École Polytechnique Fédérale de Lausanne

Master Thesis

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

Author:
Nicolas Stucki

Supervisor:
Vlad Ureche

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

in the

LAMP
Computer Science

January 2015

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

Abstract

School of Computer and Communications
Computer Science

Master in Computer Science

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

by Nicolas Stucki

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

Contents

List of Figures

List of Tables

Abbreviations

JVM	**J**ava **V**irtual **M**achine
JIT	**J**ust **I**n **T**ime
LHS	**L**eft **H**and **S**ide
RHS	**R**igth **H**and **S**ide
RB	**R**adix **B**alanced
RRB	**R**elaxed **R**adix **B**alanced

I

Chapter 1

Introduction

TODO: What are immutable vectors

TODO: Why vector is important

TODO: Why changes are needed I

Chapter 2

Vector Structure and Operations

2.1 Radix Balanced Vectors

The current immutable vector implementation[1] of the Scala Collections is wrapper around an RB-Tree. The vector implements the methods of `IndexedSeq` based on a set of core efficient operations on RB-Trees.

2.1.1 Tree structure

The RB-Tree structure is shallow densely filled tree where elements are located only in the leafs and on the left side. Each node of the three has the same amount of children and the leaf have as many elements. Currently this number is 32, but could be any power of 2 to be able to have efficient radix based implementations (see 3.1.2). Figure 2.1 shows this structure for m children on each node.

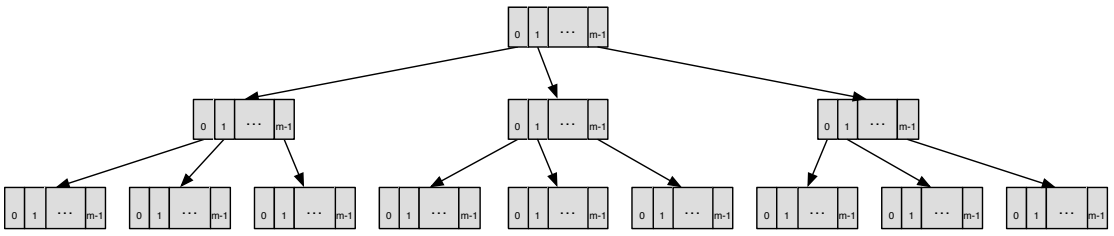

FIGURE 2.1: Radix Balanced Tree Structure

All nodes are represented with immutable[1] `Array[AnyRef]`, as this is the most compact representation for the structure and will help performance by the use of JVM primitive operations (see 4.1). Because branches and leafs are represented the same way, the vector also need to know the hight of the RB-Tree. It is kept in the `depth` field. This height will alway be upper bounded by $log_{32}(size - 1) + 1$ for nodes of 32 branches. Therefore 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 exude the maximum index representable with `Int`.

To have trees that will contain an amount of elements that does not match a full tree we fill first the elements from the left. The branches on the right that do not contain any element in any of it's leafs are not allocated (left as empty references or truncated arrays). 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 never access elements that are outside of a the trees right bound, 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).

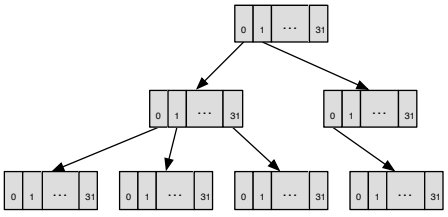

FIGURE 2.2: Radix Balanced Tree Structure with nodes of size 32 filled with 1056 elements.

Additionally, 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. The tree is left empty on the left of the first element. Arrays on the nodes of the left branch must be allocated wit the full size. As such, the tree structure is logically still full on the left but element are never access due to a shift if the index using the `startIndex`. Therefore radix indexing is still possible.

[1]Immutable arrays are just mutable arrays that do not get updated after initialisation.

2.1.2 Operations

The immutable `Vector` [1] is a subtype of `IndexedSeq` in the current Scala collections, as such it implements all operations defined on it. Most operations are implemented using a set of core operations that can be implemented efficiently on RB-Trees, those operations are: `apply`, `updated`, `append`, `prepend`, `drop` and `take`.

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

In this section the operations will be presented on a high level[2] and without optimisation. These reflect the worst case scenario or the base implementation that is used before adding optimizations (see chapter 3). To improve performance of some operations, displays are used on branches of the RB-Tree (see section 3.2). To improve the complexity of operations to amortized constant time (instead of effective constant time[3]) for some operation sequences the vectors are augmented with transient states.

2.1.2.1 Apply

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

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

The base implementation on RB-Trees of this operation requires a simple traversal from the root to the leaf containing the element. Where the path taken is defined by the index

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

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

of the element and extracted using some efficient bitwise operations (see section 3.1.2). With this traversal of the tree, the complexity of the `apply` operation is $O(log_{32}(n))$. This complexity can be reduced for some subset of elements by adding displays.

2.1.2.2 Updated

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

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

```
def updated(index: Int, elem: A) = {
  def updatedNode(node: Array[AnyRef], depth: Int) = {
    val indexInNode = // compute index
    val 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 brach. This operation can be improved to have amortized constant time updates for local updates using displays with transient states. This way the parent nodes are only updated lazily when absolutely necessary. For example, if some leaf has all it's elements updated from left to right, the leaf will be copied as many times as there are updates, but the parent of that leaf does not change during those operations.

2.1.2.3 Extentions

The main operations to extend an immutable `Vector` are `append` and `prepend` a single element. Those operations are considered the main ones because they have efficient

implementations on RB-Trees. Other operations like `concatenated` are also present, but usually avoided because of performance.

Append To append an element on the tree there are two main cases, the last leaf of the tree is not full or it is full. If the last leaf is not full the element is inserted at the end of it and all nodes of the last branch are updated. Or, 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 a 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 append(node: Array[AnyRef], depth: Int) = {
    if (depth == 1)
      copyLeafAndAppend(node, elem)
    else if (!isTreeFull(node.last, depth-1))
      copyAndUpdateLast(node, append(node.last, depth-1))
    else
      copyBranchAndAppend(node, newBranch(depth-1))
  }
  def createNewBranch(depth: Int): Array[AnyRef] = {
    if (depth == 1) Array(elem)
    else Array(newBranch(depth-1))
  }
  if(!isTreeFull(root, depth)))
    new Vector(append(root, depth), depth, ...)
  else
    new Vector(Array(root, newBranch(depth)), depth+1, ...)
}
```

Where 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 $O(log_{32}(n))$. This complexity can be amortized to constant time using displays with transient states for consecutive append operations on the vector (see section 3.2.2).

Prepend The key to be able to prepend elements to the tree is the additional `startIndex` that the vector keeps. 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. Follow the branch of the start index minus one from the root of the tree. If it reaches the a leaf, prepend the element to that leaf. If it 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: Array[AnyRef], 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): Array[AnyRef] = {
    val newNode = new Array[AnyRef](32)
    newNode(31) =
      if (depth == 1) elem
      else newBranch(depth-1)
    newNode
  }
  if (startIndex==0) {
    new Vector(Array(newBranch(depth), root), depth+1, ...)
  } else {
    new Vector(prepended(root, depth), depth, ...)
  }
}
```

Due to the update of all nodes in the first branch, the complexity of this operation is $O(log_{32}(n))$. This complexity can be amortized to constant time using displays with transient states for consecutive prepend operations on the vector (see section 3.2.2).

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 the one with the builder because it 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. This last one can be amortised to $min(n1, n2)$ using displays and transient states.

2.1.2.4 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 whole branch is not necessarily copied, if the resulting tree is shallower the nodes on the top that would have only one child are removed. 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.

2.2 Parallel Vectors

Parallel vector or `ParVector` are just 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. The only additional implementation requirements are a `Splitter` and `Combiner` that are used to manage the execution distribution of work.

`ParVector` has performance drawbacks on the `Combiner` because this one requires an efficient concatenation implementation. To avoid excessive overhead on concatenation the current implementation does it lazily, which comes with a second drawback on loss of parallelism of the combine operation.

2.2.1 Splitter (Iterator)

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

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

2.2.2 Combiner (Builder)

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

The current implementation of the immutable parallel vector [2] use the lazy approach because of it's inefficient concatenation operation. One of the consequences of this is that the parallel operations will always be bounded by this sequential combination of elements, which can be beaten by the sequential version in many cases.

2.3 Relaxed Radix Balanced Vectors

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

In an unbalanced node, the sizes are kept in an `Array[Int]` 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 the information that was usually needed on radix operations is now in the sizes.

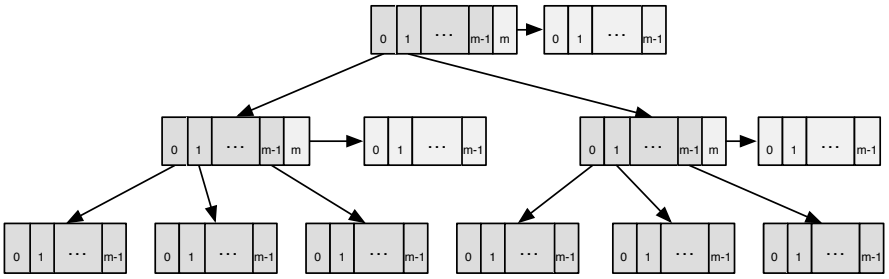

FIGURE 2.3: Radix Balanced Tree structure

In the RRB-Tree paper [3] 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 homogenous 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[4]. 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 unbalances nodes.

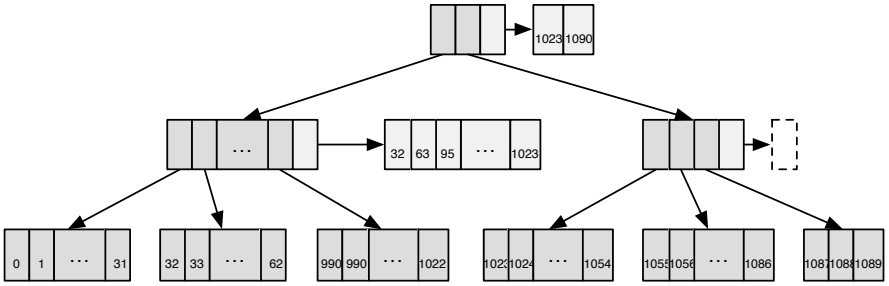

FIGURE 2.4: Concrete example of an RRB-Tree that contains 1090 elements.

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

2.3.2 Relaxing the Operations

Form most operations the implementation is relaxed using the same technique. It consist in using a generalised 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.

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 generalisation 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.

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

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 unbalances. 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 additionaly copy the sizes ad append to it a new size with value equal the the old 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 use on RRB-Vectors is the one proposed in the RRB-Trees paper [3]. 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 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 [3]. 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 2.5, 2.6, 2.7 and 2.8 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.

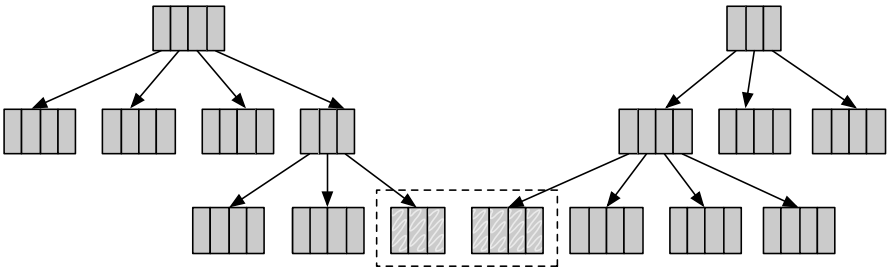

FIGURE 2.5: Concatenation example with blocks of size 4: Rebalancing level 0

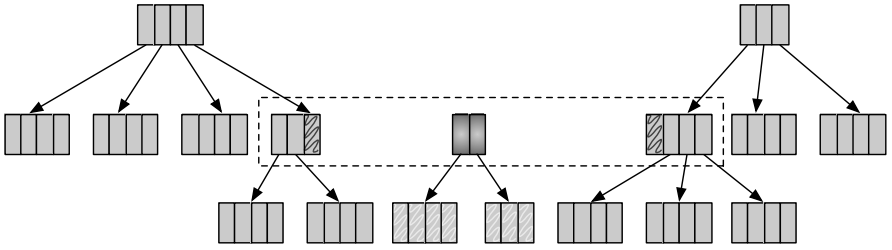

FIGURE 2.6: Concatenation example with blocks of size 4: Rebalancing level 1

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

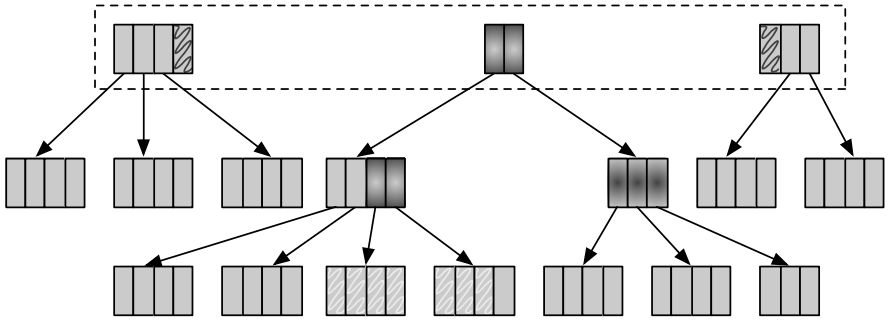

FIGURE 2.7: Concatenation example with blocks of size 4: Rebalancing level 2

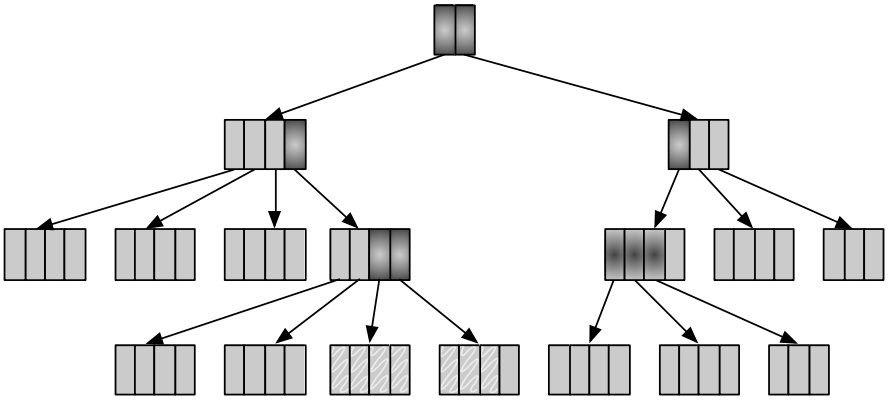

FIGURE 2.8: Concatenation example with blocks of size 4: Rebalancing level 3

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[5] 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 optimised 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

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

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
}
```

As this operation is new in the context of a vector and no real world use cases exist, this simple implementation is used. It would be is possible to optimise for localised inserts using displays and transient states.

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 the `startIndex` becomes zero when a node is not full the unbalanced nodes are used.

Both these operation will most of the time 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 alway on subindex 0. As know 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 prepend, 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: Array[AnyRef], 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): Array[AnyRef] = {
    val newNode = new Array[AnyRef](2)
    newNode(0) = if (depth==1) elem else newBranch(depth-1)
    newNode
  }
  prepended(root, depth) match {
    case Some(newRoot) => new Vector(newRoot, depth, ...)
    case None =>
      Vector(copyAndPrepend(root, newBranch(depth)),
             depth+1, ...)
  }
}
```

Sizes updated in 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 $O(log_{32}(n))$. This complexity can be amortized to constant time using displays with transient states for consecutive prepend operations on the vector (see section 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 the sizes adjusted. The sizes of the nodes get truncated at the same index and 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 numbers 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.

2.3.5.1 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 splinter 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 elements is identified, the 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 blocks are full, there is no node that requires shifting elements and therefore new block creation can be avoided.

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 they can't run in parallel on different thread of the thread pool.

I

Chapter 3

Optimizations

3.1 Where is time spent?

3.1.1 Arrays

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

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

3.1.2 Computing indices

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

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

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

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

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

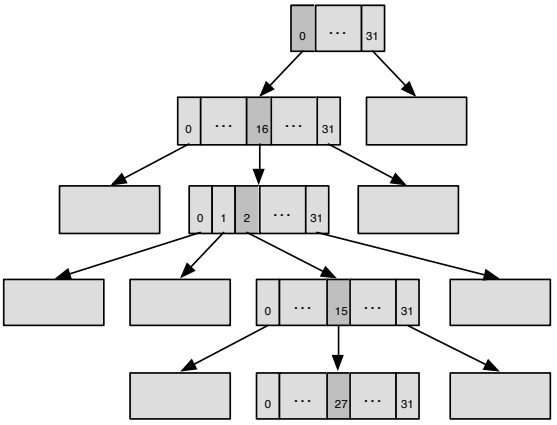

FIGURE 3.1: Accessing element at index 526843 in a tree of depth 5. Empty nodes represent collapses subtrees.

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

```
def getSubIndex(sizes: Array[Int], indexInTree: Int): Int = {
```

```
  var is = 0
  while (sizes(is) <= indexInTree)
    is += 1
  is
}
```

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

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

3.1.3 Abstractions

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 such a code. In this section we'll focus on how the actual implementation optimises 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 common 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[1].

The base mechanisms to specialize are: specialisation 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 specialisation 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`[2] 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 arithmetics cancel many 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 setup further with the `head` method that is usually implemented as `apply(0)`, using the same specialization the code becomes:

```
def head(): A = {
```

[1] This is specialisation on values not on types

[2] All `match` expressions used in this specialization get compiled to `tableswitch` bytecode instruction.

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

Removing the need for any additional arithmetic operation. The actual implementation differs from this one because it also aims to take advantage of displays (see 3.2) to avoid starting from the root in some cases.

In relaxed radix operation 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[3], they are allocated by default and nulled if the tree if shallower. The highest non null display is and replaces the root field. All displays bellow the root are never null. This implies that the vector will always be focused on some branch.

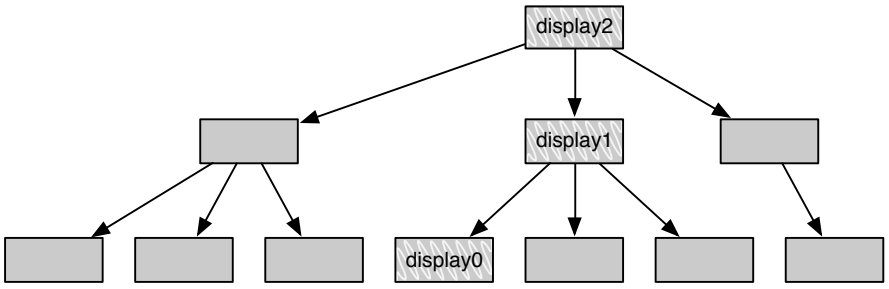

FIGURE 3.2: Displays

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

To follow the simple implementations scheme of immutable objects in concurrent contexts, the focus is also immutable. Therefore each vector object will have a single focused branch during its existence[4]. Each method that creates a new vector must decide which focus to set.

3.2.1 As cache

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

To know the which is the level of the lowest common node in a vector of block size 2^m (for some consistent m), only the `focus` index and the index being fetched are

[3] As in practice, only the 30 bits of the index are used.
[4] The display focus may change during the initialisation of the object as optimisation of some methods

needed. The operation $index \veebar focus$ will return a number is bounded to the maximum number of elements in a tree of that level. The actual level can be extracted with some if statements. This operation bounded by the same number of operations that will be needed to traverse the tree back down through the new branch.

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

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

3.2.2 For transient states

Transient states is the key optimisation to get append, prepend and update to amortized constant time. It consists in decoupling the tree by creating an equivalent tree that does not contain the edges on the current focused branch. The information missing in the edges of the tree is represented and can be reconstructed from the displays. In the current version of the collections vector [1] this state is identified by the `dirty` flag.

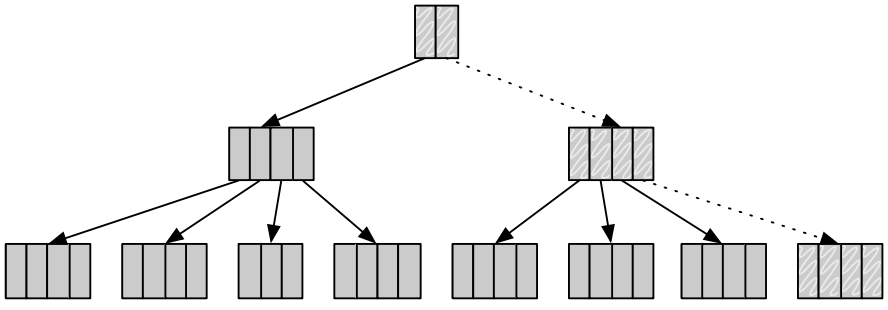

FIGURE 3.3: Transient Tree with current focus displays marked in white and striped nulled edges.

Without transient states when some update is done on a leaf, all the branch must be updated. On the other hand, if the state is transient, it is possible to update only the subtree affected by the change. In the case of updates on the same leaf, only the leaf

must be updated. When appending or prepending, $\frac{31}{32}$ operations must only update the leaf, then $\frac{31}{1024}$ need to update two levels of the tree and so on. These operations will thus be amortized to constant ($\sum_{k=1}^{\infty} \frac{k*31}{32^k} = \frac{32}{31}$ block updates per operation) time if they are executed in succession.

There is a cost associated to the transformation from 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 abel to compute the lowest common node, and there is a loss in performance due to increased memory accesses.

To still take advantage of efficient operations on balanced trees, the display is relaxed to be focused on a branch of some balanced subtree[5]. 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[6]. The operations that can take advantage of the the efficient display operations will check if the index is in the subtree index range and invoke the efficient operation. If not, it will invoke the relaxed version of the operation, that starts from the root of the tree.

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

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

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

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

```
    getElementFromRoot(index)
  else
    throw new IndexOutOfBoundsException(index)
}
```

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

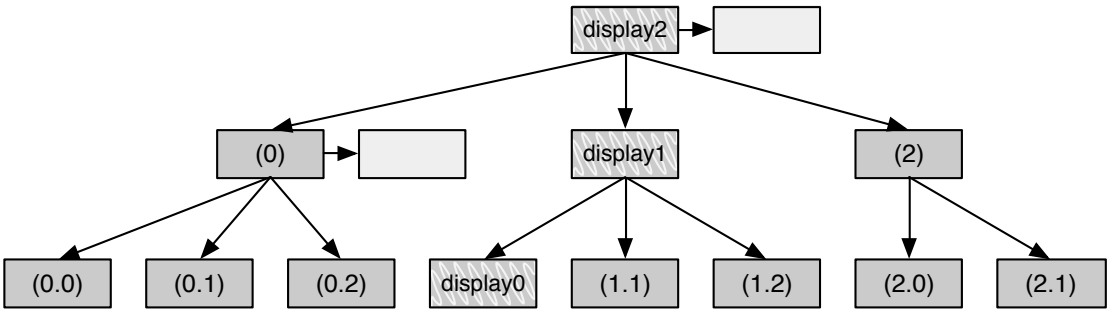

FIGURE 3.4: Relaxed Radix Balanced Tree with a focus on a balanced subtree rooted of `display1`. Light grey boxes represent unbalanced nodes sizes.

3.2.4 Vector Canonicalization

TODO: Explain

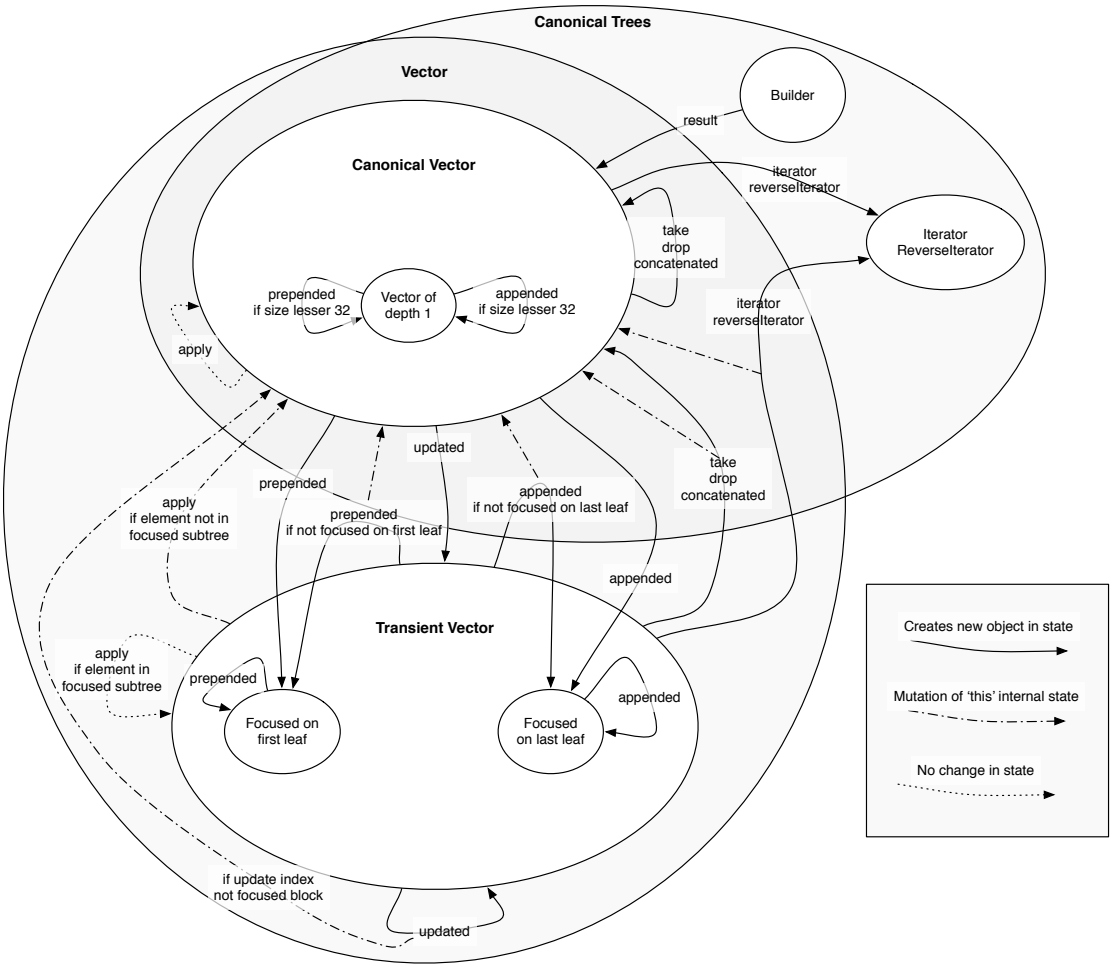
Canonical Trees
Vector
Builder
result
Canonical Vector
iterator
reverseIterator
Iterator
ReverseIterator
take
drop
concatenated
prepended
if size lesser 32
Vector of
depth 1
appended
if size lesser 32
iterator
reverseIterator
apply
updated
take
drop
concatenated
prepended
apply
if element not in
focused subtree
prepended
if not focused on first leaf
appended
if not focused on last leaf
appended
Transient Vector
apply
if element in
focused subtree
prepended
Focused on
first leaf
Focused
on last leaf
appended
if update index
not focused block
updated
Creates new object in state
Mutation of 'this' internal state
No change in state

FIGURE 3.5

3.3 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 charge on a node, nodes are allocated one and field 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 every all elements (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.

3.4 Iterator

The default implementation for iterator in `IndexedSeq` is by iterating on the indices and for each one use the `apply` method on it. 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 tree traversal of the underlying RB-Tree amortises the computational complexity to $O(n)$. The iterator can use the display optimisation tho 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[7].

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

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

Chapter 4

Performance

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 optimisations and heuristics. The code of the vectors tries to gains 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. 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 optimised 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 optimisation 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 optimisation 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 optimisation that can be done if inline manual, 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 optimisation possibilities.

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

4.2 Measuring performance

ScalaMeter framework[4] is used to measure performance of operations on different implementations of vector. This framework was used to spot 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-Vector 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] that is compared 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 the differences in performances of the operations and identify sizes for which permanence degrades due to loss of cache locality. This is the only number in the name of the artefact.

Both version of the RRB-Vector are to generate vectors. This 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 5.1.2). 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 leas one of the artefact and a fixed bug fixes it 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 RRB-Trees. 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 see the amortised access time of elements. Amortising the displays optimisation 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 sightly 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).

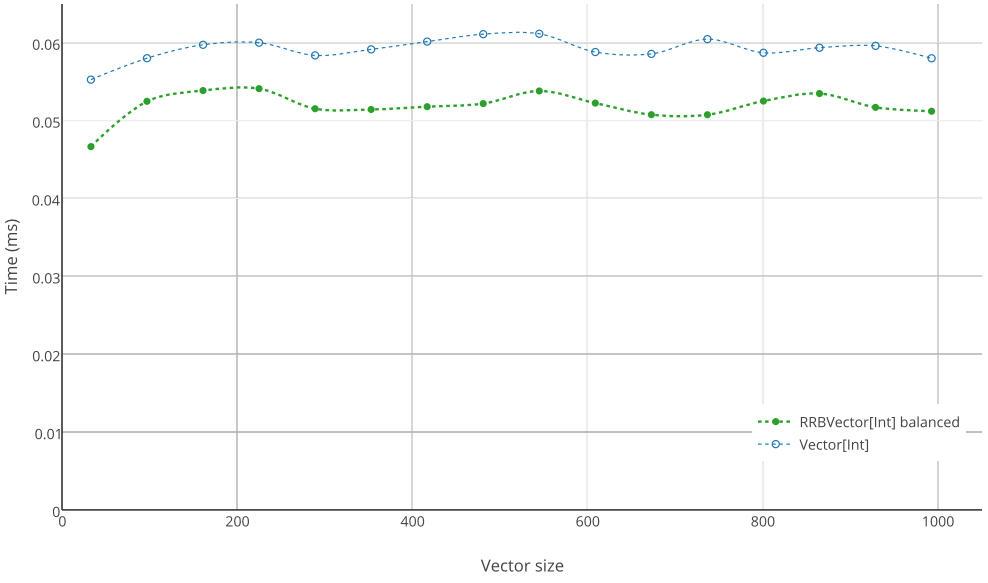

FIGURE 4.1: Time to execute 10k apply operations on sequential indices on vectors of height 2.

The performance improvements where achieved by analysing 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 faster.

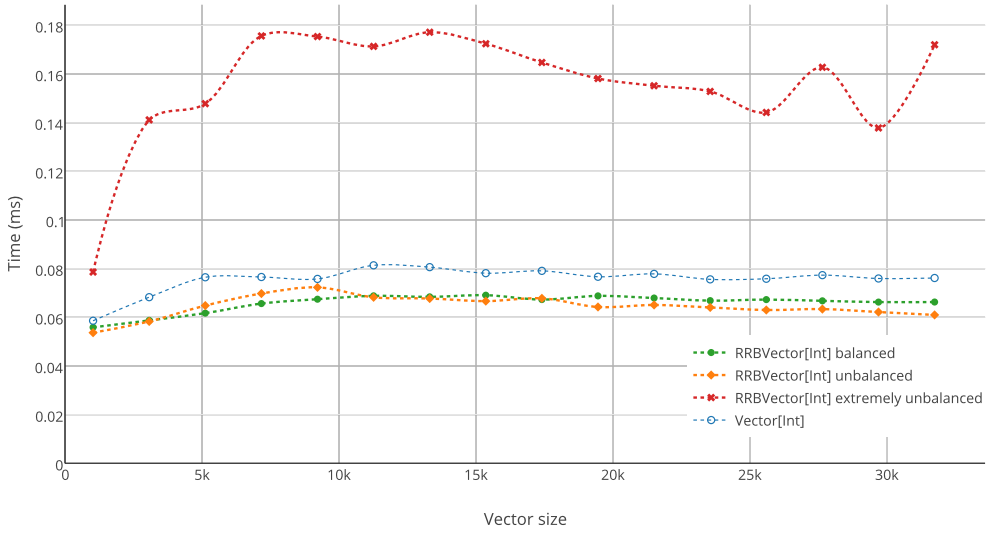

FIGURE 4.2: Time to execute 10k apply operations on sequential indices on vectors of height 3.

Figure 4.3 show 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.

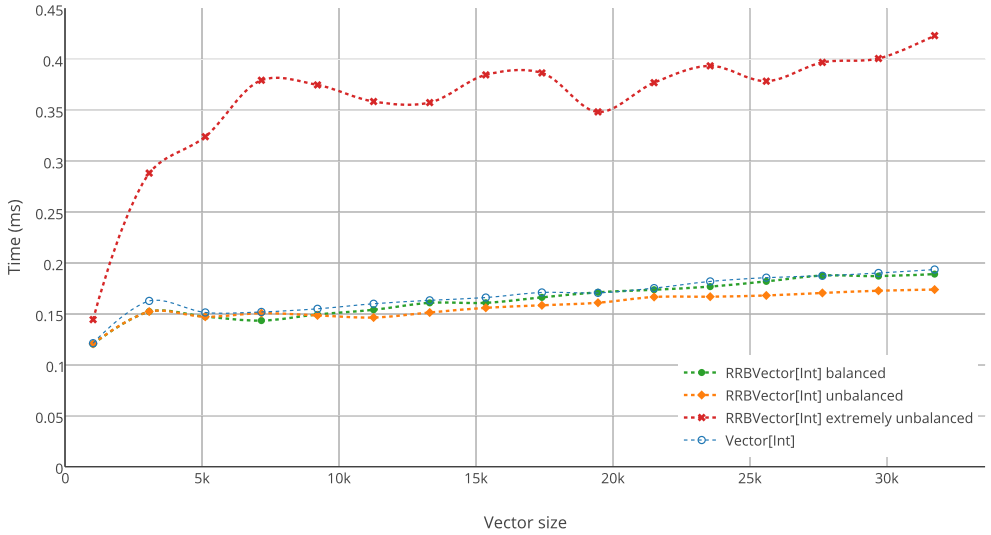

FIGURE 4.3: Time to execute 10k apply operations on random indices on vectors of height 3.

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 nocive to the performance of the apply method. Increasing the sizes of the arrays will improve the performance of the apply method.

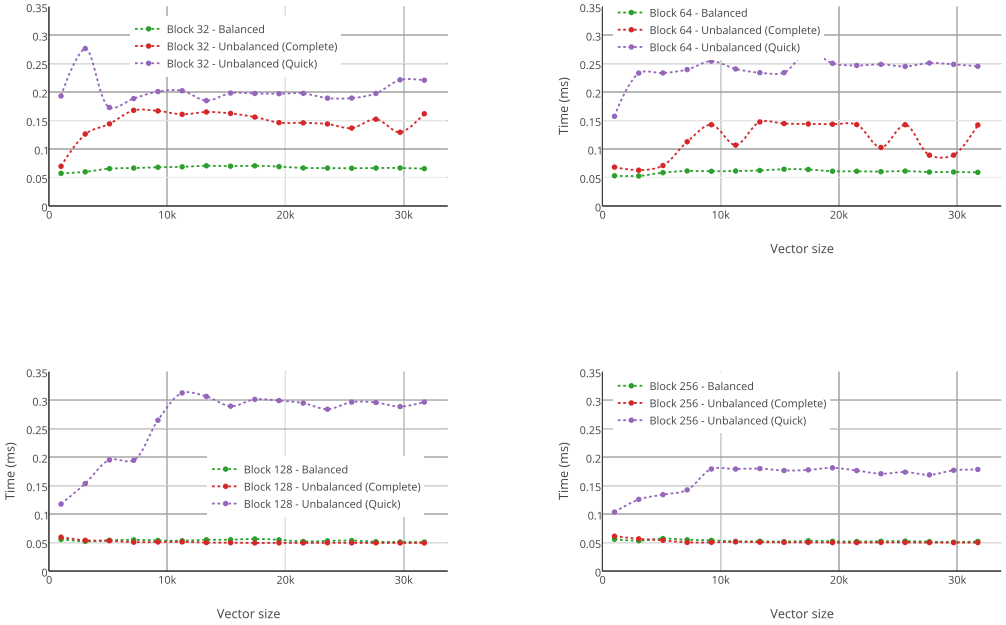

FIGURE 4.4: Time to execute 10k apply operations on sequential indices. Comparing performances for different block sizes and different implementation of the concatenation inner branch rebalancing (Complete/Quick).

4.4.2 Concatenation

TODO

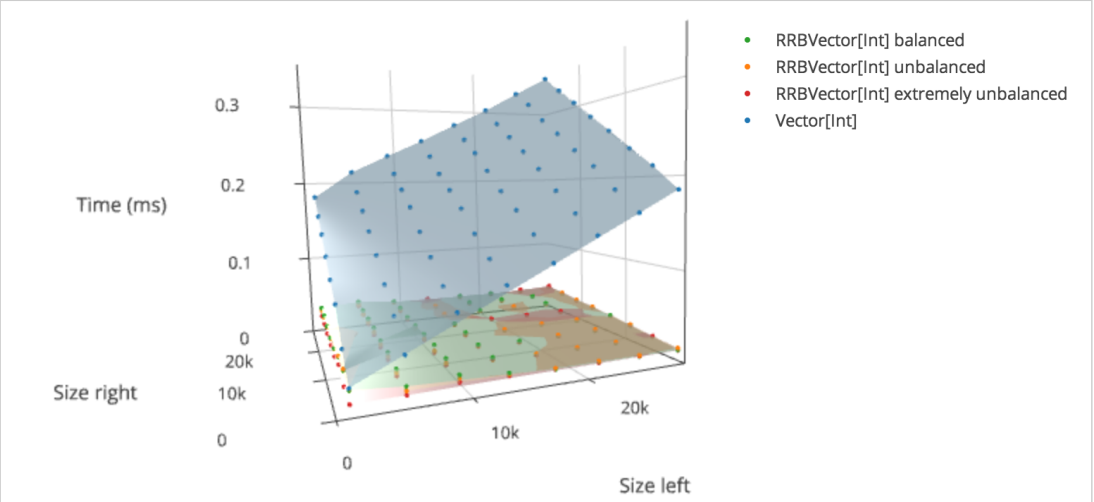

FIGURE 4.5: Execution time for a concatenation operation on two vectors. In theory (and in practice) Vector concatenation is $O(left + right)$ and the rrbVector concatenation operation is $O(log_{32}(left + right))$.

4.4.3 Append

The `append` benchmarks aim to see the amortised time taken to appending 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.

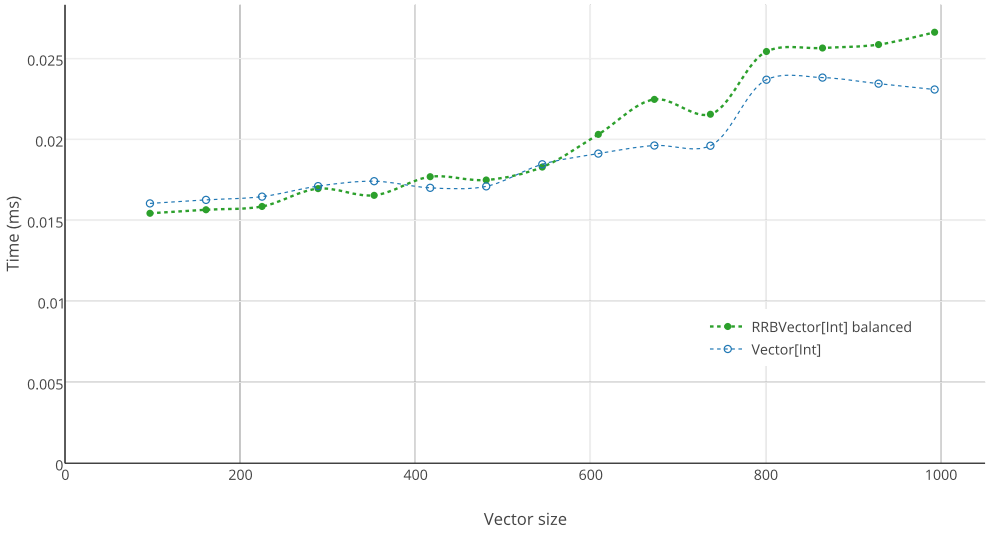

FIGURE 4.6: Time to execute 256 append operations on vectors of height 2. This shows the amortized cost of the append operation.

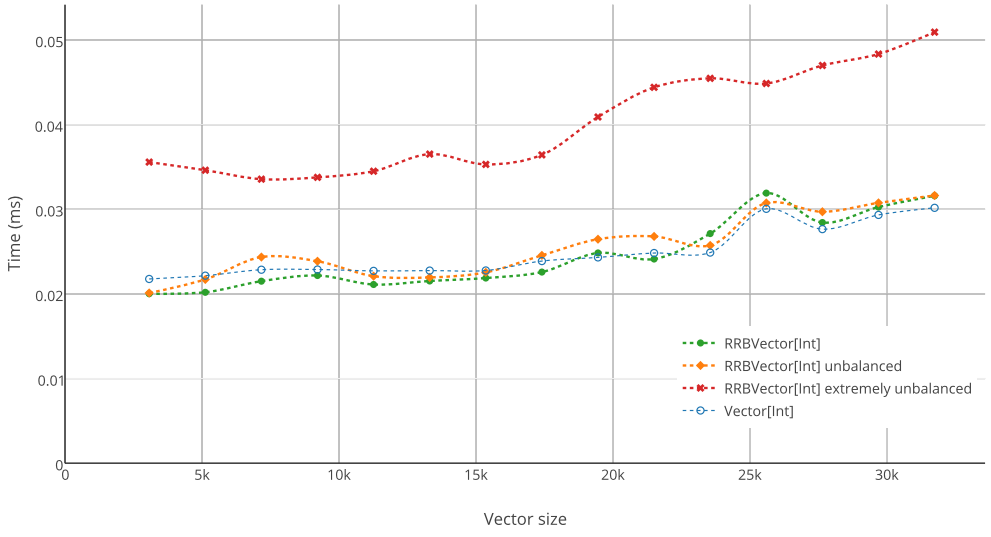

FIGURE 4.7: Time to execute 256 append operations on vectors of height 3. This shows the amortized cost of the append operation.

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 nocive 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.

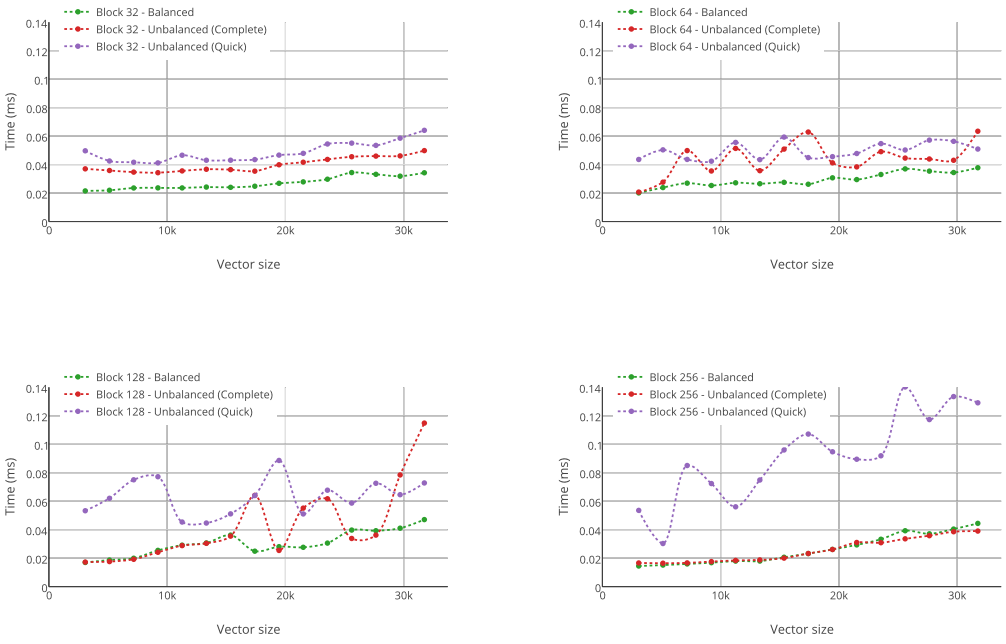

FIGURE 4.8: Time to execute 256 append operations. This shows the amortized cost of the append operation. Comparing performances for different block sizes and different implementation of the concatenation inner branch rebalancing (Complete/Quick).

4.4.4 Prepend

The `prepend` benchmarks aim to see the amortised 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 there it is possible to find space to perpend 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.

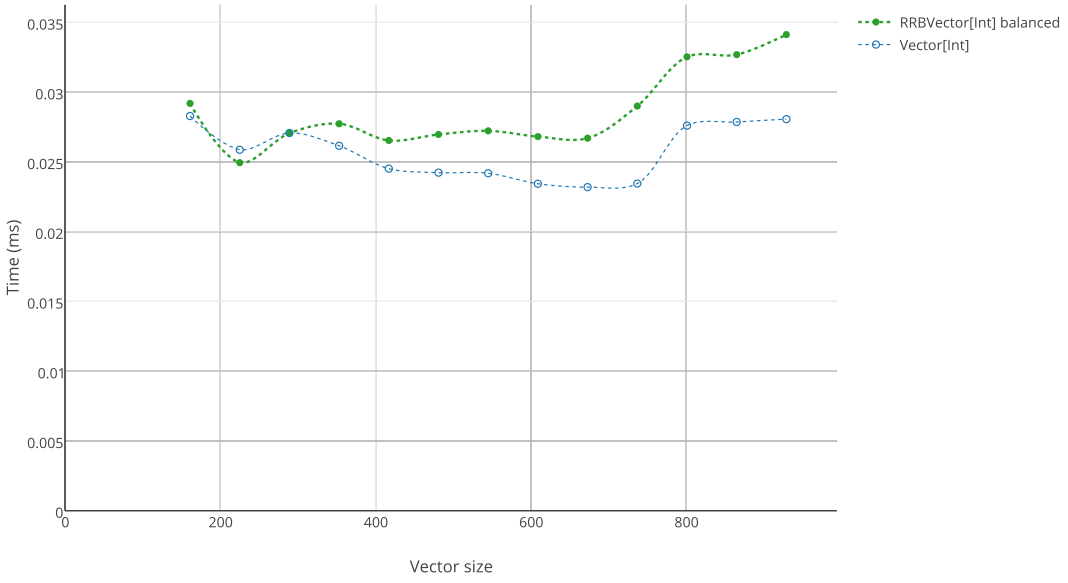

FIGURE 4.9: Time to execute 256 prepend operations on vectors of height 2. This shows the amortized cost of the prepend operation.

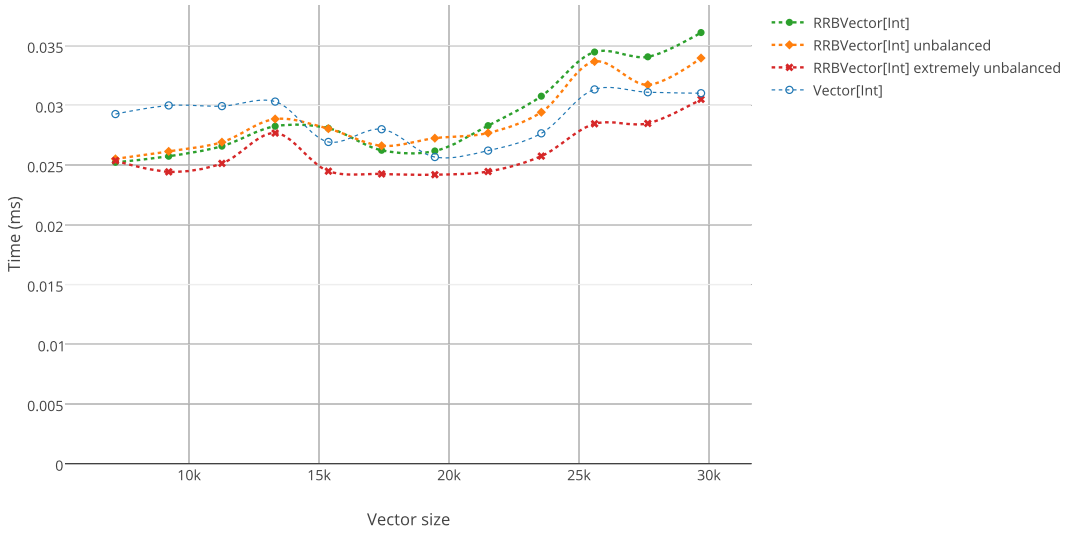

FIGURE 4.10: Time to execute 256 prepend operations on vectors of height 3. This shows the amortized cost of the prepend operation.

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 cacheline 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.

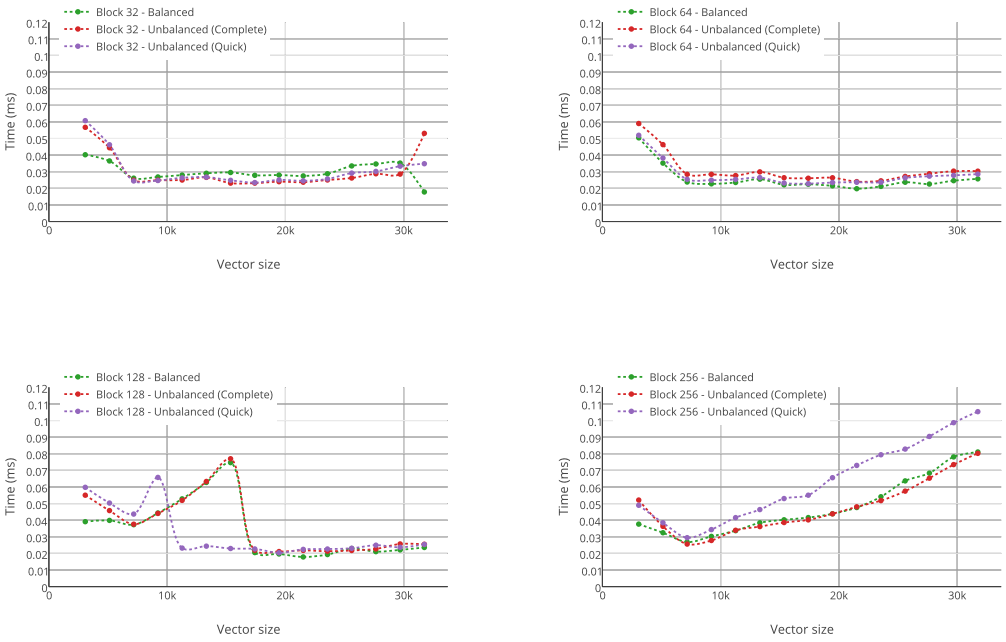

FIGURE 4.11: Time to execute 256 prepend operations. This shows the amortized cost of the append operation. Comparing performances for different block sizes and different implementation of the concatenation inner branch rebalancing (Complete/Quick).

4.4.5 Splits

TODO

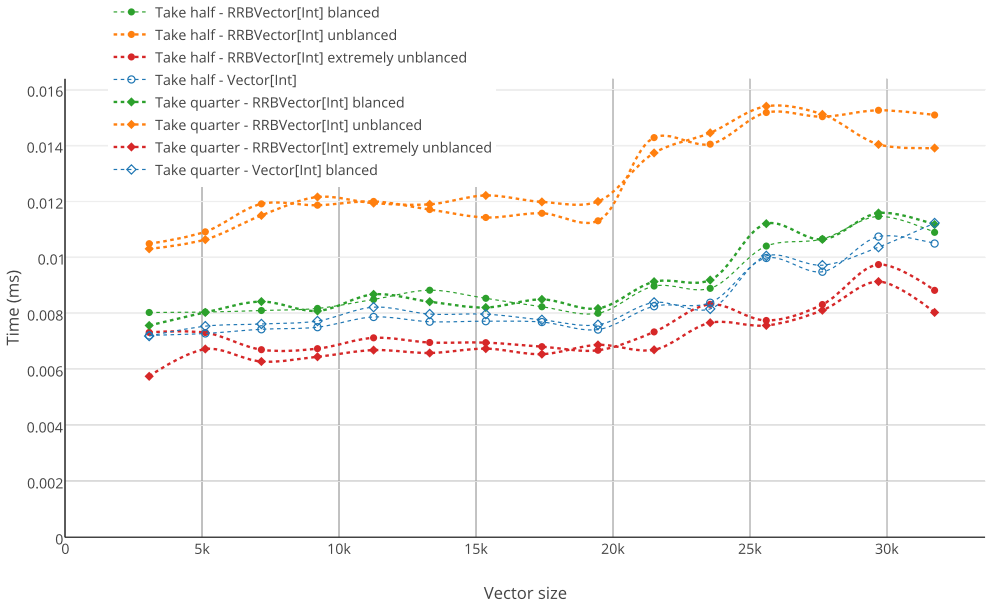

FIGURE 4.12: Execution time of take.

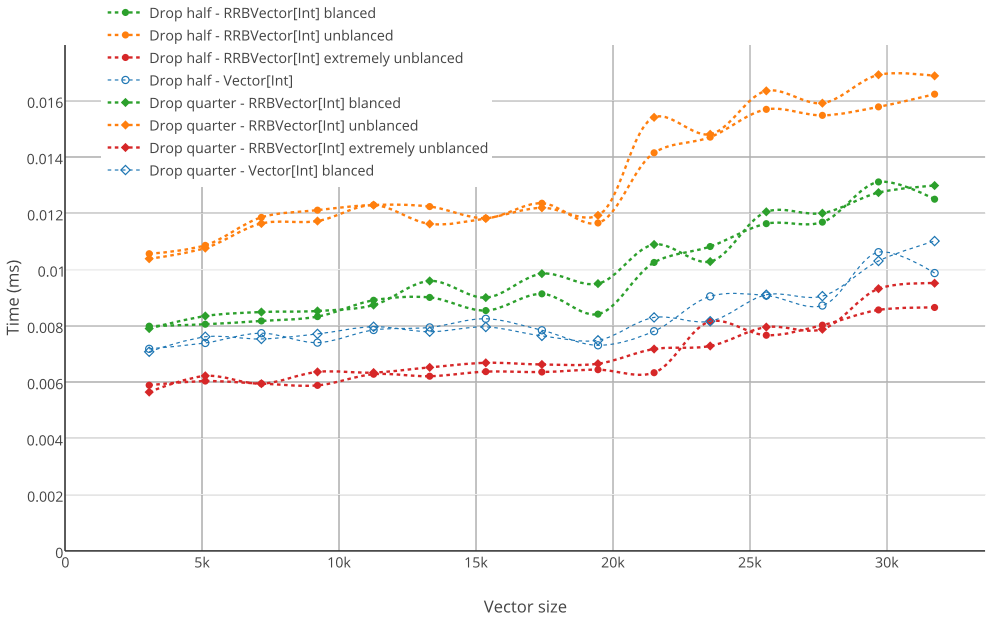

FIGURE 4.13: Execution time of drop.

4.4.6 Iterator

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

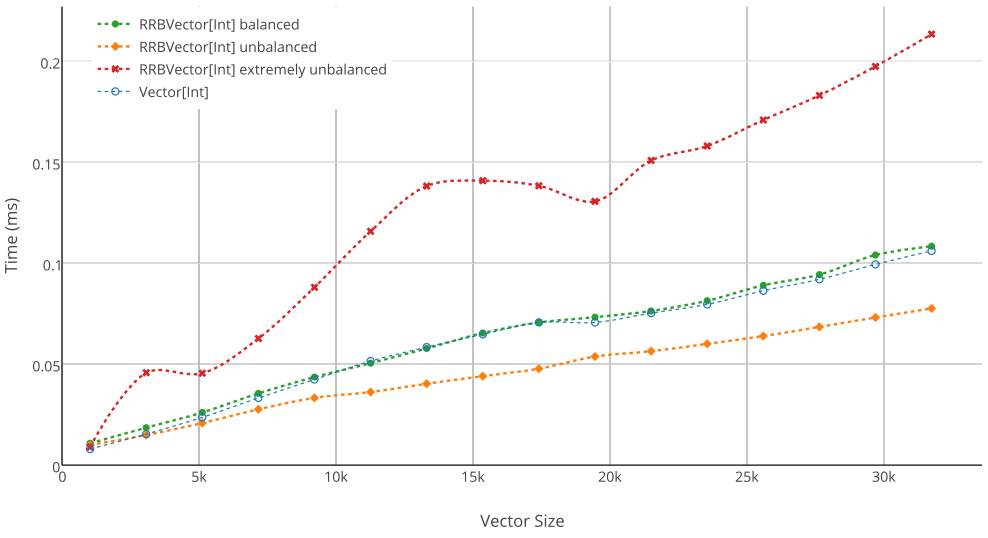

FIGURE 4.14: Excecution time to iterate through all the elements of the vector.

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 optimiser 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 initialisation time and might affect smaller vectors.

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

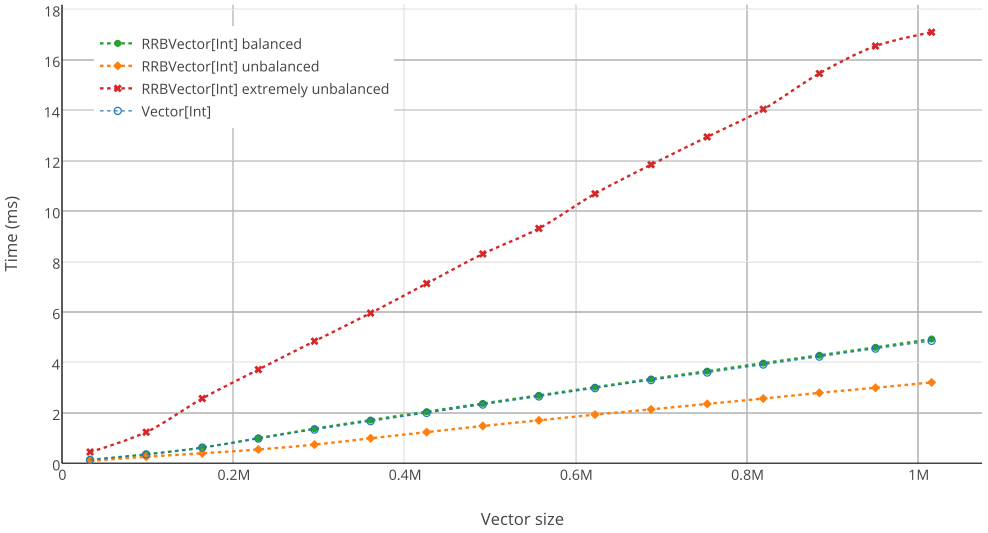

FIGURE 4.15: Excecution time to iterate through all the elements of the vector.

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.

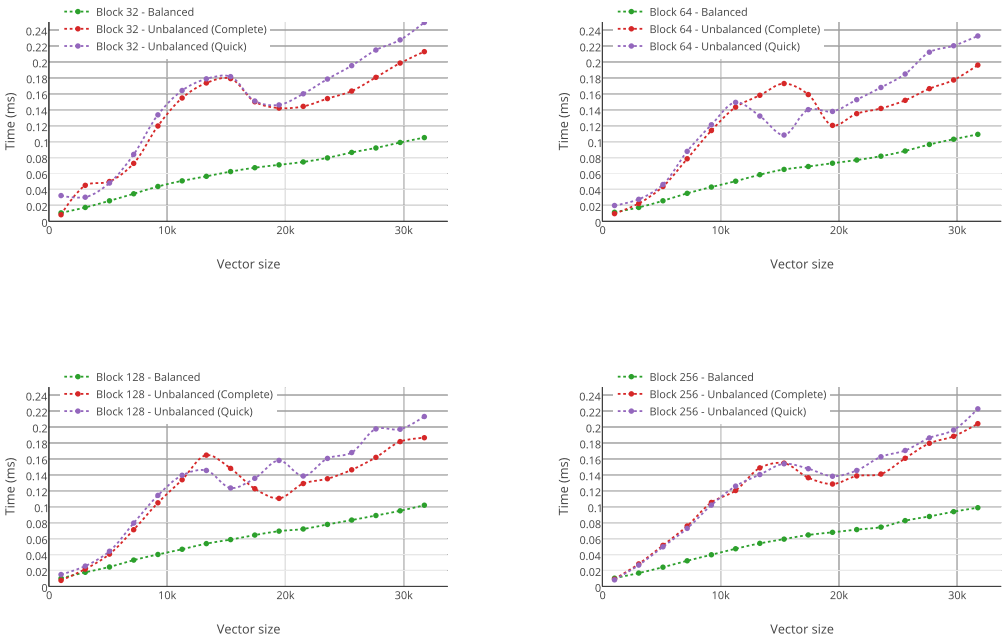

FIGURE 4.16: Excecution time to iterate through all the elements of the vector. Comparing performances for different block sizes and different implementation of the concatenation inner branch rebalancing (Complete/Quick).

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.

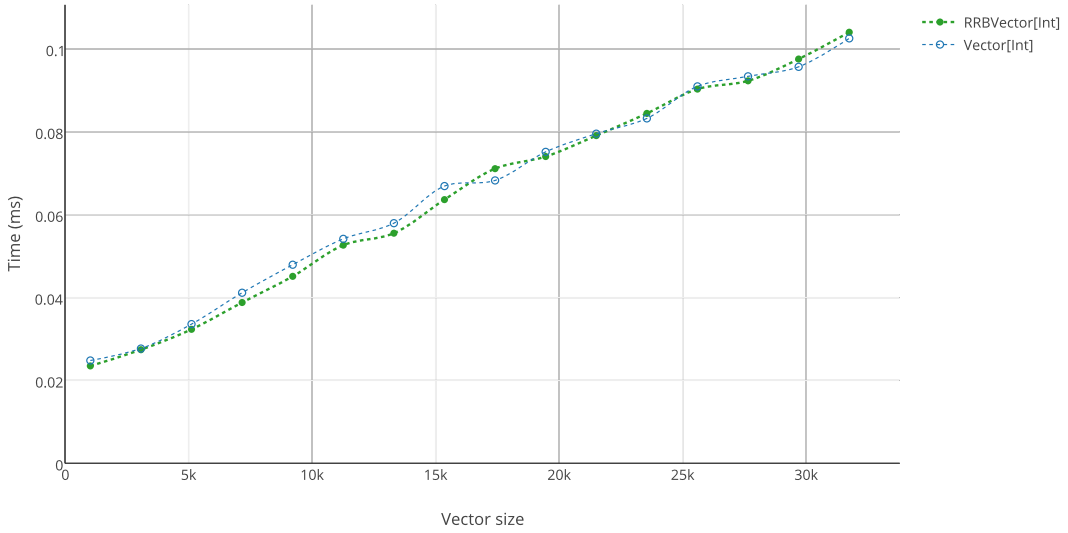

FIGURE 4.17: Execution time to build a vector of a given size.

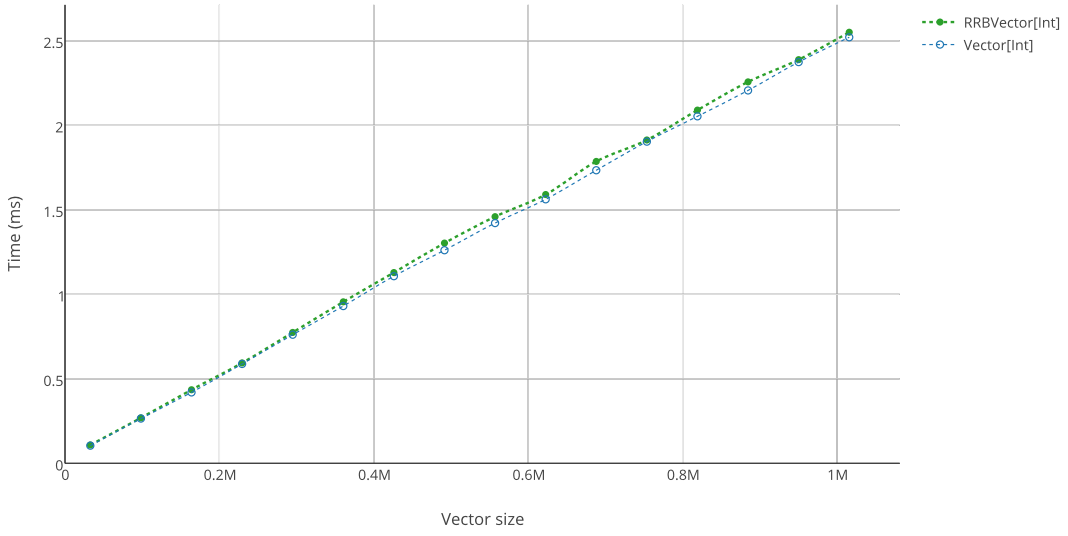

FIGURE 4.18: Execution time to build a vector of a given size.

Increasing the sizes of the blocks will reduce the amounts of blocks that need to be allocated during the building. Larger block will also take slightly longer to allocate. Figure 4.19 shows that the difference is not really big, but with a sweet spot on blocks of size 64 for builders.

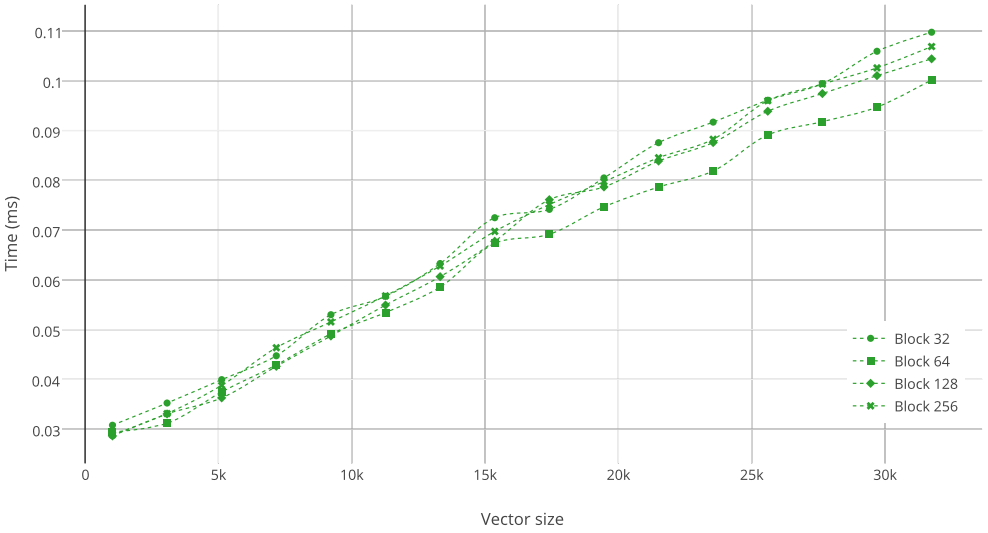

FIGURE 4.19: Execution time to build a vector of a given size. Comparing performances for different block sizes.

4.4.8 Parallel split-combine

TODO

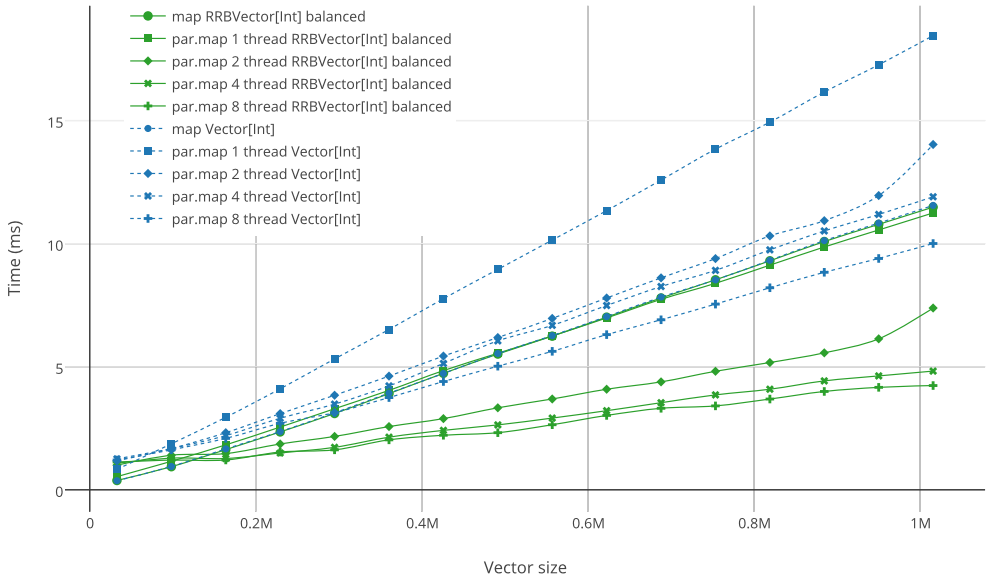

FIGURE 4.20: Benchmark on map and parallel map using the function (x=>x) to show the difference time used in the framework. This time represents the time spent in the splitters and combiners of the parallel collection (iterator and builder for the sequential version).

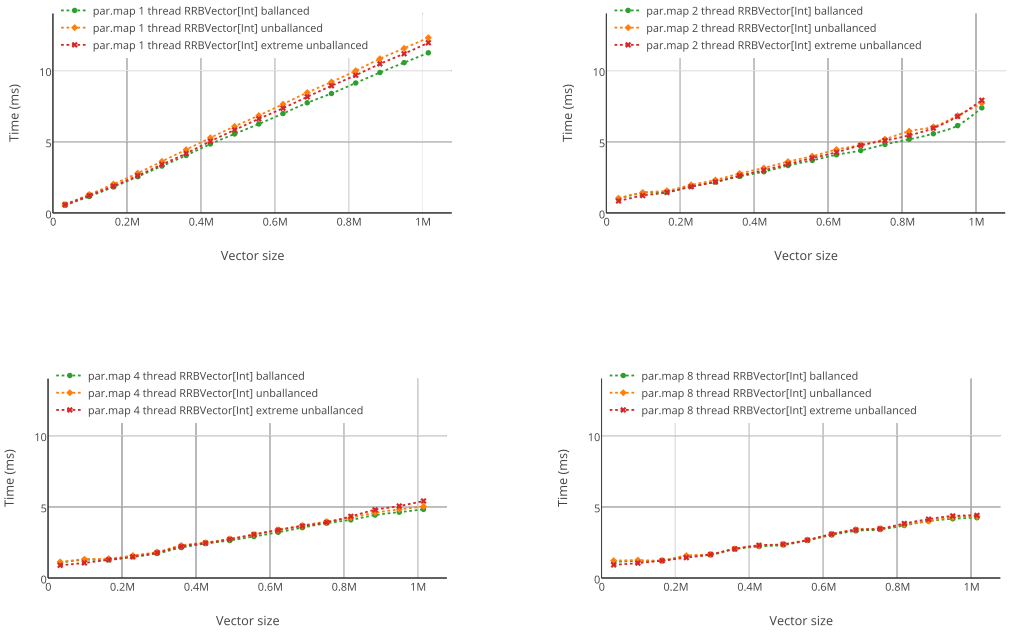

FIGURE 4.21: Benchmark on map and parallel map using the function (x=>x) to show the difference time used in the framework. This time represents the time spent in the splitters and combiners of the parallel collection.

4.4.9 Memory footprint

This is not really a benchmark, it is rather the characterisation of the memory footprint of the vector 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. Fact for balanced ones it is even possible to have a smaller footprint (a few bytes) caused by the truncation of the last branch.

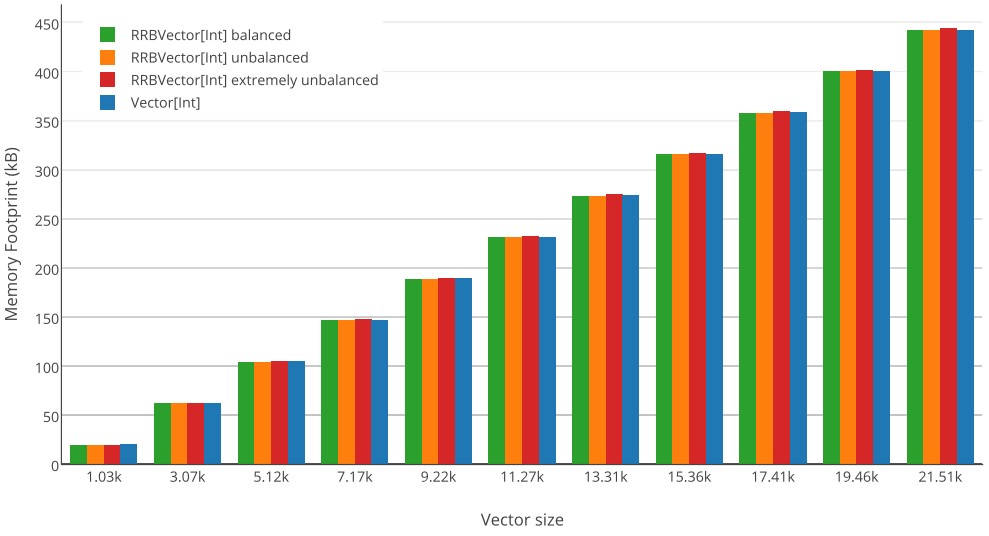

FIGURE 4.22: Memory Footprint for different vectors.

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 Teststing correctness

5.1.1 Unit tests

TODO

5.1.2 Invariant Assertions

TODO

I

Chapter 6

Related Work

List of related subjects:

- Scala Collections and Parallel Collection
- Vector
- RRB-Trees and RRB-Vectors
- Semi-mutable data structures
- Performance and Code specialization (manual staging)
- JVM: Arrays, GC, JIT compiler
- ScalaMeter
- Scala Test
- Scala Reflection and Quasicuotes

6.1 RRB-Vectors in Clojure

TODO

I

Chapter 7

Conclusions

TODO

Bibliography

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

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

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

[4] ScalaMeter. https://scalameter.github.io/, .