option does not need to be seen as a reduction in performance because the improvement is in relation to the RB concatenation linear complexity. An interesting consequence of the this choice, is that all vectors of size at most 1024[7] that where created by concatenation will be completely balanced.

When using displays and transient states, concatenating a small vector the concatenation algorithm is less performant that strait forward append or prepends on the other vector. That case is identified by using bounds on their lengths. If one is big and the other is small, a `prependAll`/`appendAll` variant is used. Those two operations are optimized version of the `append`/`prepend` that do not create the intermediate `Vector` wrapper object and some of the leaf arrays. When both vector are small enough the rebalance is done directly with `mergedLeafs`.

**InsertAt** The operation `insertAt` is not currently defined on `Vector` because there is no way to implement it efficiently. But with RRB-Trees it is possible to implement this operation quite simply using `splitAt`, `append` and `concatenate`. This implementation has a time complexity of $log_{32}(n)$ that comes directly from the operations used.

```
def insertAt(elem: A, n: Int): Vector[A] = {
  val splitted = this.splitAt(n)
  (splitted._1 :+ elem) ++ splitted._2
}
```

[7] The maximum size of a two level RRB-Tree.

As this operation is new in the context of a vector and no real world use cases exist, this simple implementation is used[8].

### 2.3.5 Prepend and Drop

From a high level, the `prepend` and `drop` operations on RB-Trees and RRB-Trees are quite similar. The difference is in the way the nodes are copied, updated and cut. In the RRB operations, instead of using a `startIndex`, the nodes on the leftmost branch can become unbalanced and hence represent a first branch that is not fully filled.

Most of the time, both these operation will create new unbalanced nodes, but only on the left most branch. Calling several times combinations of these two will not contribute in generating even more unbalanced branches. As such they are only responsible for the creation of at most $log_{32}(n)$ unbalanced nodes on any vector.

**Prepend** Just like with the RB version, the first step is to traverse down the left most part of the tree. Traversing the tree becomes trivial because now the first branch is always on sub-index 0. As now there is a need for checking if there is still space in the leftmost branch the prepend returns a result if it could prepend it on that subtree.

When prepending an element on a subtree, if the element was added the parent nodes are updated with there corresponding sizes. If the element could not be prepended, then we check if the root of the current subtree has still space for another branch. If there is space a new branch is prepended on it, otherwise the prepend is delegated back to the parent node.

```
def +:(elem: A): Vector[A] = {
  def prepended(node: Node, depth: Int) = {
    if (depth == 1) {
      Some(copyAndPrepend(node, elem))
    } else {
      prepended(node(0), depth-1) match {
        case Some(newChild) =>
          Some(copyAndUpdate(node, 0, newChild)))
        case None if canPrependBranchOn(node) =>
          Some(copyAndPrepend(node, newBranch(depth-1)))
        case _ => None
    }
  }
  def newBranch(depth: Int): Node = {
    val newNode = Node.ofDim(2)
    newNode(0) = if (depth==1) elem else newBranch(depth-1)
    newNode
  }
```

[8] It would be possible to optimize for localized inserts on the same leaf using the same optimizations that are used for `updated`, `appended` and `prepended`

```
  prepended(root, depth) match {
    case Some(newRoot) => new Vector(newRoot, depth, ...)
    case None =>
      Vector(copyAndPrepend(root, newBranch(depth)),
             depth+1, ...)
  }
}
```

Sizes are updated by adding one to each size and in the case of `copyAndPrepend` additionally prepend a 1. Nodes on new branches do not need sizes as they are balanced, but they still need the empty slot.

This operation will generate unbalanced nodes, but will not unbalance the tree each time the `prepend` operation is called. In fact it will only generate a new unbalanced node when prepending a new branch on a balanced subtree. Then will start filling that new branch up to the point where it becomes balanced.

Due to the update of all nodes in the first branch, the complexity of this operation is $O(log_{32}(n))$. Like the RB version, `prepended` can be optimized by keeping transient states of the immutable vector 3.2.2).

**Drop** Like with the RB-Tree `drop` operation, the first step is to traverse down the tree to the cut index. The difference is that when cutting the nodes, instead of clearing the left side of the node the node is truncated and the sizes are adjusted. For each node the adjustment is equal to the number of nodes that will be dropped in the left of the subtree. This number is already part of the computation of the cut indices on each node.

The computational complexity of any split operation is $O(log_{32}(n))$ due to the traversal and copy of nodes on the branch where the cut index is located.

### 2.3.6 Related Structures

#### 2.3.6.1 Vector Builder and Iterator

The relaxation of the vector builder is discussed in section 3.3.1 and the relaxation of the vector iterator in section 3.3.2.

#### 2.3.6.2 Parallel Vector

The main difference between the RB and RRB parallel vectors is in the implementation of the combiner. This combiner is capable of combining in parallel and each combination is done in $O(log_{32}(n))$. The splitter also changed a bit to add an heuristic that helps on the performance of the combination and will tend to recreate balanced trees.

**Splitter** The splitter heuristic consists in creating partitions of the tree that contain a number of elements equal to a multiple of a completely filled tree (i.e. $a \cdot 32^b$ elements). The splitter will always split into two new splitters that have a size that is as equivalent as possible taking into account the first rule. To do so, the mid point of the splitter remaining elements is identified, then shifted to the next multiple of a power of 32. This way all nodes will be full and the subsequent concatenation rebalancing will be trivial. As all nodes are full, there is no node that requires shifting elements and therefore new node creation can be avoided in some cases.

**Combiner** The combiner is a trivial extension of the RRB-Vector builder. It wraps an instance of the builder and for each operation of the `Combiner` that is defined in `Builder` it delegates it to the builder. The `combine` operation concatenates the results of the two builders into a new combiner. The advantages of this combiner are that the combination is done in $O(log_{32}(n))$ and that they can run in parallel on the thread pool.

I

# Chapter 3

# Optimizations

## 3.1 Where is time spent?

### 3.1.1 Arrays

All nodes of the trees are represented with immutable `Array[AnyRef]`[1], as this is the most compact representation for the structure and will help performance by the use of JVM primitive operations (see 4.1). The leafs do not use specialized type arrays, as the type information is lost in methods due to generic types[2]. One consequence is that elements are already boxed when they arrive to the leafs and another one is the lack of runtime type information[3] that `Arrays` use upon creation in a generic context[4].

Most of the memory used in the vector data structure will be composed of arrays. There are 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 initialized. 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 arrays in the nodes) 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 slow down the update operations, as there is the need to copy the entire array for a single update. Benchmarks show that the sweet spot for performance is with blocks of size 32[5].

[1] Immutable arrays are just mutable arrays that do not get updated after initialization.
[2] Scala Collections [5] uses generic types sequences.
[3] A `classTag` implicit evidence.
[4] Limitations that could be circumvented by Miniboxing[6].
[5] Could also be 64, but the tradeoffs must be considered.

### 3.1.2 Computing indices

Computing the indices in each node while traversing or modifying the vector is key in performance. This performance 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 =
  (indexInTree >> (5*level)) & 31
```

This scheme can be generalized 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 generalize 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 generalization is not used because it reduces slightly the performance and it complicates other index manipulations.

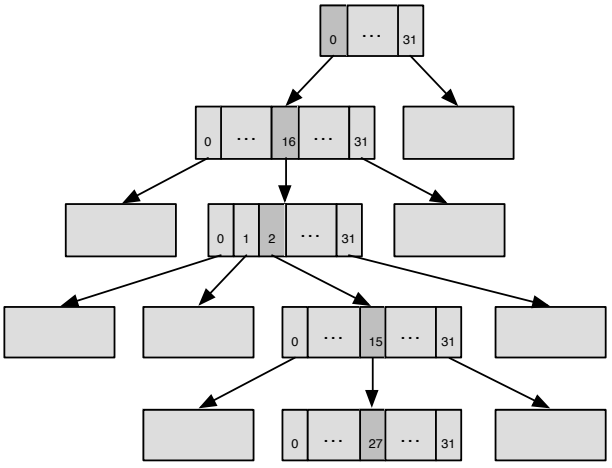


FIGURE 3.1: Accessing element at index 526843 in a tree of depth 5. Empty nodes represent collapses subtrees.

**Relaxing the Radix** When the tree is relaxed it is not possible to know the sub-indices directly from the index. That is why we keep the sizes array in the unbalanced nodes. This array keeps the accumulated sizes to make the computation of sub-indices as trivial as possible. The sub-index is the same as the first index in the sizes array where $index < sizes[subIndex]$. The simplest way to find this sub-index 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 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 sub-indices 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

When creating an implementation for a data structure it is always a good practice to have simple abstractions to simplify the code. This makes the code easier to read an reason about but it will complicate the execution of this code. In this section we'll focus on how the actual implementation optimizes out overheads that would be generated by functions and generic code abstractions.

**Functions** When writing an algorithm we usually divide it into small self-contained subroutines and we try to factor out the common ones. In practice this will add overhead for function invocations each time a function is used and break the locality of the code. To improve the performance we avoid as much as possible function call on simple operations. The code also avoids the creation of any higher order function to avoid additional object allocations. As such `while` loops are used rather than `for` loops.

**Generalization** A common way to reduce the amount of code that needs to be written when implementing common functionality is using generalization of code. But this comes with a computational cost for cases where a second implementation that takes into account the context of to simplify the operation. When considered beneficial the code is specialized by hand on the current context[6].

The base mechanisms to specialize are: specialization of a value, loops unrolling and arithmetic simplifications. Value specialization consist in explicitly identifying the value of some field and then using code specialized on that value. Loop unrolling (including recursions) consist in writing explicitly the code for each loop instead of the loop. This one is usually used on loops that are bounded by some value specialization. Once the first two specialization are applied code can be simplified with arithmetic transformations.

Concretely, most of the value specialization is done on heights and indices on radix operations. The height are a bounded small range of numbers and therefore can be efficiently specialized with a `match`[7] expression. On ranges of indices that are on trees of the same size specialization is done using nested `if/else` expressions. It is also possible to specialize of the first and last index, the first index (the 0 index) gains much from specialization because of arithmetic cancelation of terms. As an example here is a version of the radix `apply`.

```
def apply(index: Int): A = {
    vectorDepth match {
      case 1 =>
        vectorRoot(indexInNode & 31)
      case 2 =>
        val node1 = vectorRoot((indexInNode>>5) & 31)
        node1(indexInNode & 31)
      case 3 =>
        val node2 = vectorRoot((indexInNode>>10) & 31)
        val node1 = node2((indexInNode>>5) & 31)
        node1(indexInNode & 31)
      case 4 =>
        ...
    }
  }
```

Where the `vectorDepth` is specialized, then the recursion is rolled and finally a `>>` is removed on each branch of the `match`. Taking this one step further with the `head` method that is usually implemented as `apply(0)`, using the same specialization, removing the need for any additional arithmetic operation, the code becomes:

[6] This is specialization on value.

[7] All `match` expressions used in this specialization get compiled to `tableswitch` bytecode instruction. In some cases it would be beneficial to have `tableswitch` with fallthrough, but the there is currently no Scala construct that compiles to that.

```
def head(): A = {
    vectorDepth match {
      case 1 => vectorRoot(0)
      case 2 => vectorRoot(0)(0)
      case 3 => vectorRoot(0)(0)(0)
      ...
    }
  }
```

In relaxed radix operations the loop unrolling is usually avoided because in those methods the amount of nodes traversed can't be known in advance. But will invoke specialized versions of the code as soon as it finds a node on which it is possible. For a perfectly balanced RRB-Tree that node is to root, and hence the performance for such vectors is similar to the one for an RB-Tree.

## 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[8], they are allocated by default and nulled if the tree is 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.

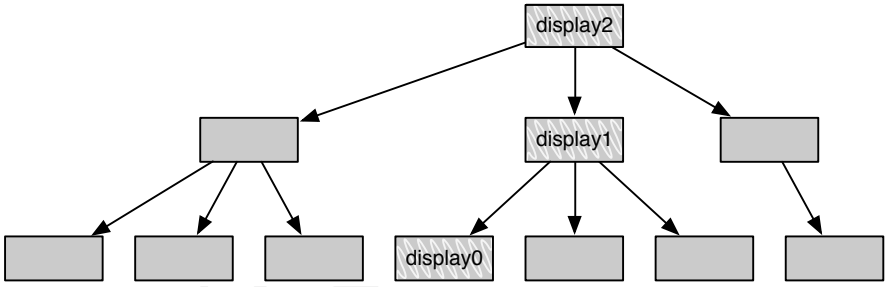


FIGURE 3.2: Displays

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 sub-indices 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[9]. 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 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^focus` will return a number that is bounded to the maximum number of elements in a tree of that level. The actual level can be extracted with some

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

[9] The display focus may change during the initialization of the object as optimization of some methods

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: 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`.

### 3.2.2 For transient states

Transient states is the key optimization 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 [3] this state is identified by the `dirty` flag[10].

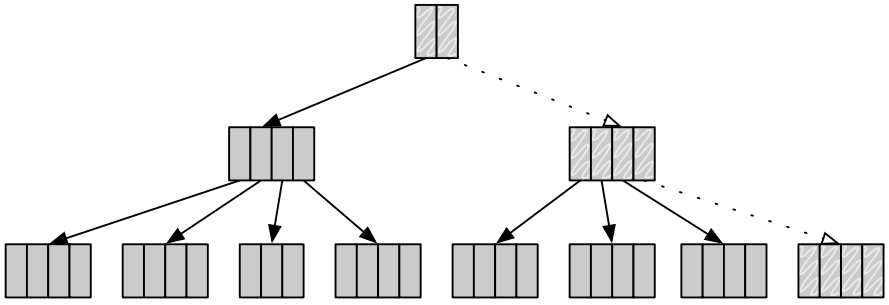

FIGURE 3.3: Transient Tree with current focus displays marked in white and striped nulled edges.

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.

[10] In the RRB Vector [2] implementation it was replaced by `transient`.

There is a cost associated to the transformation from canonical 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 canonical 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 able 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[11]. 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[12]. 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 `apply` would become:

```
def apply(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 `apply` method 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) starting from `display1`. The rest is fetched from the root using `getElementFromRoot`.

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

[12] As an optimization, 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.

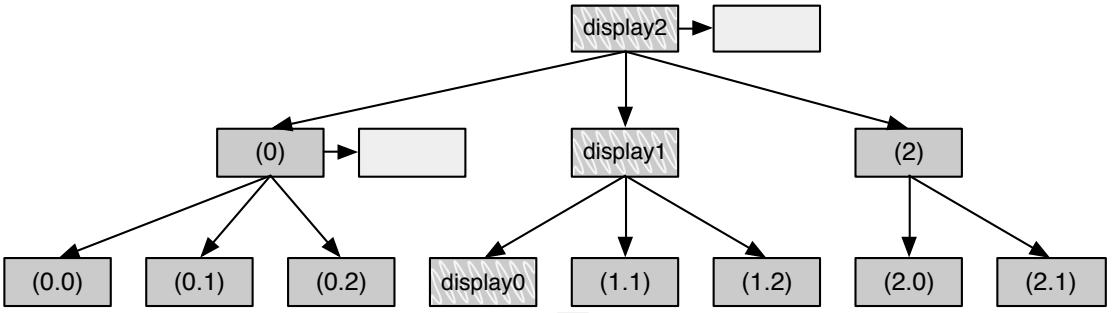


FIGURE 3.4: Relaxed Radix Balanced Tree with a focus on a balanced subtree rooted of `display1`. Light grey boxes represent unbalanced nodes sizes.

### 3.2.4 Vector Canonicalization

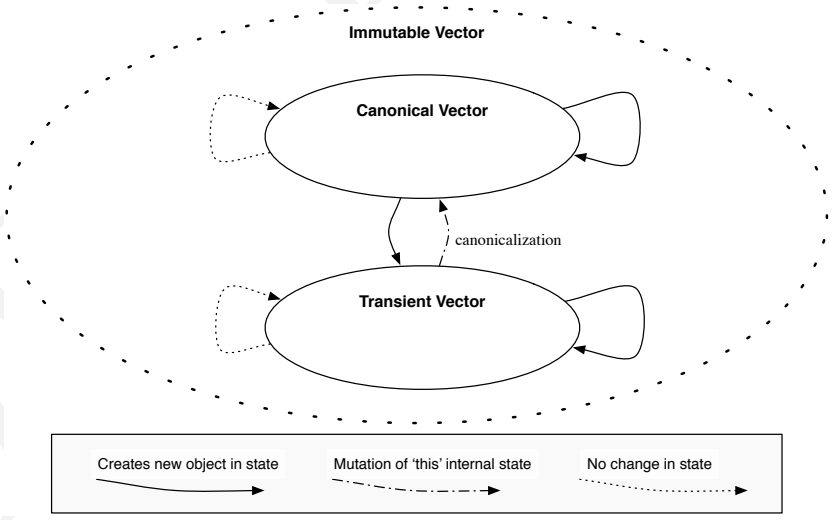


FIGURE 3.5: Objects states and effect of operations.

As stated in section 3.2.2 vectors have two representations: canonical and transient. The transient state aims to improve performance of some operations by amortizing costs. But, the transient state is not ideal for performance of other operations. For example an `apply` operation on an unbalanced vector may lack the sizes information it requires to access certain indices. And an iterator relies on a canonical tree for performance. It is always possible to implement those operations on transient states, but they would involve additional overhead on each call and duplication of code.

The solution is to convert the transient representation to a canonical one when an operation that requires it is called on an instance of the immutable vector. The mutation of the vector is not visible from the outside and only happens at most once (see figure 3.5). This transformation only affects the nodes that are on the display, it copies each one (except the leaf) and links the trees. If the node is unbalanced, the size of the subtree in focus is inserted. This transformation could be seen as a lazy initialization of the current branch.

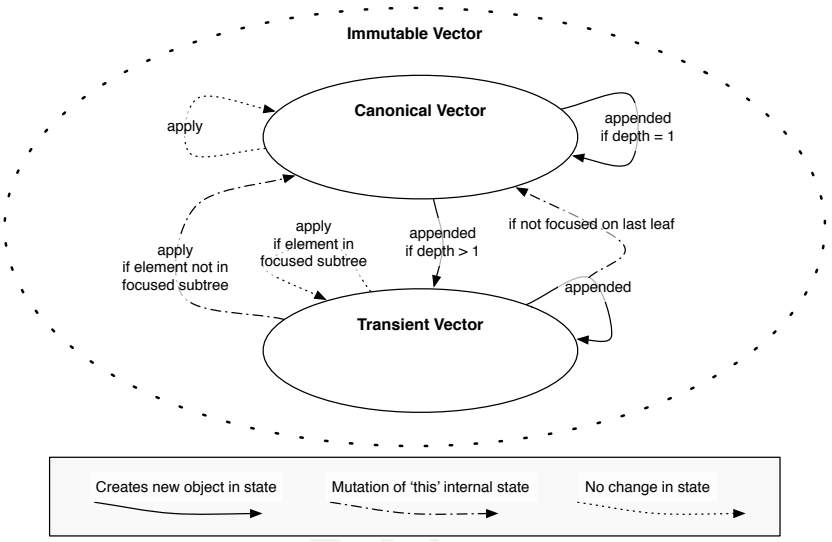


FIGURE 3.6: Transient state creation and canonicalization of vectors for `append` and `apply` operations.

Vector objects can only be in transient state if they where created that way. For example, the `append`/`prepend` operation will create a new object that is in transient state and focused on the last/first branch. If the source object was not focusing the last branch, then it is canonicalized (if needed) before change of branch operation. Vectors of depth 1 are special cases, they are always in canonical form and their operations are equivalent to those in transient form.

Figure 3.6 shows the different states and the effects of the `append` and `apply` operations on them. Figure 3.7 shows the same for all operations. Note that the mutation of `this` object only happened from transient to canonical and that the only way to get to transient is with the creation of a new vector object.

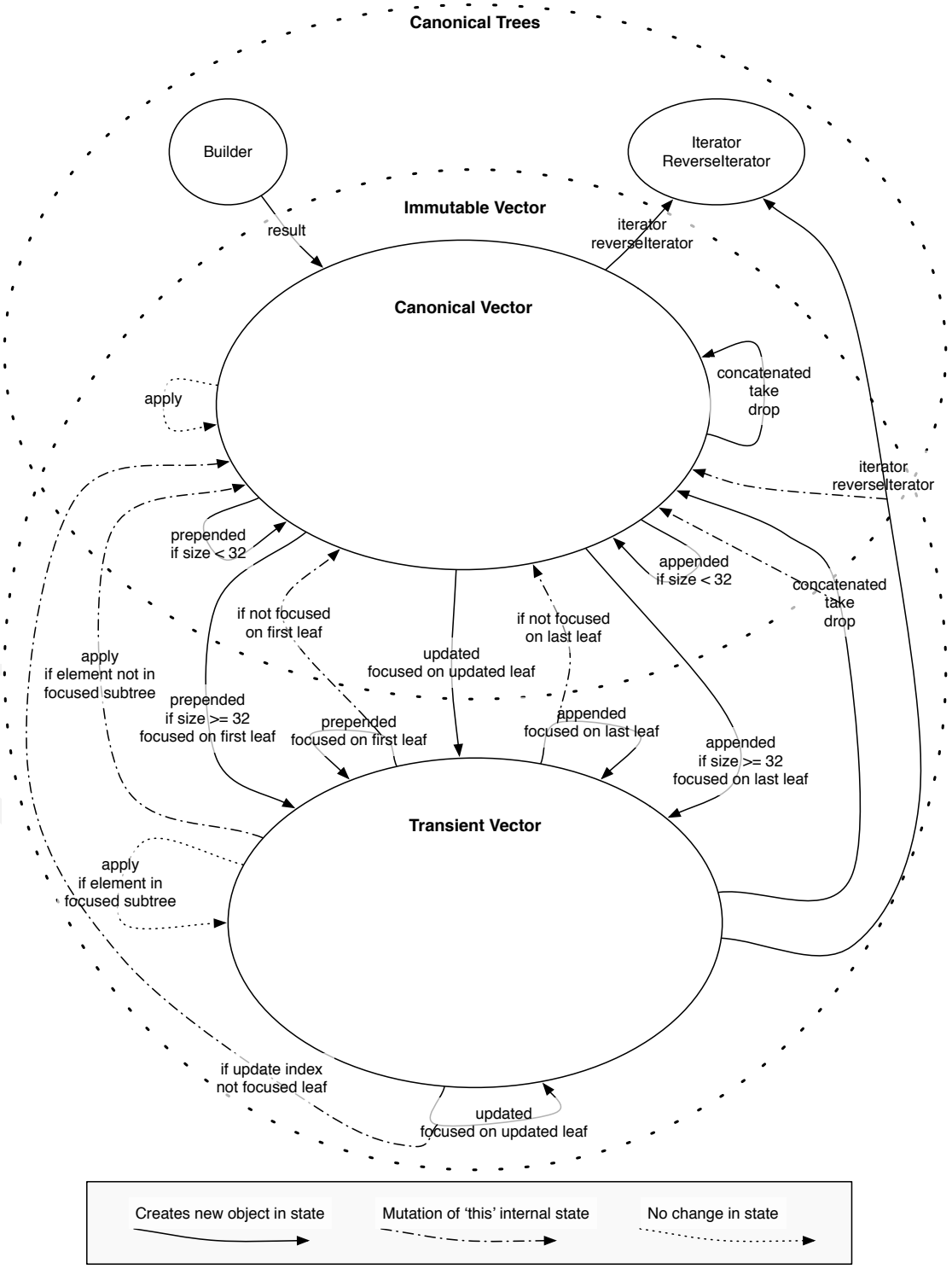


FIGURE 3.7: Transient state creation and canonicalization of vectors for every operation.

## 3.3 Related Structures

### 3.3.1 Builder

A vector builder is a special wrapper for a RB-Tree that has an efficient `append` operation. It is implemented using encapsulated mutable arrays during the building of the vector and then frozen on the creation of the result. Any array that the builder can still mutate is cloned and possibly truncated to generate the RB-tree of the vector.

The benefits of the mutable `append` operation of the builder over `appended` operation of the vector are on the reduced amount of memory allocations needed in the process of appending. There is no need to allocate a new `Vector` object each time an element is appended. Even more important is that there is no need to create a new array in each change on a node, nodes are allocated once and filled as needed.

**Relaxing the Builder** Most of the operations implemented in the collections framework that use builder usually create the new collection by appending one element at a time. To retain the performance of all existing operation that use builders the implementation builder append does not change. Therefore the tree where elements are appended is always perfectly balanced.

To add performance when concatenating a `Vector` to a builder a new field is added to retain a vector that will be lazily concatenated to the result. This vector is the accumulation `acc` of all elements (or all vectors) that where added before (and including) the last concatenation and elements in the current tree been build. All elements added after the last concatenation are added to the main the mutable tree of the builder.

### 3.3.2 Iterator

The default implementation for iterator in `IndexedSeq` is by iterating on the indices and accessing elements with the `apply` method. This is not optimal because it requires traversing the tree down from the root each time an element is accessed. The traversal of the vector using this would have a computational complexity of $O(n \cdot log_{32}(n))$.

To improve performance of the iteration on vector a normal inorder tree traversal of the underlying RB-Tree amortizes the computational complexity to $O(n)$. The iterator can use the display optimization to keep track of the current branch and move to the next

branches efficiently using the radix index scheme. In this case the display is allowed to mutate. The current implementation of vector is implemented this way[13].

**Relaxing the Iterator** To keep the performance advantages of the tree traversal on balanced RRB-trees, iteration is done the same on every balanced subtree. When the end of a balanced subtree is reached the first element of the next balanced subtree is focused and the traversal continues efficiently from there. If the tree is not too unbalanced the traversal will tend to be $O(n)$. But if the tree is extremely unbalanced it will tend to fall back on $O(n \cdot log_{32}(n))$, where the additional $log_{32}(n)$ comes from the changes of balanced subtrees. Even in this bad last case the performance is better than the traversal by index because each leaf is a balanced subtree, on which efficient traversal is used. The RRB-Vector implements both `iterator` and `reverseIterator` this way.

I



[13]The current `reverseIterator` is still using traversal by index.

# Chapter 4

# Performance in Practice

## 4.1 In practice: Running on JVM

In practice, Scala compiles to Java bytecode and executes on a JVM, where we take the Java SE HotSpot as a reference. This imposes additional characteristics of performance that can't be evaluated on the algorithmic level alone.

The JVM use *just in time* (or JIT) compilation of code to take advantage of knowledge about how the code is used at runtime. At first it runs on interpreter mode, it consists of interpreting the bytecode and collecting statistics on the codes execution. Eventually it will compile the code to make it faster using several optimizations and heuristics. The code of the vectors tries to gain performance by aligning with those heuristics and hence taking advantage of the JVM optimizations.

One of the components of the JVM that will be affected by vectors is the *garbage collector* (or GC). The vector tends to create a large amount of `Arrays` during transformations, of which many are only necessary for a short time. Those objects will use up memory and possibly degrade performance of future allocations, until the next GC. Having all these unused object will also contribute in an increase in the frequency of GC executions. For that reason the code is optimized to avoid excessive creation of intermediary objects.

When running on some machine we have several memory cashes helping with performance. The vector tries to align to the underlying cache model to improve performance. The size of the arrays is chosen to take advantage of cache lines while they are copied and traversed. This makes the behaviour of node copies look more as a constant time operation. Further more when copying arrays the `System.arraycopy` primitive is used. This way, the critical operation that is executed on each update will use the lowest level

implementation available for the JVM. This could potentially go down to efficient host machines code when compiled.

### 4.1.1 Cost of Abstraction and JIT Inline

One of the optimizations that the JVM provides is function call elimination (abstraction elimination) based on a set of heuristics. This is equivalent to the functions optimization in section 3.1.3 but done on another level of the pipeline. Critical parts of the code are written with careful detail to match the heuristics of the JVM.

The optimization works as follows: the code is run in interpretation mode first while keeping track of statistics on which parts of the code are executed and how many times. When the code is eventually compiled, function calls that are deemed *hot* are inlined. The heuristic favours inline when the function has less than 35 bytes.

We use this knowledge of the heuristic in two ways. The first is to avoid inlining everything, if the function is small and there is no other optimization that can be done if inlined manually, the code is kept as a function. The second is to inline the methods of the vector API into the clients code. When implementing functions with less than 35 bytes, performance was tested and cross referenced with the VMs inlining diagnostic[1]. The code was inspected using `javap -c`, which gives the sizes of the function and helped identifying some optimization possibilities.

[1]The JVMs `"-XX:+UnlockDiagnosticVMOptions -XX:+PrintInlining"` flag was used to track the inlining.

## 4.2 Measuring performance

ScalaMeter framework[7] is used to measure performance of operations on different implementations of vector. This framework was used to identify performance improvements or regressions during the development process.

To have reproducible results with low error margins, ScalaMeter was configured on a per benchmark basis[2]. Each test is run on 32 different JVM instances to average out badly allocated VMs. On each JVM 32 measurements are taken, while using outlier elimination to remove benchmark runs that where exceptionally different. This could happen if one particular run has a garbage collection, JIT compilation of code or OS executing something else. Before the benchmark runs the JVM is warmed up by running some times the benchmark without taking measurements. During these first execution the VM will be loading classes, taking some statistic on the code and then eventually compiling it.

There are two main axis of performance comparisons. The first is between the RB-Vector and a perfectly balanced RRB-Tree where the aim is to have an equivalent performance, even if the RRB-Vectors have an inherent additional overhead. The second axis is the one that shows the effects of unbalanced nodes on RRB-Tree. For this we compare the same perfect balanced vector with one that is the result of one concatenation of two vectors and with an extremely unbalanced vector. The later vector is generated by concatenating random[3] small vectors together. The amount of unbalanced nodes is in part affected by the size of the vector. Other axes are discussed in the next section.

[2]The JVM `-XX:+PrintCompilation` flag vas used to help identifying the ideal configuration.
[3]Pseudo random generator where used to be able to reproduce the same ones each time.

## 4.3 Implementation Generators

Until now, only one implementation of the RRB-Vector[4] was mentioned, comparing it against the RB-Vector. But in fact to compare performance on different characteristics like block sizes and concatenation algorithm variant concrete implementations where generated using Scala reflection. Other characteristics involve a complete structure assertion while testing and benchmarks generation. For each combination of characteristics there is a concrete artefact: class implementation, tests and benchmarks.

Code is generated by combining AST using Quasiquotes with some domain specific optimizations. The optimizations are the same that where applied manually on the main implementation of the RRB-Vector. The resulting code is equivalent, except that it lacks formatting. The name of the artefact is used to identify the different characteristics[5].

The block sizes used where 32, 64, 128 and 256. The main aim is to show the differences in performances of the operations and identify sizes for which performance degrades due to loss of cache locality. This is the only number in the name of the artefact.

Both versions of the RRB-Vector concatenation algorithm are used to generate vector implementations. This is done mostly to compare the performances of other operation on vector that got unbalanced using concatenation. `Complete` (or `c` in the name of the artefact) is used to name identify the version that rebalances completely the subtree. The other one is named `Quick` or `q` in the name of the artefact.

For testing purposes, for each implementation there is a second one generated with heavy assertions on the whole structure of the vector on most methods (see 5.2). Concrete benchmarks are also generated at the same time.

Generating this code was the only reasonable way to implement this huge amount of classes while implementing new features on them. Changes that would otherwise had to be propagated by hand on each one. This also help to find and fix bugs, because a bug has a higher probability of affecting at least one of the artefact and a fix for a bug is reflected on all of them.

[4] Located in the `scala.collection.immutable.rrbvector` package on GitHub

[5] All generated artefacts are in the `scala.collection.immutable.generated.rrbvector` package on GitHub

## 4.4 Benchmarks

Performance benchmarks aim to compare the performance of all core operations of the vectors. These operations are: `apply`, `concatenate`, `append`, `prepend`, `take`, `drop`. The performance of specialized operations for the iterators, builder and the parallel vector where also benchmarked. Additionally, the memory footprint of different vectors was checked.

The axis of comparisons are:

- Size of the vector. Split into ranges that correspond to vectors of the same height, when perfectly balanced.
- `Vector` against completely balanced `RRBVector`
- Differently balanced `RRBVectors`: perfectly balanced, unbalanced (by one concatenation) and extremely unbalanced. This is ignored on vector of depth lesser than 3 because there all those vectors end up being perfectly balanced.
- Different block sizes: 32 (original), 64, 128 and 256.
- Different implementation of the rebalancing in `concatenate`: Complete rebalance and Quick rebalance
- Thread pool size for parallel vectors.

For the results of this sections, all benchmarks where executed on a Java HotSpot(TM) 64-Bit Server VM on a machine with an Intel(R) Core(TM) i7-4770 CPU @ 3.40GHz with 32GiB on RAM. Each benchmarking VM instance was setup with 16GiB of heap memory.

### 4.4.1 Apply

The `apply` benchmarks aim to show the amortized access time of elements. Amortizing the displays optimization and memory caches. For this, each benchmark invokes the `apply` operation $10k$ times on different elements.

There are two variants: the first variant access the elements sequentially and the second one chooses each time a random index. The first one is supposed to have better performance because the array that are access tend to be in cache. The computations of the next index is also a bit more complex on the second one, but mostly irrelevant compared with the cache misses of all the arrays on a branch.

The benchmark results for the sequential access in figures 4.1 and 4.2 show that if the RRB-Vector is balanced or just slightly unbalanced will be faster than the RB-Vector. But when it gets extremely unbalanced the performance can drop by around 0.5X which is proportional to the increase of arrays that must be accessed (two per node).

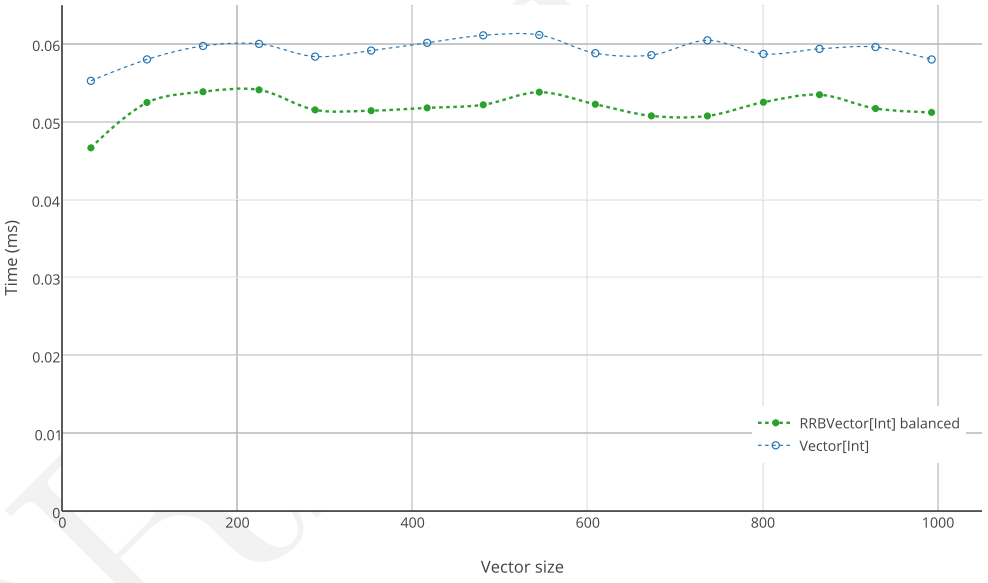


FIGURE 4.1: Time to execute 10k apply operations on sequential indices on vectors of height 2.

The performance improvements where achieved by analyzing the sizes of the functions involved, there inlining and which parts of the code is used in each case. The aim was to improve performance of unbalanced vectors, that is why in some cases the slightly unbalanced vector is the fastest.

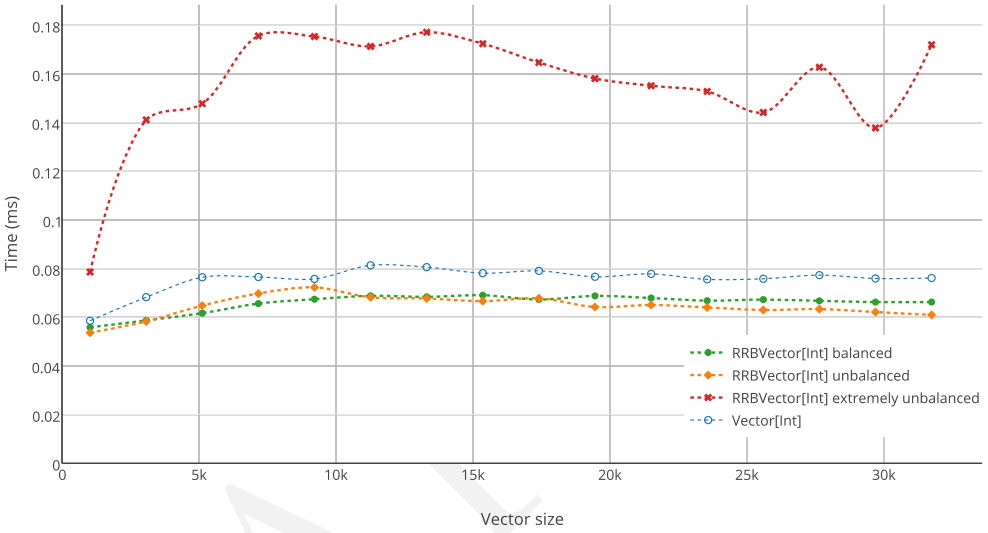


FIGURE 4.2: Time to execute 10k apply operations on sequential indices on vectors of height 3.

Figure 4.3 shows the results for random indices. It can be observed that the behaviour of the curve is quite similar, but the sequential one is around 2X faster due to memory locality.

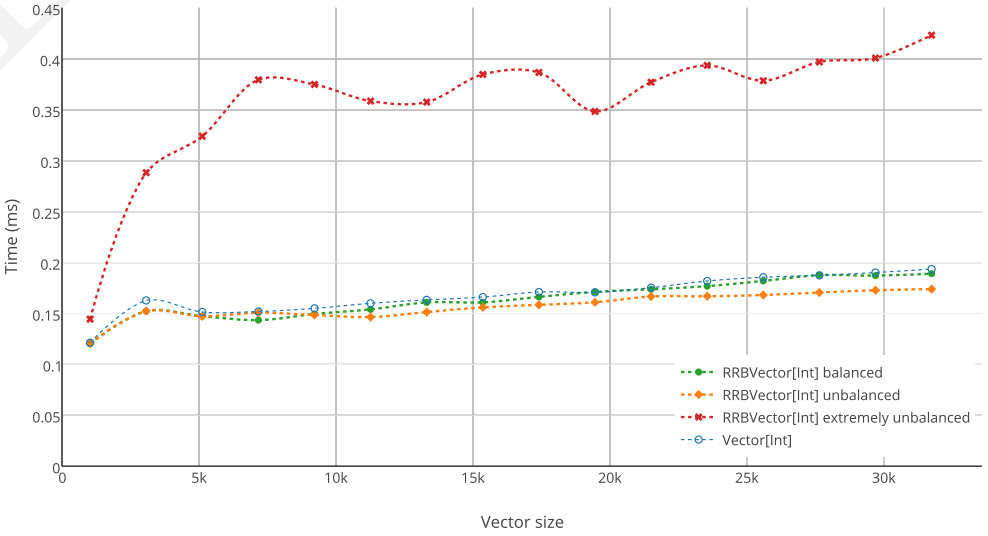


FIGURE 4.3: Time to execute 10k apply operations on random indices on vectors of height 3.

Finally, the figures 4.4 show what would happen if the block sizes changed or if the

rebalance algorithm is changed. Note that in the cases for blocks of 128 and 256 the complete rebalance is actually creating completely balanced vectors. It can be observed that the quick rebalance is harmful to the performance of the apply method. Increasing the sizes of the arrays will improve the performance of the apply method.

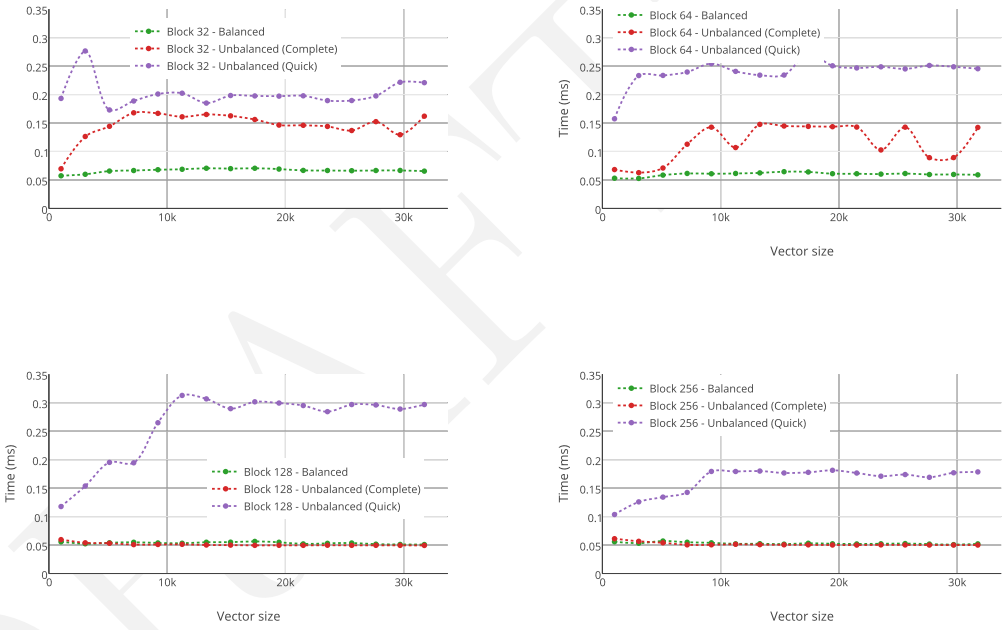


FIGURE 4.4: Time to execute 10k apply operations on sequential indices. Comparing performances for different block sizes and different implementation of the concatenation inner branch rebalancing (Complete/Quick).

### 4.4.2 Concatenation

The `concatenation` benchmarks aim to show the benefit of having the logarithmic concatenation algorithm against the linear version. The benchmarks concatenate two vectors of the same type and balance characteristics. As such these benchmarks have two axis representing the sizes of the LHS and RHS vector in the operation.

Figure 4.5 shows how the concatenation time for RB-Vectors is linear on the size of the resulting vector. Below that, the three surfaces that represent the RRB-Vector concatenation are effectively constant time (or $log_{32}(n)$). The performance difference between the differently balanced RRB-Vectors is negligible in relation to the difference with RB-Vector.

In the case of concatenation, being balanced or not does not impose performance improvements or degradations. Performance will be determined by the number elements in the branches that get merged and their alignment.

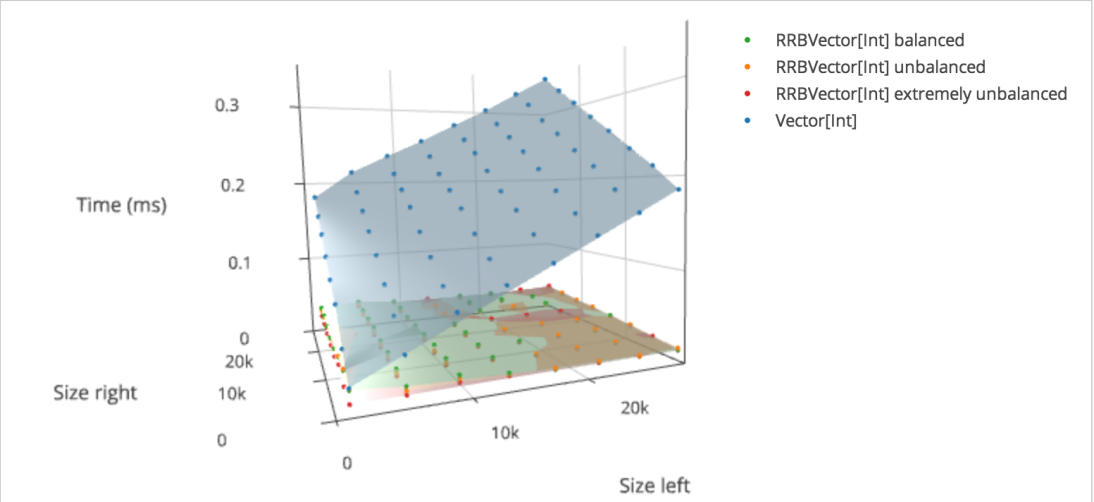


FIGURE 4.5: Execution time for a concatenation operation on two vectors. In theory (and in practice) Vector concatenation is $O(left + right)$ and the rrbVector concatenation operation is $O(log_{32}(left + right))$.

The benchmark for the Quick rebalancing version where not included here because the implementation for that one is suboptimal and the comparison would not be fair. The main purpose of that version is to show the differences it would cause on other operations. The current version of the Quick rebalancing can achieve 2X performance benefit, but it could gain a lot form the same optimizations that were used on the Complete rebalancing.

### 4.4.3 Append

The `append` benchmarks aim to show the amortized time taken to append elements. With the transient states, the branch updates are amortized over leaf updates. For this, each benchmark appends 256 elements on a vector. This way there is at least one branch update for each benchmark, 8 if the block size is 32.

The figures 4.6 and 4.7 show that if the RRB-Vectors are balanced or just slightly unbalanced the execution time is equivalent to the one for RB-Vectors. But, if it gets extremely unbalanced the performance can fall to by 0.6X. In figure 4.6 it is possible to see that after 768 the performance does a step upward, this correspond to the addition of a level in the tree.

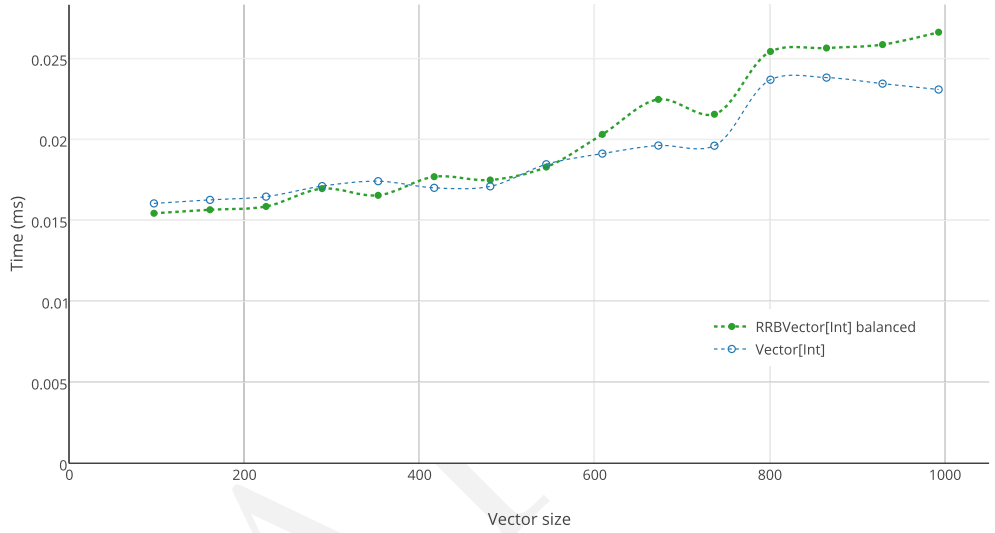


FIGURE 4.6: Time to execute 256 append operations on vectors of height 2. This shows the amortized cost of the append operation.

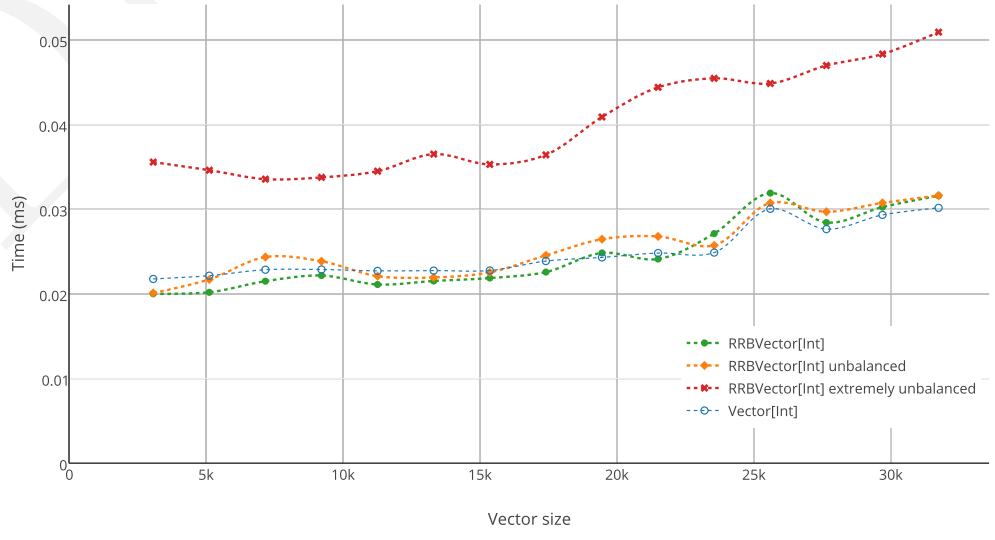


FIGURE 4.7: Time to execute 256 append operations on vectors of height 3. This shows the amortized cost of the append operation.

Finally, the figures 4.8 show what would happen if the block sizes changed or if the rebalance algorithm is changed. Note that in the cases for blocks of 128 and 256 the complete rebalance is actually creating completely balanced vectors (from 16384 down in the case of 128). It can be observed that the quick rebalance is hurtful to the performance

of the append method. Increasing the sizes of the arrays will decrease the performance for balanced trees, but can potentially reduce the amount of unbalanced nodes in the extremely unbalanced trees.

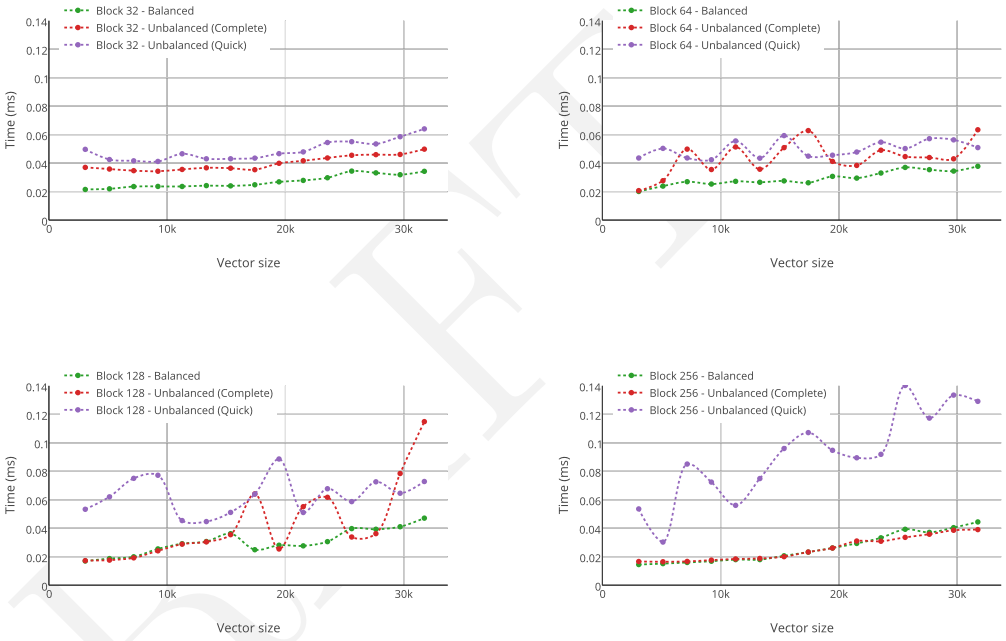


FIGURE 4.8: Time to execute 256 append operations. This shows the amortized cost of the append operation. Comparing performances for different block sizes and different implementation of the concatenation inner branch rebalancing (Complete/Quick).

### 4.4.4 Prepend

The `prepend` benchmarks aim to show the amortized time taken to prepending elements. With the transient states, the branch updates are amortized over leaf updates. For this, each benchmark appends 256 elements on a vector. This way there is at least one branch update for each benchmark, 8 if the block size is 32.

The figures 4.9 and 4.10 show that the prepend operation on a balanced RRB-Vector is a bit slower. This is because prepending on this kind of vector requires the creation or update of an unbalanced node. But, this is a a case where the unbalanced vectors end up being more performant. This is because on these vectors it is possible to find space to prepend elements on the left of the tree. In figure 4.9 it is possible to see that after 768 the performance does a step upward, this correspond to the addition of a level in the tree.

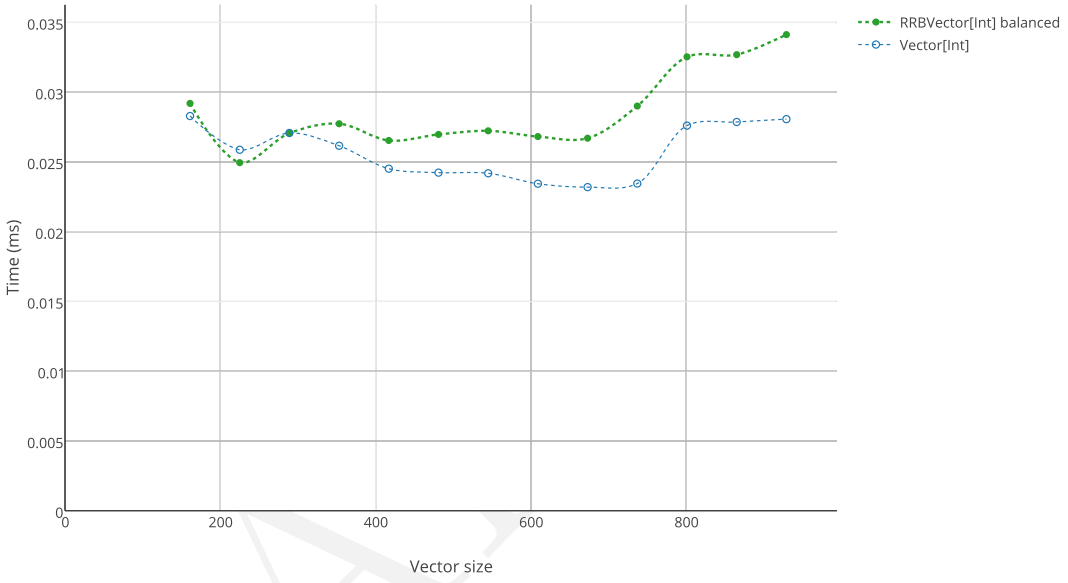


FIGURE 4.9: Time to execute 256 prepend operations on vectors of height 2. This shows the amortized cost of the prepend operation.

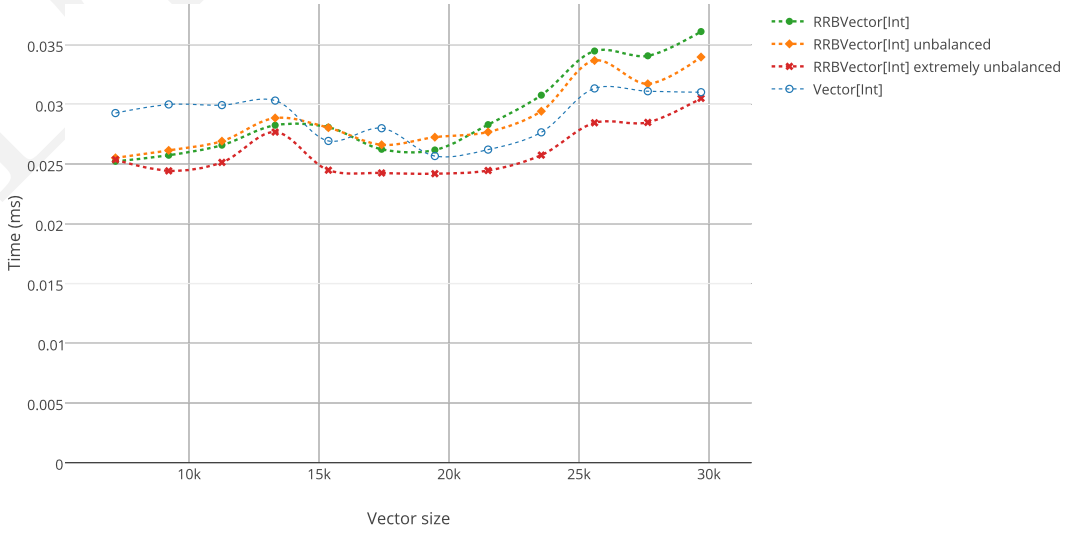


FIGURE 4.10: Time to execute 256 prepend operations on vectors of height 3. This shows the amortized cost of the prepend operation.

Finally, the figures 4.11 show what would happen if the block sizes changed or if the rebalance algorithm is changed. Note that with 128 and 256 there are some performance

degradation. They probably reflect the limit in cache-line locality where the node updates become linear again due to memory accesses and updates. The performance for blocks of size 32 and 64 are quite similar, but it is more stable for 64 because it needs a lesser number of branch updates.

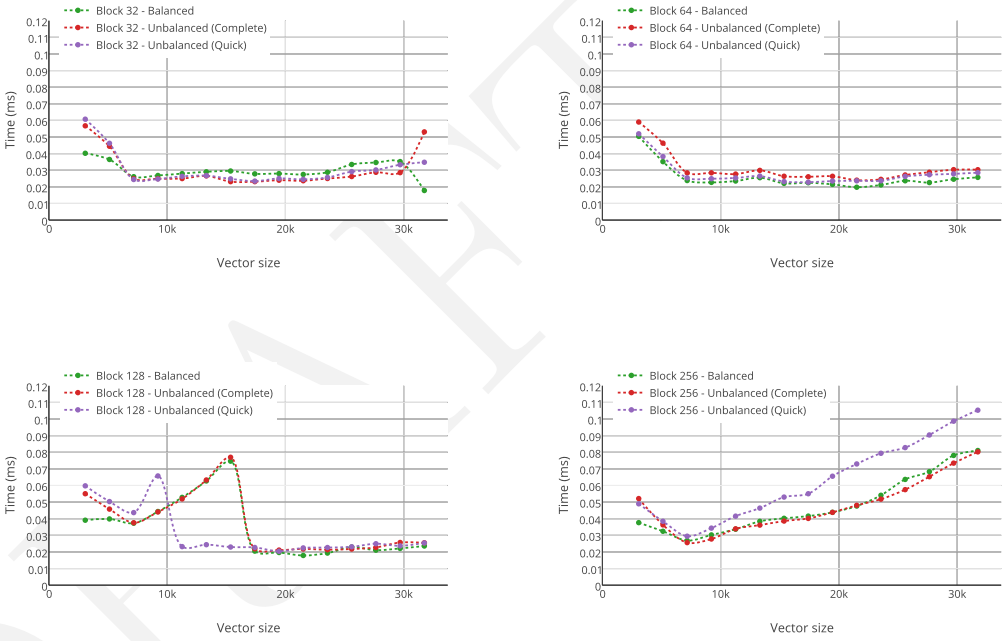


FIGURE 4.11: Time to execute 256 prepend operations. This shows the amortized cost of the append operation. Comparing performances for different block sizes and different implementation of the concatenation inner branch rebalancing (Complete/Quick).

### 4.4.5 Splits

The `take` and `drop` benchmarks aimed to show the performance of those operation on a couple of splitting points. The splitting points where done on the middle of the vectors an on the first quarter of the vector.

Figure 4.12 shows the performance for the take operation, and figure 4.13. The performances are not always ideal. But the different types of vector have a consistent splitting time with different splitting indices and operation (`take` against `drop`).

To be consistent with the other benchmarks, the characterization of vector was done by size. But, maybe this was not ideal as the characteristics of the branch that is being split and the index where it's split are more influential on performance than the size.

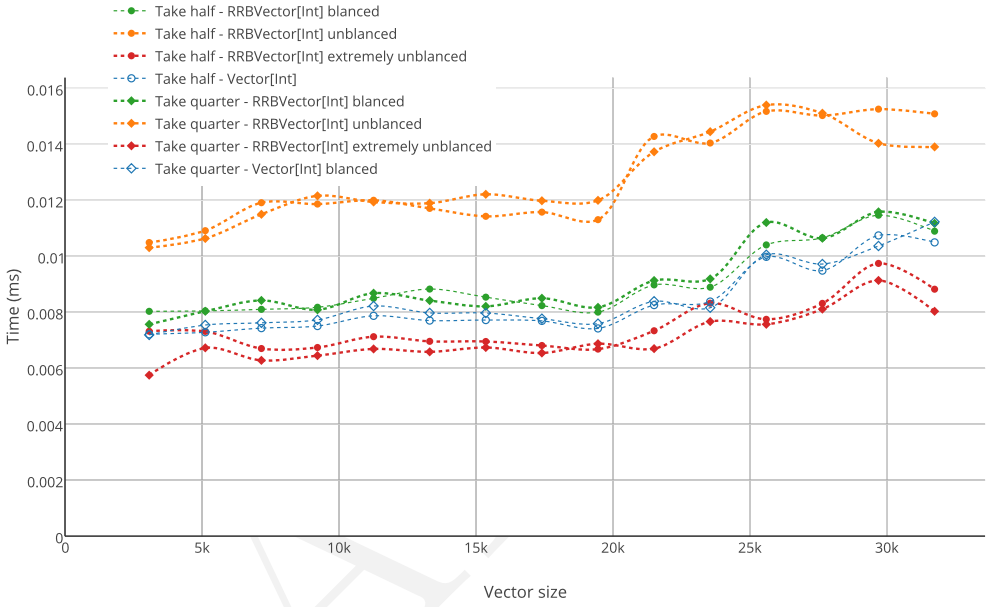


FIGURE 4.12: Execution time of take.

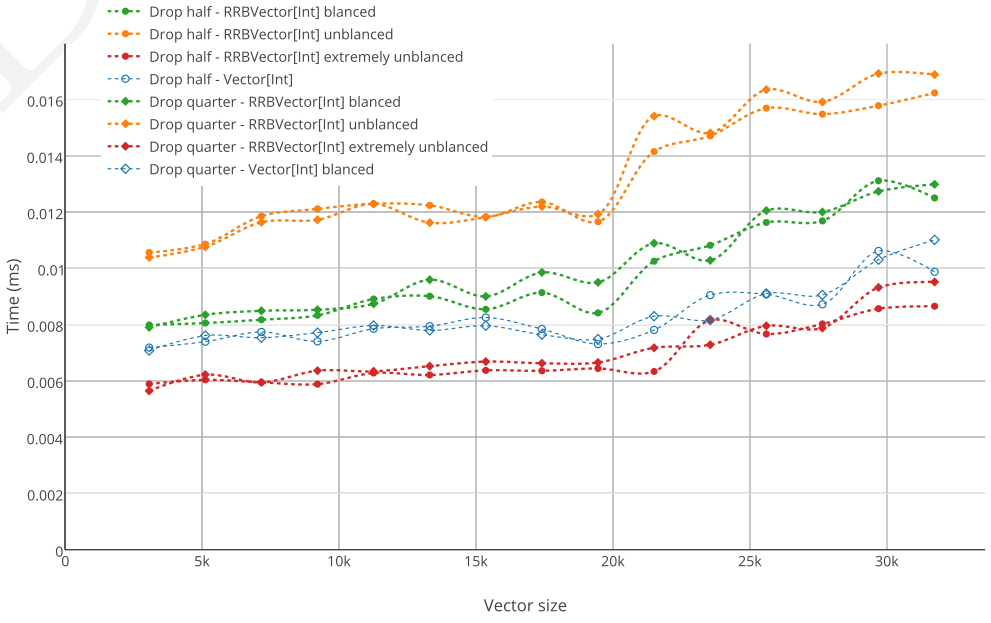


FIGURE 4.13: Execution time of drop.

### 4.4.6 Iterator

The iteration benchmarks aim to show the performance of traversing the whole vector using an iterator. Linear behaviour is expected from these benchmarks.

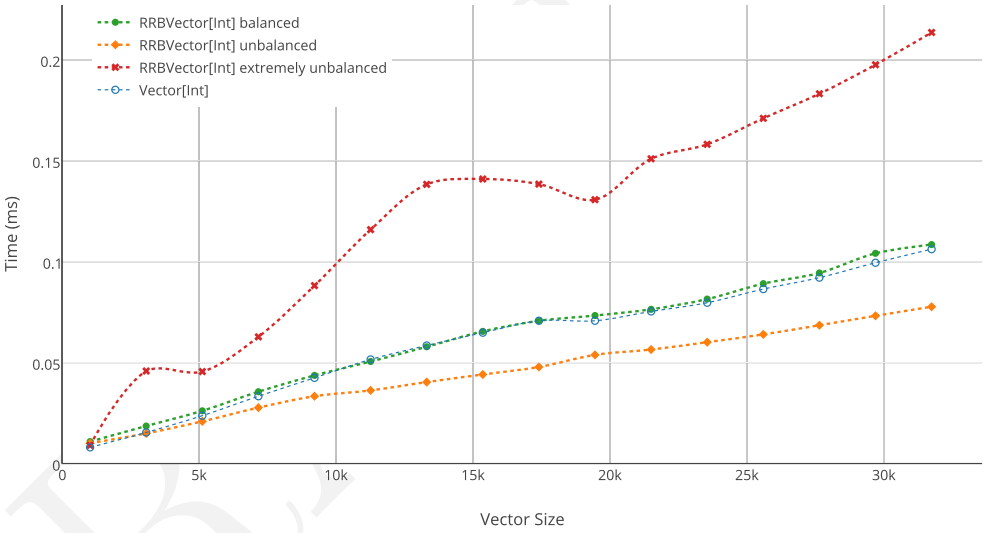


FIGURE 4.14: Excecution time to iterate through all the elements of the vector.

Figures 4.14 and 4.15 show that for balanced RRB-Vectors the performances is identical. For slightly unbalanced once there is an increase in performance that seams to come from JIT optimizations of the code. In that case the code that is used to change branches uses fewer lines of code in each function (some are never access at runtime), it could be possible that the optimizer is using this to improve the performance[6].

When the vectors become extremely unbalanced, the performance degenerates by a factor proportional to the height of the vector. Because it changes from one balanced subtree to the next one from the root. It should be possible, by keeping more state in the iterator, to improve the performance to go from one balanced subtree to the next. But this would increase the initialization time and might affect smaller vectors.

[6]Further analysis on this situation should be done in the future.

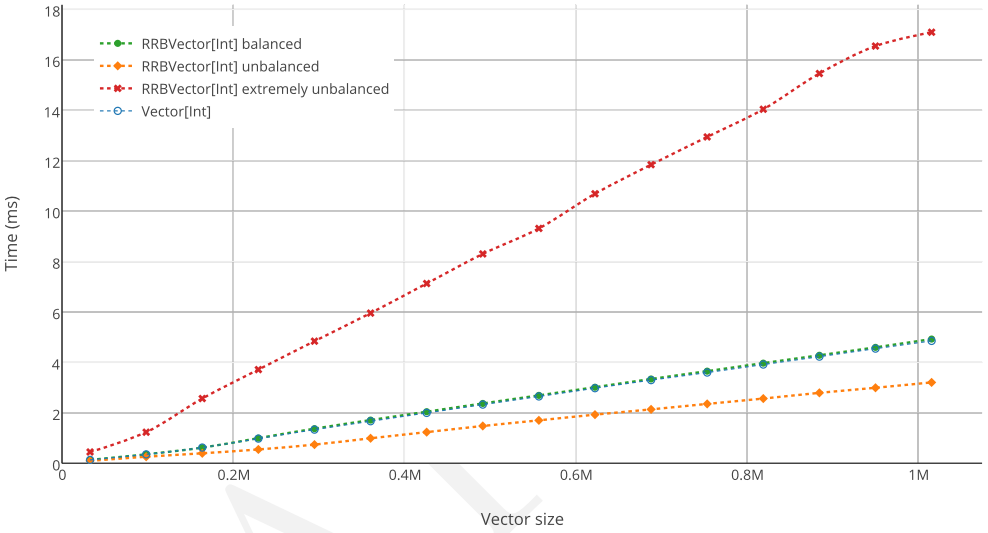


FIGURE 4.15: Excecution time to iterate through all the elements of the vector.

Finally, the figures 4.16 show what would happen if the block sizes changed or if the rebalance algorithm is changed. In general the quick rebalancing is usually worst than complete rebalancing for the iterator. The block sizes do not change much the performance of the iterator. This is a great result because it means that the overhead of traversing the tree is insignificant when traversing the vector, the cost is in the traversal of the leafs.

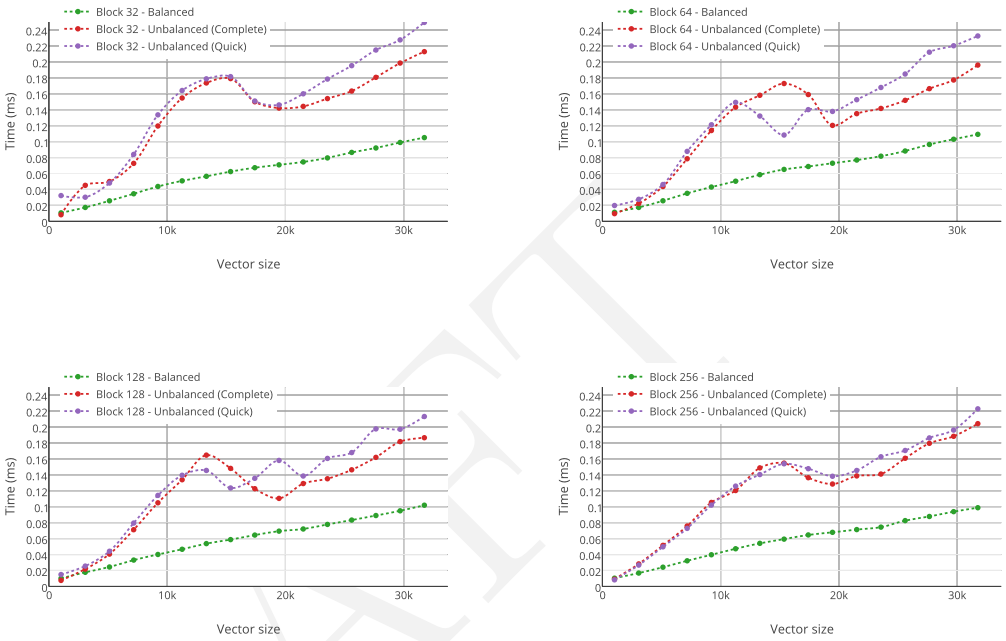


FIGURE 4.16: Excecution time to iterate through all the elements of the vector. Comparing performances for different block sizes and different implementation of the concatenation inner branch rebalancing (Complete/Quick).

### 4.4.7 Builder

The builder benchmarks aim to show the time it takes to build a vector of some size by appending elements. The performance of this is expected to be linear. Note that all vectors that are generated will be completely balanced.

Figures 4.17 and 4.18 show that the RB and RRB vector builder have the same results. This is not surprising, because the have the same code for appending elements, except for a small detail. The RRB Vector allocates arrays with one additional slot for all branch nodes.

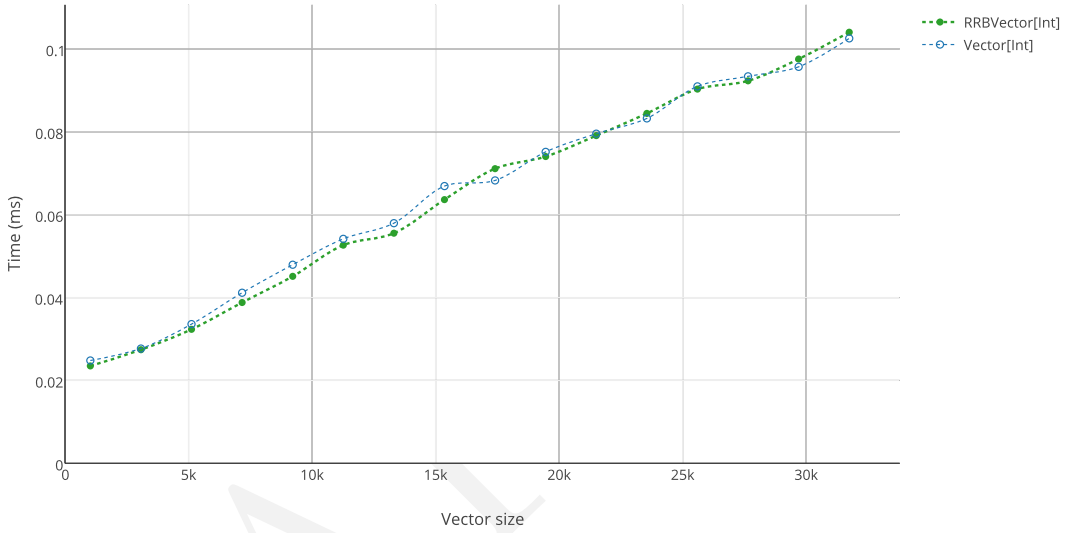


FIGURE 4.17: Execution time to build a vector of a given size for trees of depth 3.

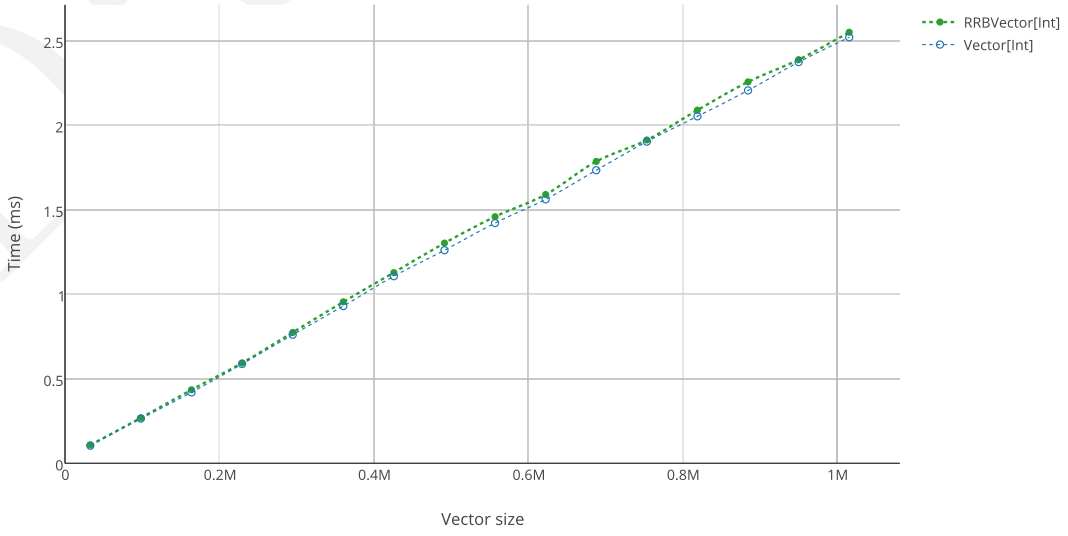


FIGURE 4.18: Execution time to build a vector of a given size for trees of depth 4.

Increasing the sizes of the blocks will reduce the amounts of blocks that need to be allocated during the building. Larger blocks will also take slightly longer to allocate. Figure 4.19 shows that the difference isn't substantial, but has a sweet spot on blocks of size 64 for builders.

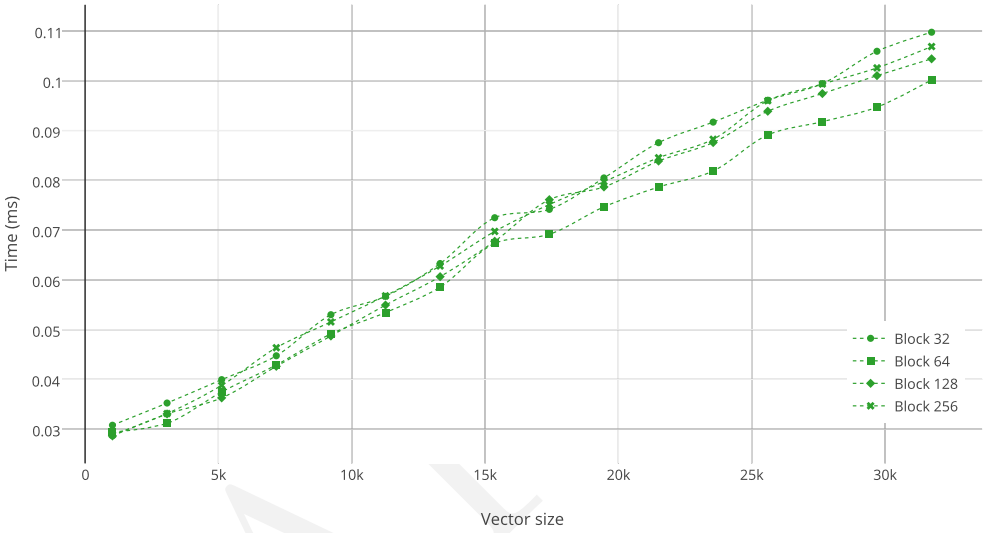


FIGURE 4.19: Execution time to build a vector of a given size. Comparing performances for different block sizes.

### 4.4.8 Parallel split-combine

The aim of the parallel benchmarks is to show the overhead of the parallel framework. Specifically, the time spent in splitting and combining the vectors during parallel execution. For this the benchmarks execute a `map` on the identity function[7]. This operation is done on the standard sequential version of the vector and on the parallel one.

In figure 4.20 it can be seen that the base sequential `map` is exactly the same for both implementation. This was expected because the `map` implementation uses an `iterator` and `builder` to compute it in both cases, which have the same performance.

The results for the parallel RB `Vector` show that with 1, 2 and 4 threads the performance of the operation is worse. With 8 threads it becomes a little bit faster. This is what we expected, due to the lazy concatenation at the end of the combination of results. But, it shows that this parallel collection is useless in practice.

When using the RRB-Vectors parallel, the parallelism is evident. With two threads the performance improvement is of almost 2X, then it tends to go towards 2.5X. This is the type of parallelism behaviour that is expected (see Amdahl's law[8]). Even the case on a thread pool of one thread is a bit faster than the normal sequential one, as the thread is dedicated.

[7]The function that simply returns the element passed as parameter.

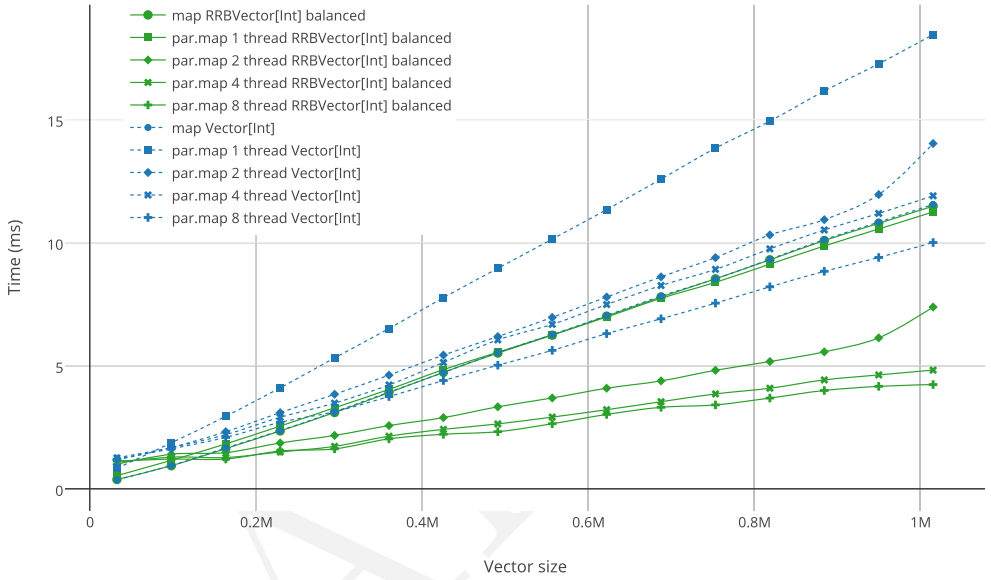


FIGURE 4.20: Benchmark on map and parallel map using the function (x=>x) to show the difference time used in the framework. This time represents the time spent in the splitters and combiners of the parallel collection (iterator and builder for the sequential version).

Figure 4.21 shows how unbalanced parallel vector behave. There is a slight loss in performance when they are unbalanced. But, this operations will create a balanced vector and therefore this small overhead can be amortized over the next operations.

For vectors smaller than 1024 the combine operation parallelism is lost. This is a consequence of the rebalance in the concatenation algorithm. Better calibration or a different approach for this range of sizes could improve it further.

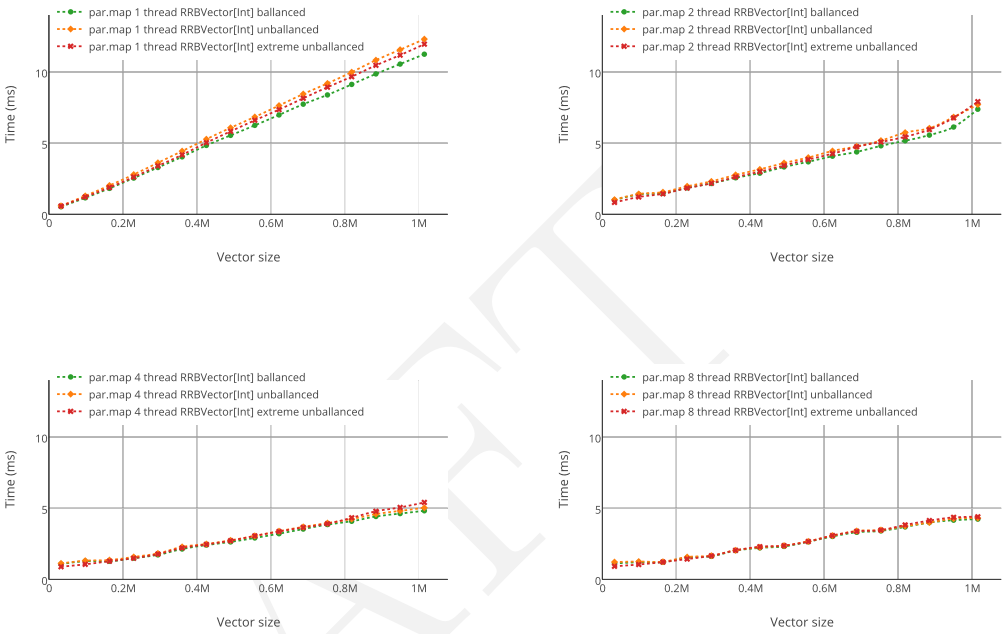


FIGURE 4.21: Benchmark on map and parallel map using the function (x=>x) to show the difference time used in the framework. This time represents the time spent in the splitters and combiners of the parallel collection.

### 4.4.9 Memory footprint

This is not really a benchmark, it is rather the characterization of the memory footprint of the vectors that where used as input in the benchmarks. The aim of this is to show that even with the additional sizes of the unbalanced nodes and additional fields, the vectors size in memory is almost the same.

Figure 4.22 shows that even for an extremely unbalanced RRB-Vector the increase in size is negligible. In fact, for balanced ones it is even possible to have a smaller footprint (a few bytes) caused by the truncation of the last branch.

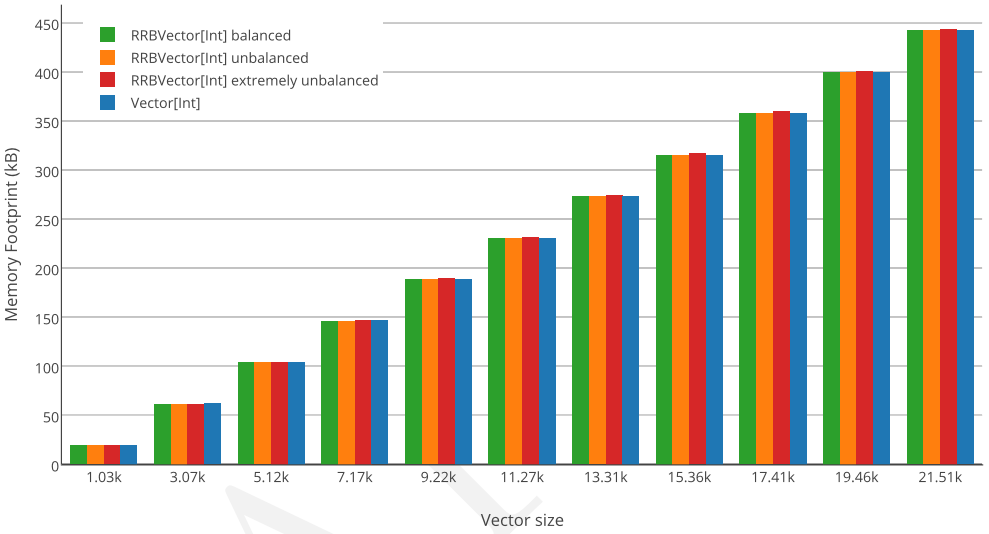


FIGURE 4.22: Memory Footprint for different vectors.

Figure 4.23 shows the amount of space that could be saved by increasing the size of the block. The difference is not significant and as such the memory footprint is not a reason to change the size of blocks.

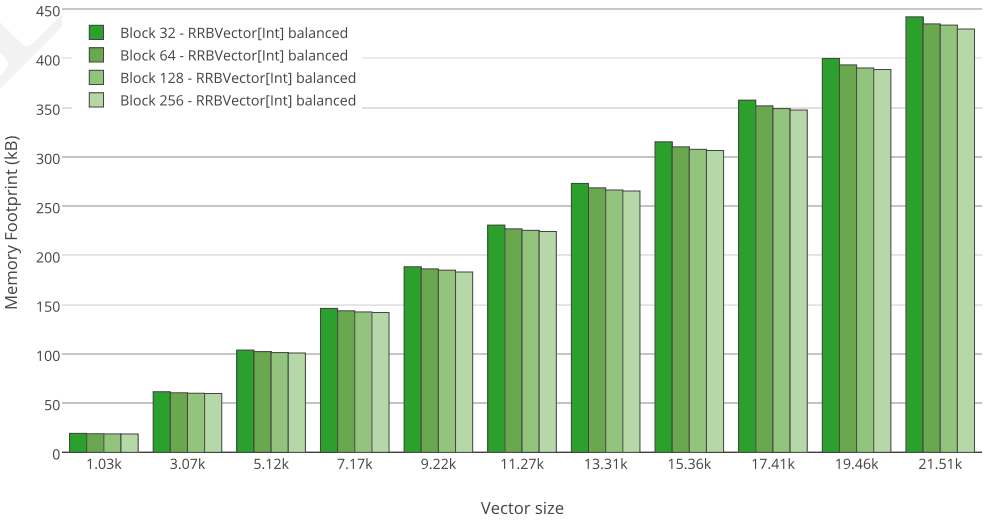


FIGURE 4.23: Memory Footprint for different block sizes.



I

# Chapter 5

# Testing

## 5.1 Black Box Testing

Tests on vector where done using the Scala Test framework. The test suite covers all core operations and some of the ones that are implemented in `IndexedSeq`. It also covers iterators, builders and parallel versions. Operations on `IndexedSeq` are crossed checked with other existing implementations.

The same test suite is used on the current `Vector` and on `RRBVector`. This way the test suite is also checked against the expected results from the current implementation. The test suite on `RRBVector` is executed on several sizes of vectors, with perfectly balanced vectors and several pseudo randomly unbalanced vectors. Each time a bug was identified, the tests where extended to include the case where it failed.

These tests where usually executed in combination with the white box tests. This way test have a larger coverage and will fail at the moment where the bug first appeared.

## 5.2 White Box Testing

White box testing is done using a set of heavy assertions on the invariants of the vector[1]. They test that the coherence of the three structure and the fields of vector. This test is done after the creation of any new vector object and after any canonicalization operation (see 3.2.4). This covers all the cases as no mutation is done in between of after those operations (assuming a non concurrent context during the test).

[1] This is implemented in the private method `assertVectorInvariant` of `RRBVector`.

The vector invariant is divided into canonical and transient, where both test the same characteristics with some differences on the focused branch. The invariant includes tests on the bounds of `depth`, `endIndex`, `focus`, `focusStart`, `focusEnd`. It traverses the whole tree structure checking the structure of each node and crosschecking sizes (from `sizes` array and expected `endIndex`). For canonical states it checks the the branch in the displays is coherent with the tree an `focus`. For transient state it checks that the focused branch is unlinked.

I

# Chapter 6

# Related and Future Work

## 6.1 Related Work

List of related subjects:

- RRB Trees and Vectors [1, 3, 4, 9, 10]
- Scala [11, 12]
- Scala Collections [3, 13–17]
- Scala Parallel Collections [4, 5, 18–20]
- Functional data structures and semi-mutable data structures and related data structures [21–28]
- Performance and Code specialization [29–32]
- JVM: Arrays, GC, JIT compiler [33–35]
- ScalaMeter [7, 36]
- Scala Test [37]
- Scala Reflection and Quasiquotes [38, 39]
- Type specialization and Miniboxing [6, 15, 40–42]

## 6.2 Future Work

**Mesure unbalance** One analysis that was left out of the scope of this project was the characterization of the vectors unbalance. There is currently no way to quantitatively measure the unbalance of on the tree node. Some ideas for this are: number of unbalanced nodes, number of balanced subtrees, average height of balanced subtrees, ...

Once this measurement exists it would be possible to conduct a real world application characterization of these vectors. And see how common the unbalanced vectors are and if they are only slightly unbalanced or extremely unbalanced. From this it would be possible to give a true expected performance for the cases where the performances differ.

**Miniboxing** As discussed in section 3.1.1, when primitive types are used in vector they become boxed. A proposed solution to this phenomenon is using Miniboxing [6, 15] on the vectors. Applying this transformation on the type parameter of the vector will remove boxing in the method calls. Additionally, this allows the use of `MbArray` on the leafs, these arrays take advantage of the additional type information that Miniboxing provides and applies it opportunistically. A prototype of this vector is already implemented in this project repository[1]. The prototype shows that the boxing is removed and that the leafs arrays are getting specialized. Tests and benchmarks specific to this implementation details are still required.

**Simplify Code** It should be possible to simplify and reduce the amount of code of the Scala implementation of vector using Scala Macros [38] to expand and optimize the code that was done manually. This was partially done on the vector implementation generators (see 4.3), where some key abstractions generate most of the expansions. This is a good basis to write such macros.

Another way to simplify the code would involve the creation of a new abstraction that would compile down to bytecode `tableswitch` instructions with fallthroughs. A switch with no `breaks` should be enough[2]. This could help a bit with performance.

**Formalization** There is still the need for formal proof on some operations. Mainly for the optimized implementations of the relaxed operations. The canonicalization operation of the vector also needs a formal proof and may be a interesting study case for immutable data structures with simple internal mutation schemes. I

[1] Project [2] under the `adding-miniboxing` branch.

[2] Look at `copyDisplays` function in the code, the `if/else` expressions could be replaced by a single `switch` statement.

# Chapter 7

# Conclusions

The new implementation of vector uses the RRB-Trees while using all the optimizations that where possible on RB-Trees. In most cases they are applied on balanced subtrees. The algorithm variant chosen for the concatenation yield better balanced subtrees for a small cost in performance, a cost that is considered irrelevant considering that the algorithm improved from linear time to constant time.

The implementation achieved the performance goals in most cases. Usually the performance degrades only when the vector is extremely unbalanced. Effective constant time (or in some cases amortized constant time from $log_{32}(n)$) was achieved for all the core operations: apply, appended, prepended, updated, take, drop, concatenated and insert. It was also showed that with relaxed trees the branch sizes of 32 are still the best choice for the performance of these operations.

Parallel vector achieved the desired parallelism on the fork-join pools using their split-combine operations. Benchmarks showed a 2.3X improvement on the data split and combination on a 4 thread pool.

# Bibliography


[1] Tiark Rompf Phil Bagwell. RRB-Trees: Efficient Immutable Vectors. Technical report, EPFL, 2011.

[2] Nicolas Stucki. Scala rrb vectors project repository. https://github.com/nicolasstucki/scala-rrb-vector, 2014.

[3] Scala 2.11 - Vector.scala. https://github.com/scala/scala/blob/394da59828b830f639d2418960052655d9dd040a/src/library/scala/collection/immutable/Vector.scala, 12 2013.

[4] Scala 2.11 - ParVector.scala. https://github.com/scala/scala/blob/f4267ccd96a9143c910c66a5b0436aaa64b7c9dc/src/library/scala/collection/parallel/immutable/ParVector.scala, 9 2013.

[5] Aleksandar Prokopec, Tiark Rompf, Phil Bagwell, and Martin Odersky. On a generic parallel collection framework, 2011.

[6] Vlad Ureche, Cristian Talau, and Martin Odersky. Miniboxing: Improving the speed to code size tradeoff in parametric polymorphism translations. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '13, pages 73–92, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2374-1. doi: 10.1145/2509136.2509537. URL http://doi.acm.org/10.1145/2509136.2509537.

[7] Aleksandar Prokopec. ScalaMeter. https://scalameter.github.io/.

[8] David P. Rodgers. Improvements in multiprocessor system design. In *Proceedings of the 12th Annual International Symposium on Computer Architecture*, ISCA '85, pages 225–231, Los Alamitos, CA, USA, 1985. IEEE Computer Society Press. ISBN 0-8186-0634-7. URL http://dl.acm.org/citation.cfm?id=327010.327215.

[9] Jean Niklas L'orange. Improving RRB-Tree Performance through Transience. Master's thesis, Norwegian University of Science and Technology, June 2014.

[10] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, Clifford Stein, et al. *Introduction to algorithms*, volume 2. MIT press Cambridge, 2001.

[11] Martin Odersky, Lex Spoon, and Bill Venners. *Programming in scala*. Artima Inc, 2008.

[12] Josh Suereth. *Scala In Depth*. Manning Publications Co., Sound View Ct. 3B Greenwich, CT 068303B Greenwich, CT 06830, 2011. URL http://www.manning.com/suereth/. This is not complete, only available in "Early Access" edition.

[13] Adriaan Moors. *Type Constructor Polymorphism for Scala: Theory and Practice (Type constructor polymorfisme voor Scala: theorie en praktijk)*. PhD thesis, Informatics Section, Department of Computer Science, Faculty of Engineering Science, May 2009. URL https://lirias.kuleuven.be/handle/1979/2642. Joosen, Wouter and Piessens, Frank (supervisors).

[14] Bruno C.d.S. Oliveira, Adriaan Moors, and Martin Odersky. Type classes as objects and implicits. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '10, pages 341–360, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0203-6. doi: 10.1145/1869459.1869489. URL http://doi.acm.org/10.1145/1869459.1869489.

[15] Aymeric Genêt, Vlad Ureche, and Martin Odersky. Improving the Performance of Scala Collections with Miniboxing. Technical report, EPFL, 2014. URL http://scala-miniboxing.org/.

[16] Martin Odersky and Adriaan Moors. Fighting bit Rot with Types (Experience Report: Scala Collections). In Ravi Kannan and K. Narayan Kumar, editors, *IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science*, volume 4 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 427–451, Dagstuhl, Germany, 2009. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. ISBN 978-3-939897-13-2. doi: http://dx.doi.org/10.4230/LIPIcs.FSTTCS.2009.2338. URL http://drops.dagstuhl.de/opus/volltexte/2009/2338.

[17] Martin Odersky. Future-Proofing Collections: From Mutable to Persistent to Parallel. In *Compiler Construction*, volume 6601 of *Lecture Notes in Computer Science*, pages 1–1. Springer-Verlag New York, Ms Ingrid Cunningham, 175 Fifth Ave, New York, Ny 10010 Usa, 2011.

[18] Doug Lea. A java fork/join framework. In *Proceedings of the ACM 2000 Conference on Java Grande*, JAVA '00, pages 36–43, New York, NY, USA, 2000. ACM. ISBN 1-58113-288-3. doi: 10.1145/337449.337465. URL http://doi.acm.org/10.1145/337449.337465.

[19] Aleksandar Prokopec. *Data Structures and Algorithms for Data-Parallel Computing in a Managed Runtime*. PhD thesis, IC, Lausanne, 2014.

[20] Aleksandar Prokopec and Martin Odersky. Near optimal work-stealing tree scheduler for highly irregular data-parallel workloads. In *Languages and Compilers for Parallel Computing*, Lecture Notes in Computer Science, pages 55–86. Springer International Publishing, 2014. ISBN 978-3-319-09966-8. doi: 10.1007/978-3-319-09967-5_4. URL http://dx.doi.org/10.1007/978-3-319-09967-5_4.

[21] Chris Okasaki. *Purely Functional Data Structures*. Cambridge University Press, New York, NY, USA, 1998. ISBN 0-521-63124-6.

[22] Ralf Hinze and Ross Paterson. Finger trees: a simple general-purpose data structure. *J. Funct. Program.*, 16(2):197–217, 2006. doi: 10.1017/S0956796805005769. URL http://dx.doi.org/10.1017/S0956796805005769.

[23] James R. Driscoll, Neil Sarnak, Daniel D. Sleator, and Robert E. Tarjan. Making data structures persistent. *J. Comput. Syst. Sci.*, 38(1):86–124, February 1989. ISSN 0022-0000. doi: 10.1016/0022-0000(89)90034-2. URL http://dx.doi.org/10.1016/0022-0000(89)90034-2.

[24] Phil Bagwell. Fast Functional Lists, Hash-Lists, Deques and Variable Length Arrays. In *Implementation of Functional Languages*, 2002.

[25] R. Bayer and E. McCreight. Organization and maintenance of large ordered indices. In *Proceedings of the 1970 ACM SIGFIDET (Now SIGMOD) Workshop on Data Description, Access and Control*, SIGFIDET '70, pages 107–141, New York, NY, USA, 1970. ACM. doi: 10.1145/1734663.1734671. URL http://doi.acm.org/10.1145/1734663.1734671.

[26] Hans-J. Boehm, Russ Atkinson, and Michael Plass. Ropes: An alternative to strings. *Software: Practice and Experience*, 25(12):1315–1330, 1995. ISSN 1097-024X. doi: 10.1002/spe.4380251203. URL http://dx.doi.org/10.1002/spe.4380251203.

[27] Phil Bagwell. Ideal hash trees. *Es Grands Champs*, 1195, 2001.

[28] Phil Bagwell. Fast And Space Efficient Trie Searches. Technical report, 2000.

[29] Tiark Rompf. *Lightweight Modular Staging and Embedded Compilers*. PhD thesis, IC, Lausanne, 2012.

[30] Hassan Chafi, Arvind K. Sujeeth, Kevin J. Brown, HyoukJoong Lee, Anand R. Atreya, and Kunle Olukotun. A domain-specific approach to heterogeneous parallelism. In *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming*, PPoPP '11, pages 35–46, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0119-0. doi: 10.1145/1941553.1941561. URL http://doi.acm.org/10.1145/1941553.1941561.

[31] Duncan Coutts, Roman Leshchinskiy, and Don Stewart. Stream fusion: From lists to streams to nothing at all. In *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming*, ICFP '07, pages 315–326, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-815-2. doi: 10.1145/1291151.1291199. URL http://doi.acm.org/10.1145/1291151.1291199.

[32] Aleksandar Prokopec, Dmitry Petrashko, and Martin Odersky. Efficient lock-free work-stealing iterators for data-parallel collections. 2015.

[33] Thomas Kotzmann, Christian Wimmer, Hanspeter Mössenböck, Thomas Rodriguez, Kenneth Russell, and David Cox. Design of the java hotspot™ client compiler for java 6. *ACM Trans. Archit. Code Optim.*, 5(1):7:1–7:32, May 2008. ISSN 1544-3566. doi: 10.1145/1369396.1370017. URL http://doi.acm.org/10.1145/1369396.1370017.

[34] Michael Paleczny, Christopher Vick, and Cliff Click. The java hotspottm server compiler. In *Proceedings of the 2001 Symposium on JavaTM Virtual Machine Research and Technology Symposium - Volume 1*, JVM'01, pages 1–1, Berkeley, CA, USA, 2001. USENIX Association. URL http://dl.acm.org/citation.cfm?id=1267847.1267848.

[35] Thomas Würthinger. Extending the graal compiler to optimize libraries. In *Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion*, OOPSLA '11, pages 41–42, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0942-4. doi: 10.1145/2048147.2048168. URL http://doi.acm.org/10.1145/2048147.2048168.

[36] Andy Georges, Dries Buytaert, and Lieven Eeckhout. Statistically rigorous java performance evaluation. In *Proceedings of the 22Nd Annual ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications*, OOPSLA '07, pages 57–76, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-786-5. doi: 10.1145/1297027.1297033. URL http://doi.acm.org/10.1145/1297027.1297033.

[37] Bill Venner, George Berger, and Chua Chee Seng. Scalatest. http://www.scalatest.org/.

[38] Eugene Burmako. Scala Macros: Let Our Powers Combine! In *4th Annual Workshop Scala 2013*, 2013. ISBN 978-1-4503-2064-1/13/07. doi: 10.1145/2489837.2489840.

[39] Denys Shabalin, Eugene Burmako, and Martin Odersky. Quasiquotes for scala. Technical Report EPFL-REPORT-185242, EPFL, 2013.

[40] Vlad Ureche, Eugene Burmako, and Martin Odersky. Late data layout: Unifying data representation transformations. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '14, pages 397–416, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2585-1. doi: 10.1145/2660193.2660197. URL http://doi.acm.org/10.1145/2660193.2660197.

[41] Nicolas Stucki and Vlad Ureche. Bridging islands of specialized code using macros and reified types. In *Proceedings of the 4th Workshop on Scala*, SCALA '13, pages 10:1–10:4, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2064-1. doi: 10.1145/2489837.2489847. URL http://doi.acm.org/10.1145/2489837.2489847.

[42] Iulian Dragos. *Compiling Scala for Performance*. PhD thesis, IC, Lausanne, 2010.