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

2.1.1 Tree structure

TODO

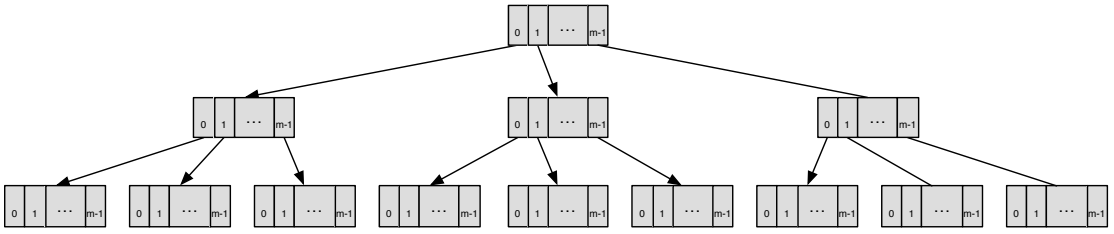


FIGURE 2.1: Radix Balanced Tree Structure

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[1] 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[2]) for some operation sequences the vectors are augmented with transient states.

2.1.2.1 Apply

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

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

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

2.1.2.2 Updated

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

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

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

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

```
def updated(index: Int, elem: A) = {
  def updatedNode(node: Array[AnyRef], depth: Int) = {
    val indexInNode = // compute index
    val 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)
      cloneAndUpdate(node, indexInNode, elem)
    else
      cloneAndUpdate(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 append 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

TODO: why are parallel vector useful

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

2.3.1 Relaxed Tree structure

TODO

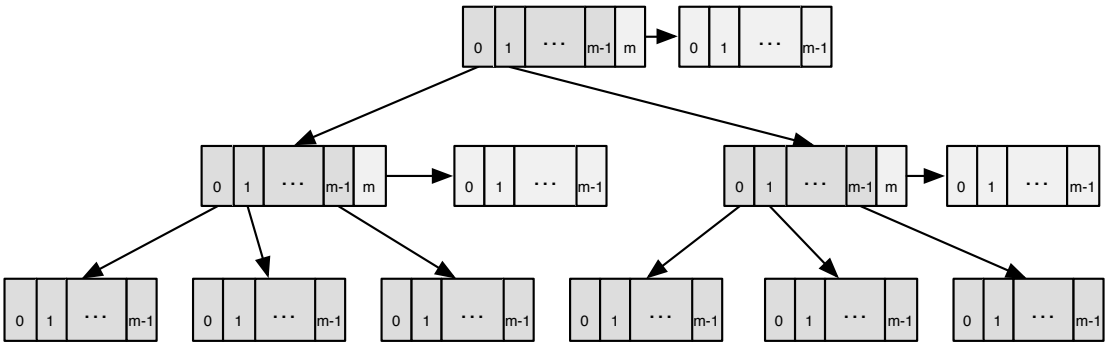


FIGURE 2.2: Radix Balanced Tree

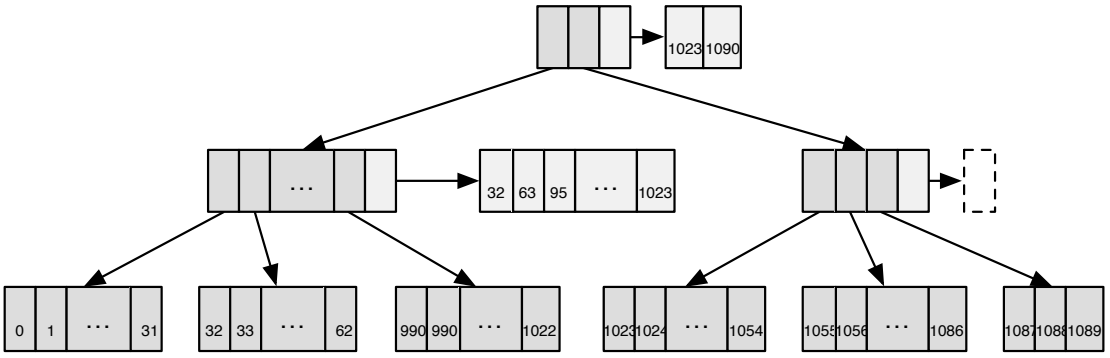


FIGURE 2.3: Relaxed radix example

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.

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

TODO

Insert 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

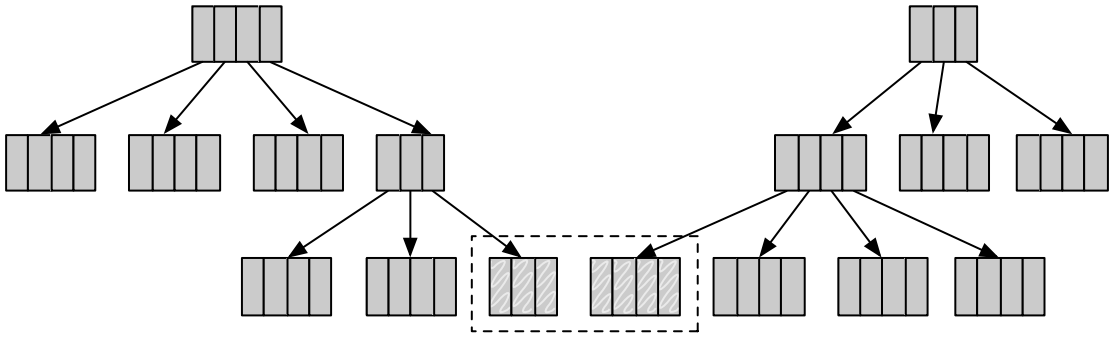


FIGURE 2.4: Concatenation example with blocks of size 4: Rebalancing level 0

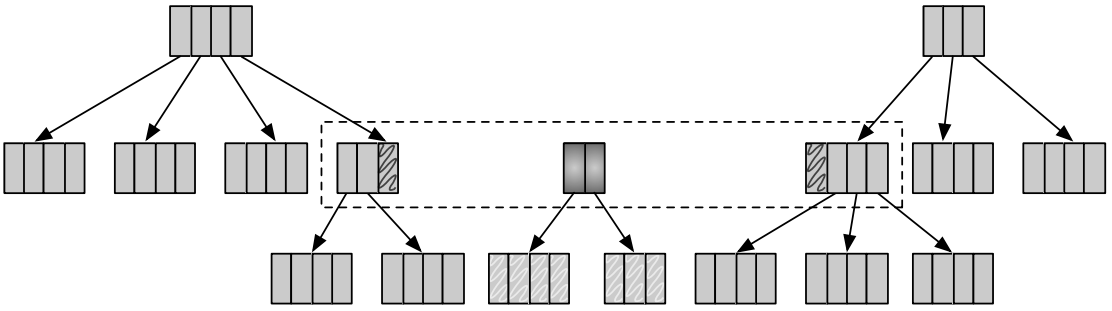


FIGURE 2.5: Concatenation example with blocks of size 4: Rebalancing level 1

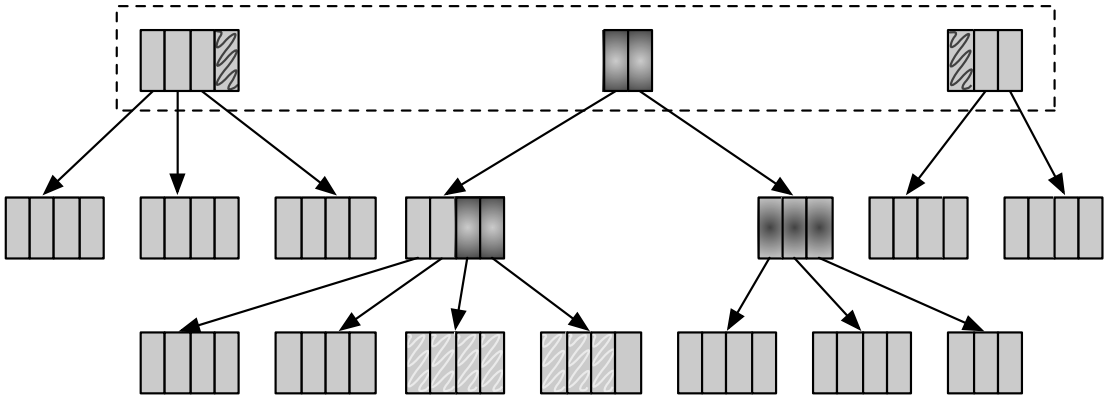


FIGURE 2.6: Concatenation example with blocks of size 4: Rebalancing level 2

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
}
```

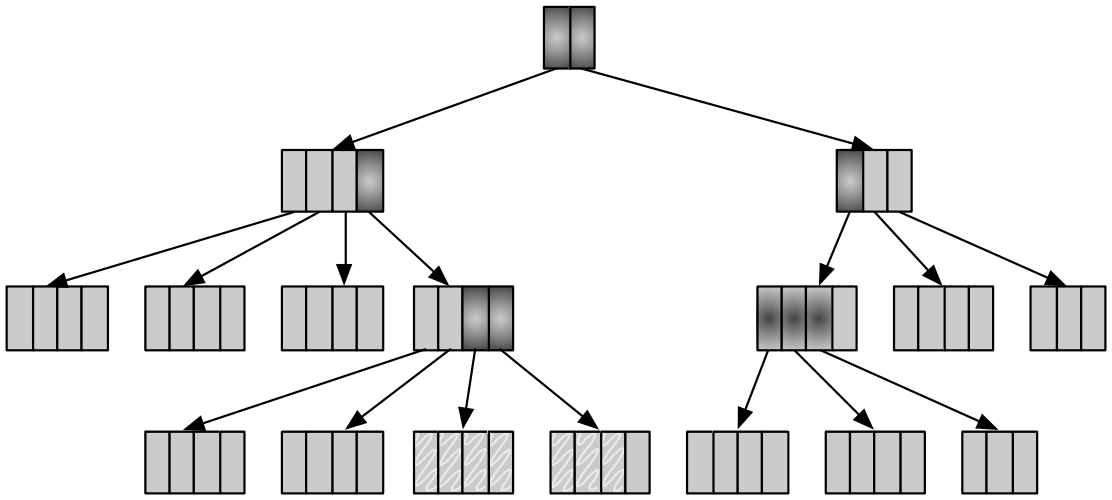


FIGURE 2.7: Concatenation example with blocks of size 4: Rebalancing level 3

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

TODO

2.3.6 Drop

TODO

2.3.6.1 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.

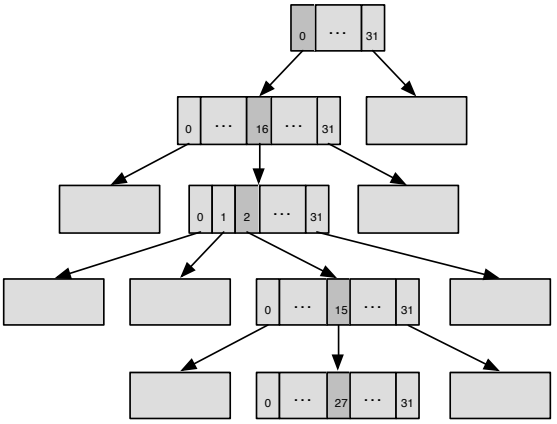


FIGURE 3.1: Accessing element at index 526843 in a tree of depth 5. Empty nodes represent collapses subtrees.

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

TODO

3.2 Displays

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

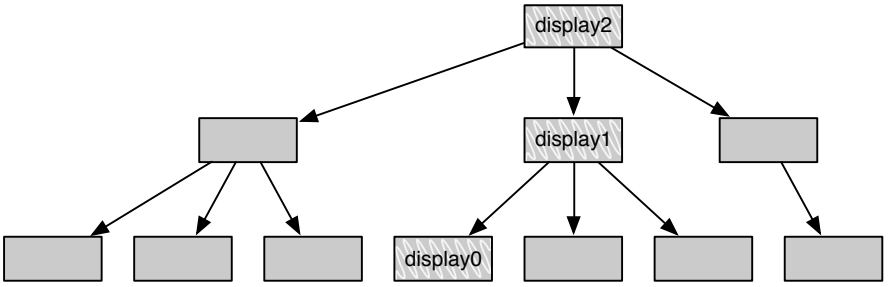


FIGURE 3.2: Displays

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

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

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

3.2.1 As cache

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

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

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

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

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

3.2.2 For transient states

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

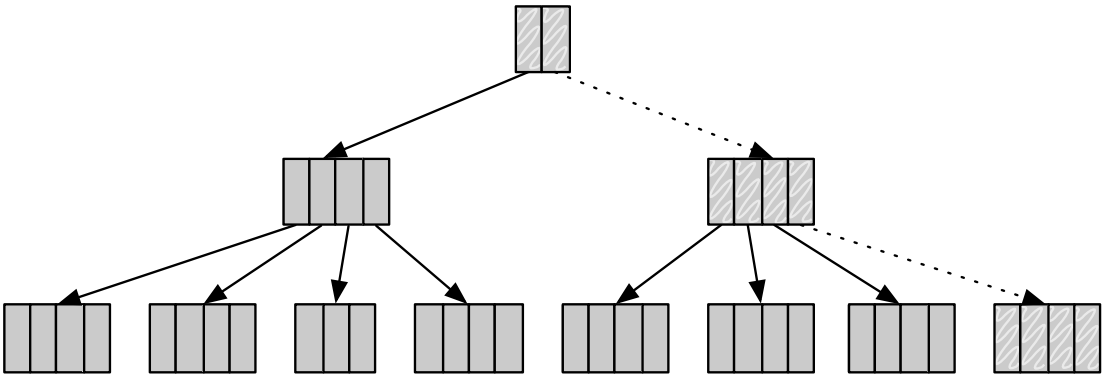


FIGURE 3.3: Transient Tree with current focus displays marked in white and striped nulled edges.

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

3.2.3 Relaxing the Displays

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

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

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

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

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

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

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

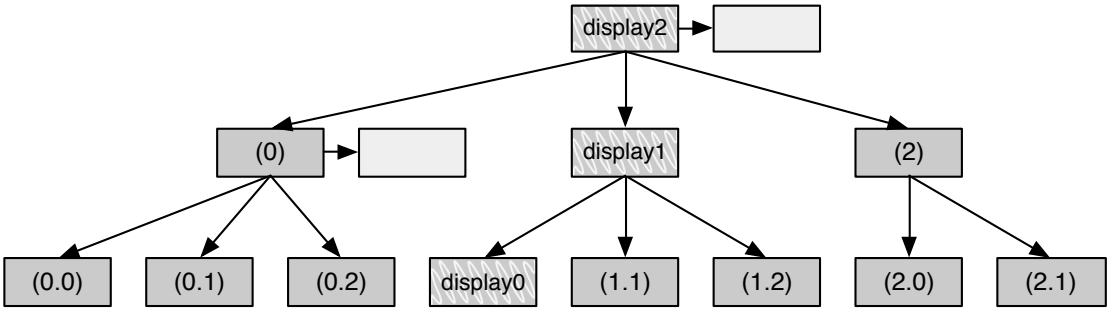


FIGURE 3.4: Relaxed Radix Balanced Tree with a focus on a balanced subtree rooted of `display1`. Light grey boxes represent unbalanced nodes sizes.

3.3 Builder

A vector builder is a special wrapper for an 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

TODO

Relaxing the Iterator TODO

I

Chapter 4

Performance

4.1 In practice: Running on JVM

TODO

4.1.1 Cost of Abstraction and JIT Inline

TODO

4.2 Measuring performance

TODO

4.3 Generators

TODO

4.4 Benchmarks

TODO

4.4.1 Apply

TODO

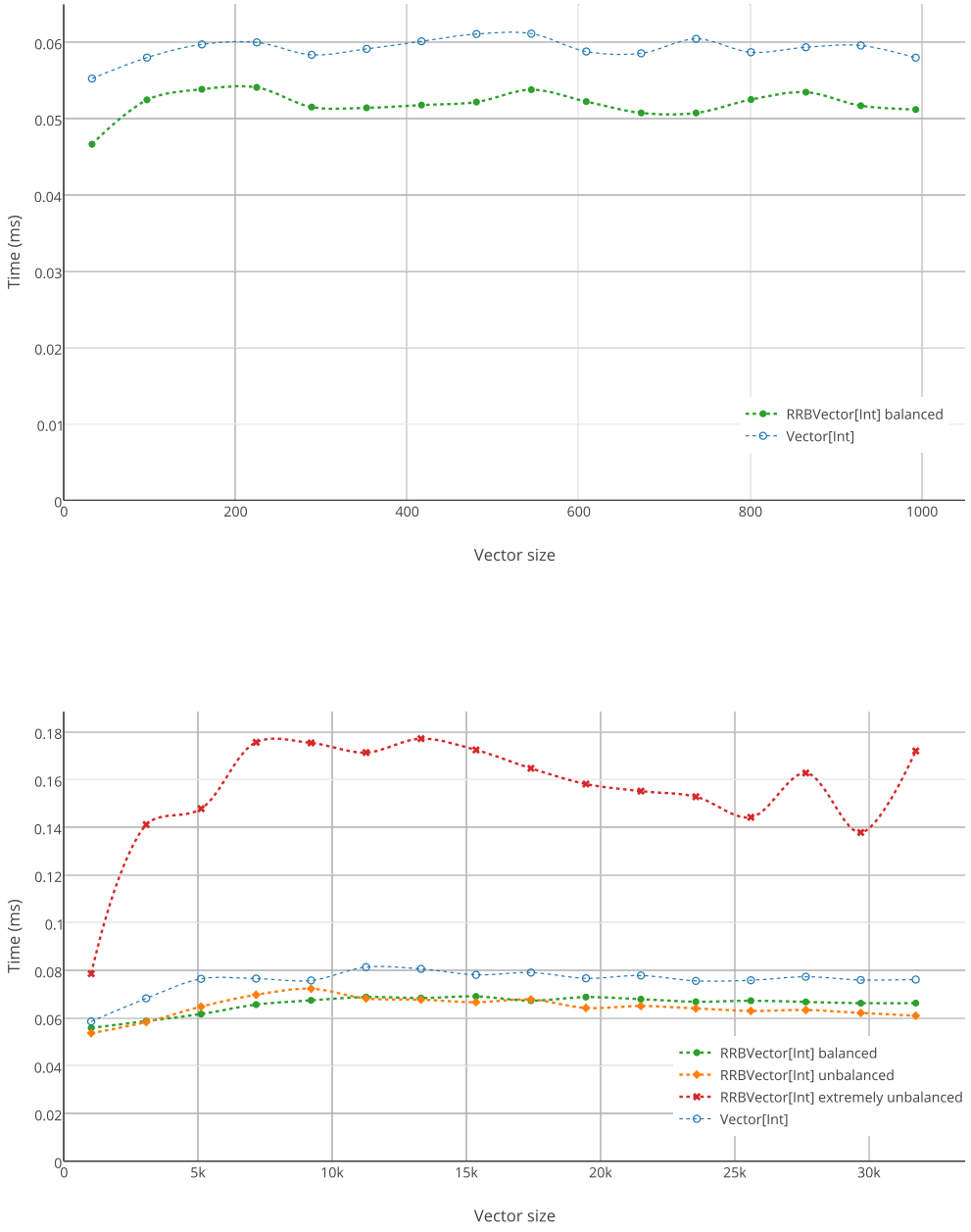


FIGURE 4.1: Time to execute 10k apply operations on sequential indices.

4.4.2 Concatenation

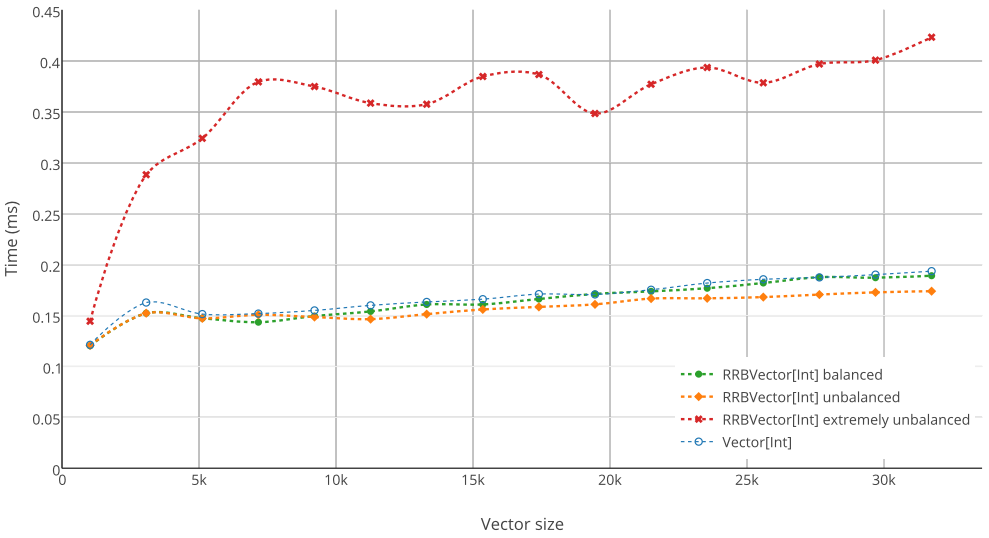


FIGURE 4.2: Time to execute 10k apply operations on random indices.

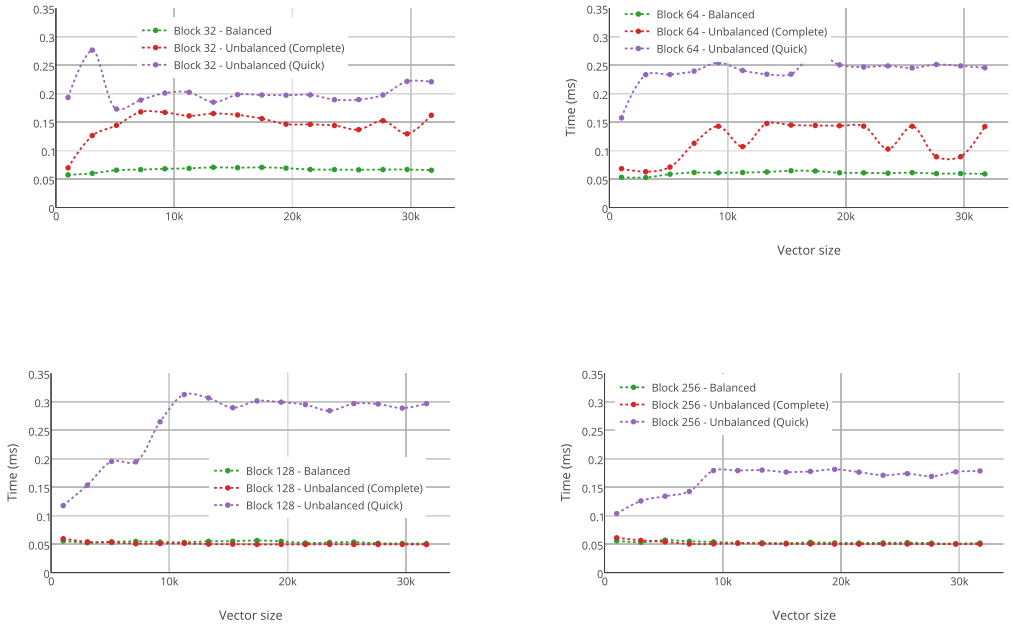


FIGURE 4.3: Time to execute 10k apply operations on sequential indices. Comparing performances for different block sizes and different implementation of the concatenation inner branch rebalancing (Complete/Quick).

TODO

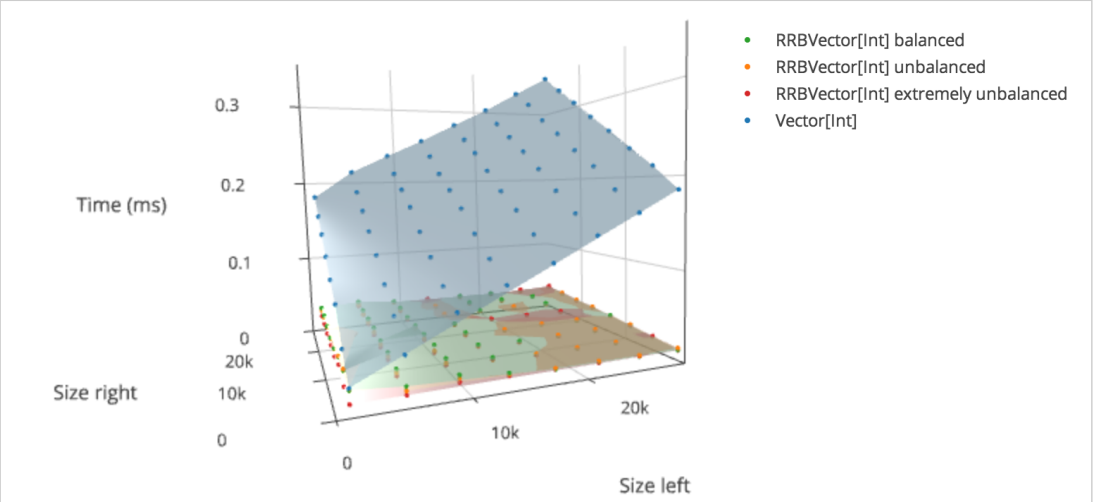


FIGURE 4.4: Execution time for a concatenation operation on two vectors. In theory (and in practice) Vector concatenation is $O(left + right)$ and the rrbVector concatenation operation is $O(log_{32}(left + right))$.

4.4.3 Append

TODO

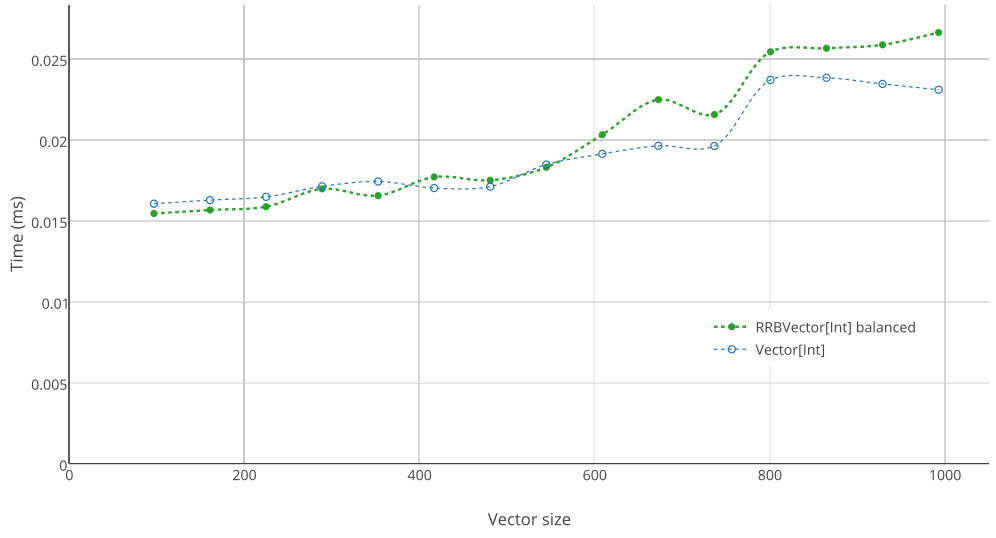


FIGURE 4.5: Time to execute 256 append operations. This shows the amortized cost of the append operation.

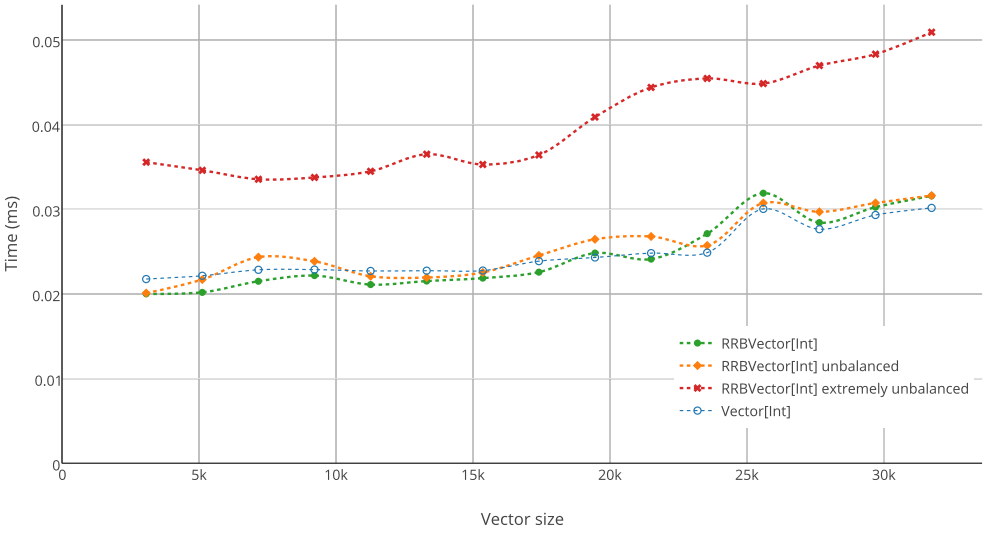


FIGURE 4.6: Time to execute 256 append operations. This shows the amortized cost of the append operation.

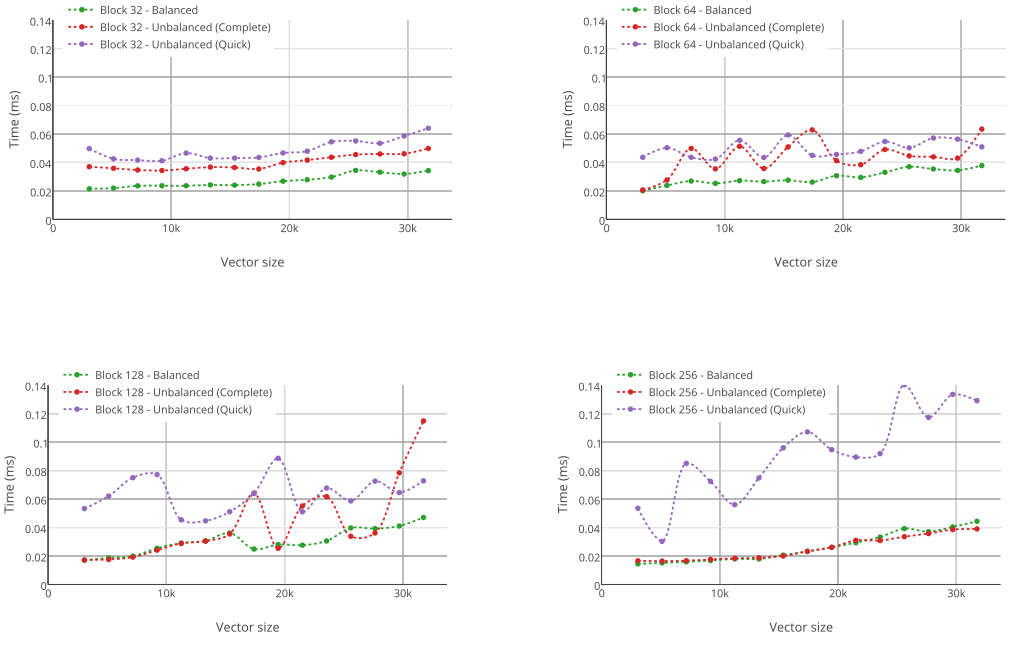


FIGURE 4.7: Time to execute 256 append operations. This shows the amortized cost of the append operation. Comparing performances for different block sizes and different implementation of the concatenation inner branch rebalancing (Complete/Quick).

4.4.4 Prepend

TODO

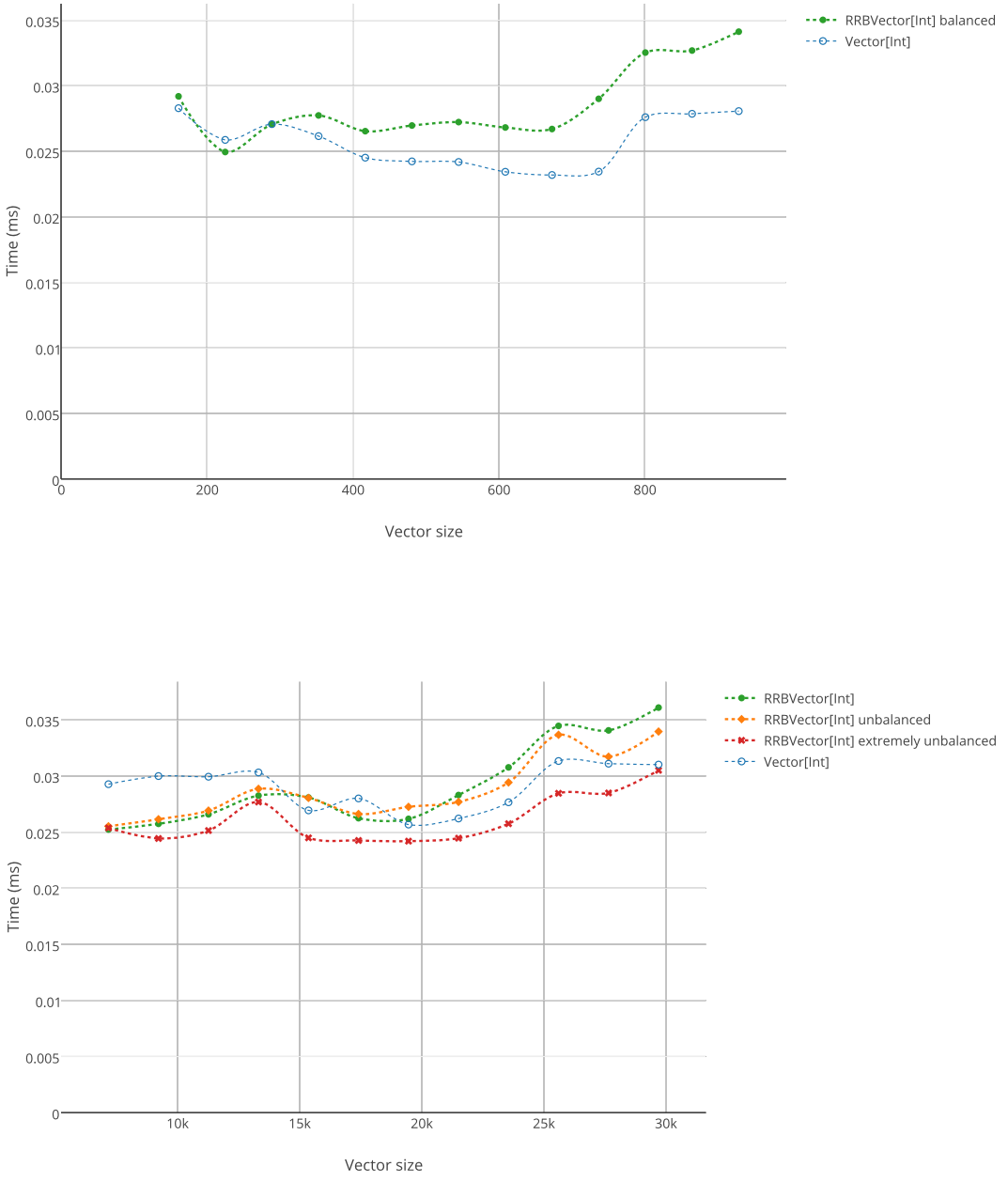


FIGURE 4.8: Time to execute 256 prepend operations. This shows the amortized cost of the prepend operation.

4.4.5 Splits

TODO

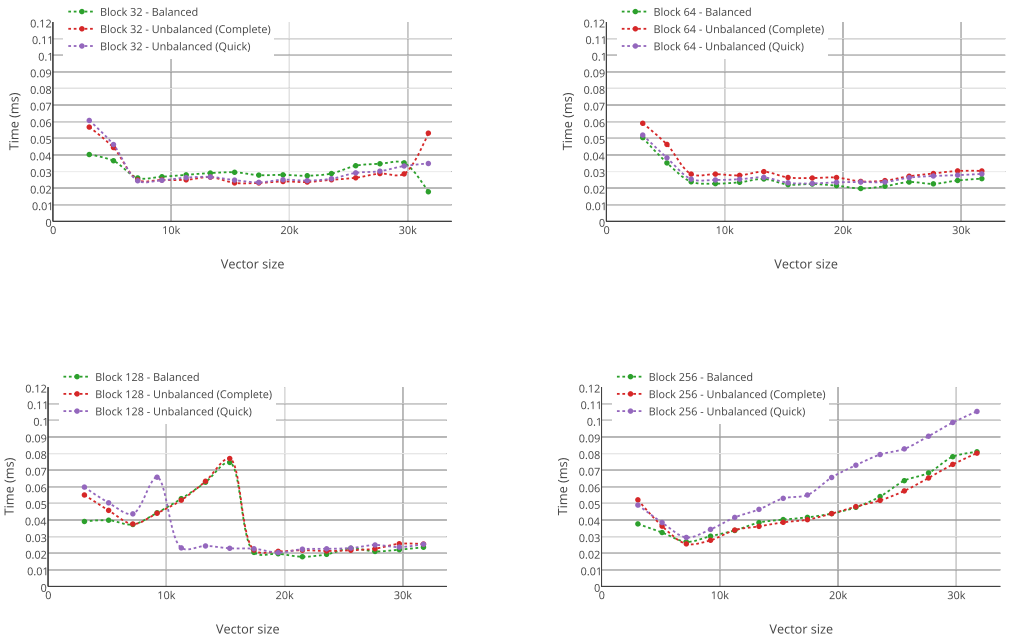


FIGURE 4.9: Time to execute 256 prepend operations. This shows the amortized cost of the append operation. Comparing performances for different block sizes and different implementation of the concatenation inner branch rebalancing (Complete/Quick).

4.4.6 Iterator

TODO

4.4.7 Builder

TODO

4.4.8 Parallel split-combine

TODO

4.4.9 Memory footprint

TODO

I

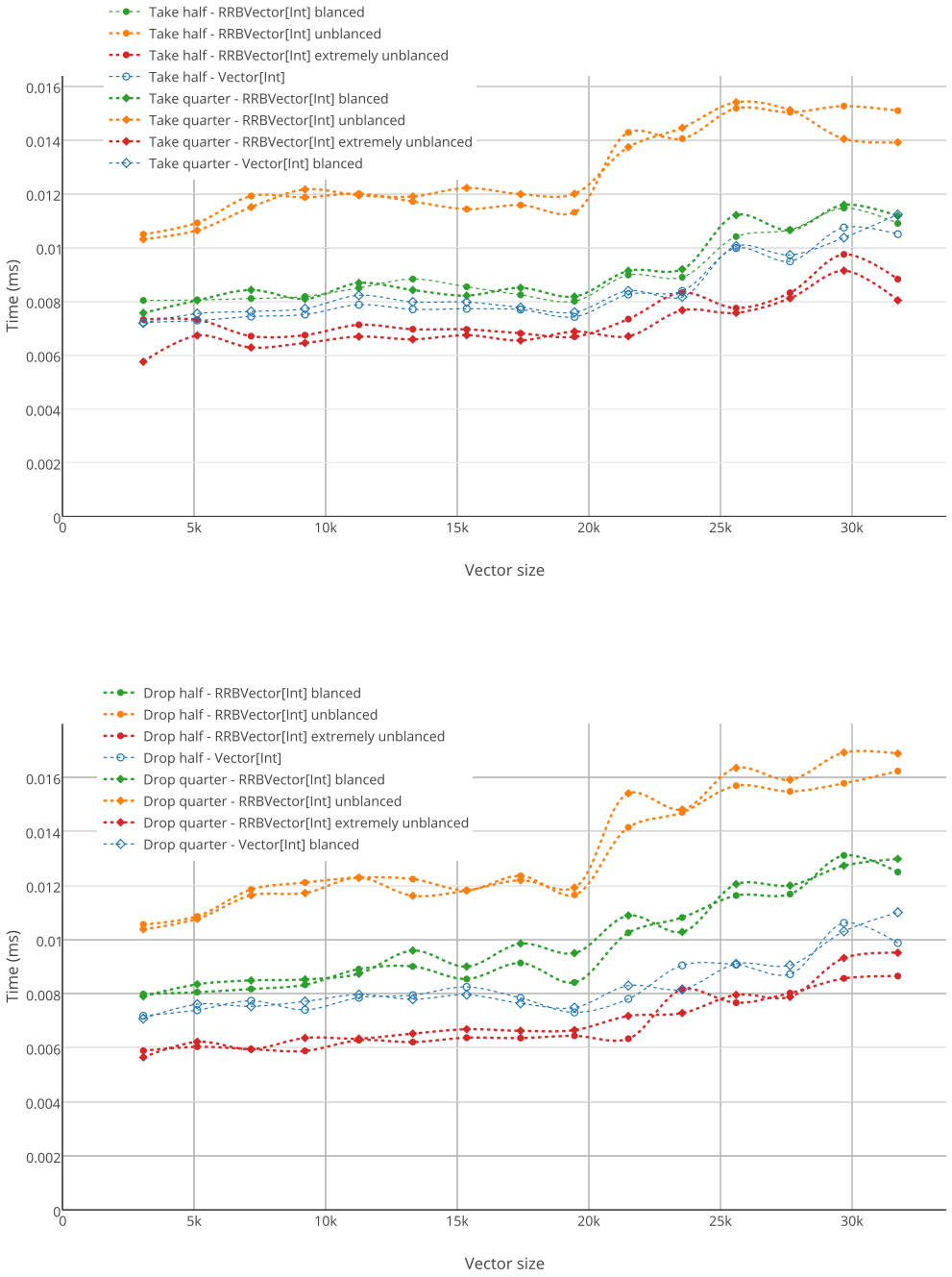


FIGURE 4.10: Execution time of take and drop.

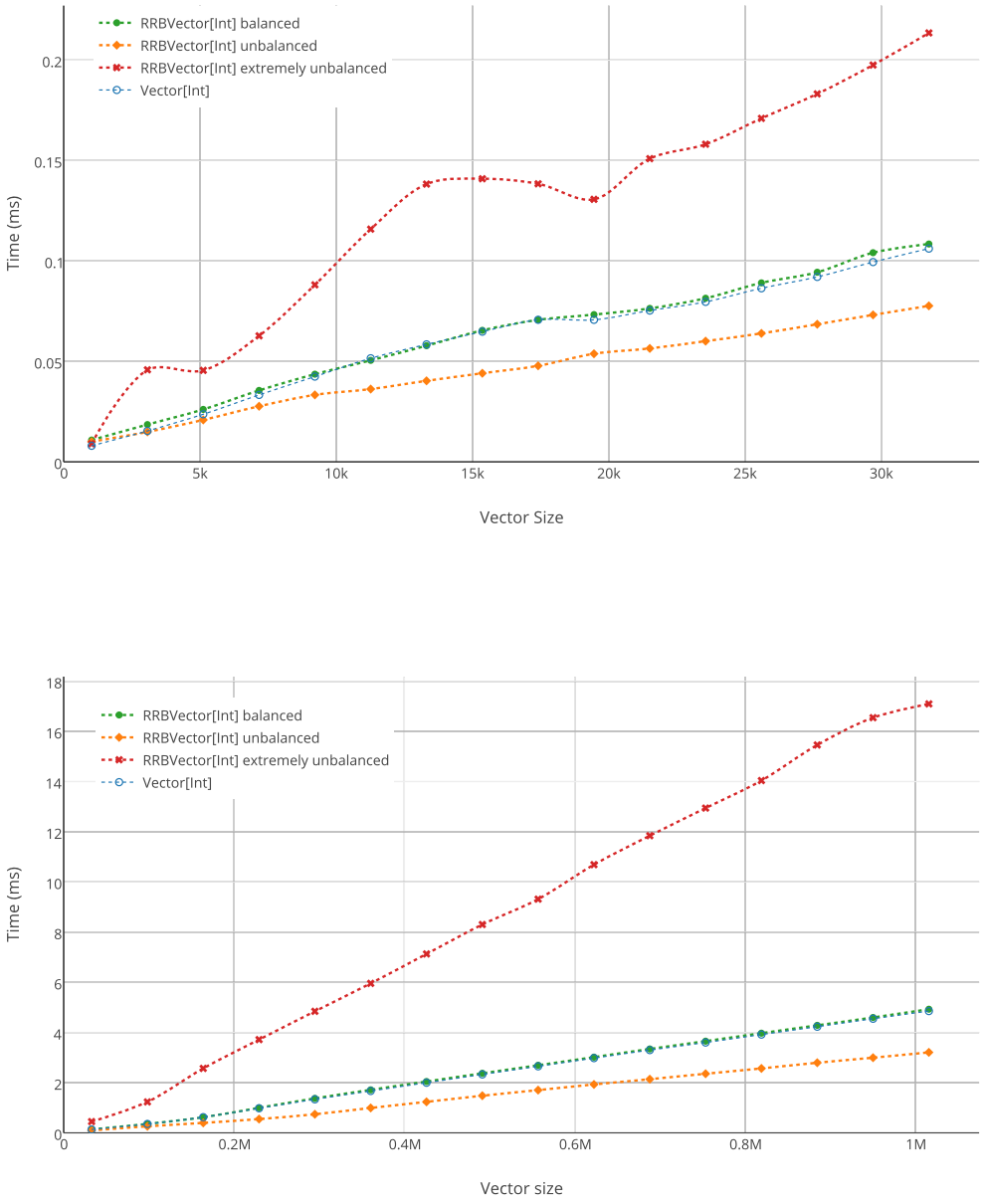


FIGURE 4.11: Excecution time to iterate through all the elements of the vector.

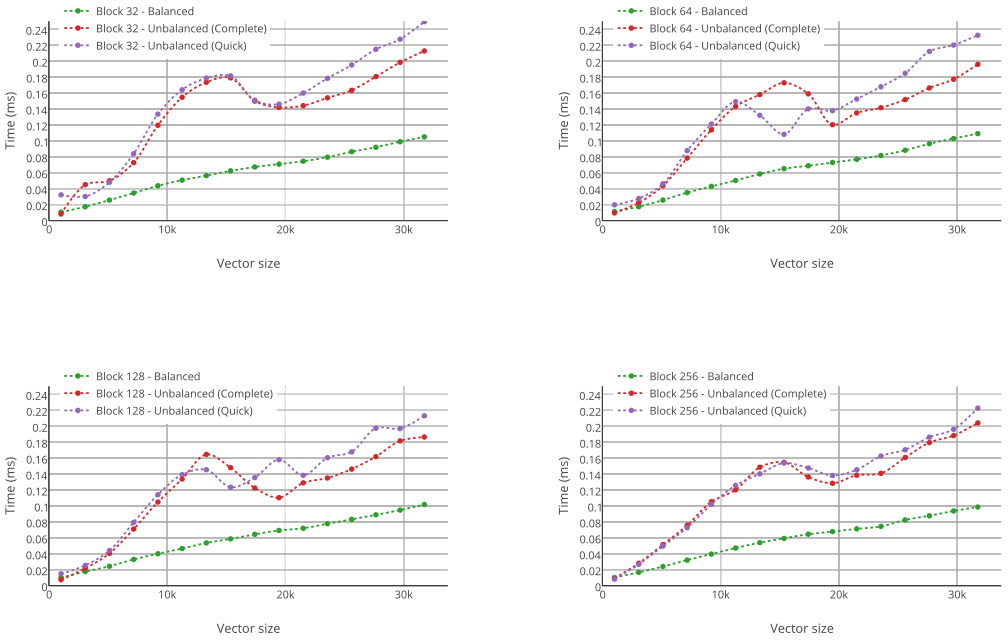


FIGURE 4.12: Excecution time to iterate through all the elements of the vector. Comparing performances for different block sizes and different implementation of the concatenation inner branch rebalancing (Complete/Quick).

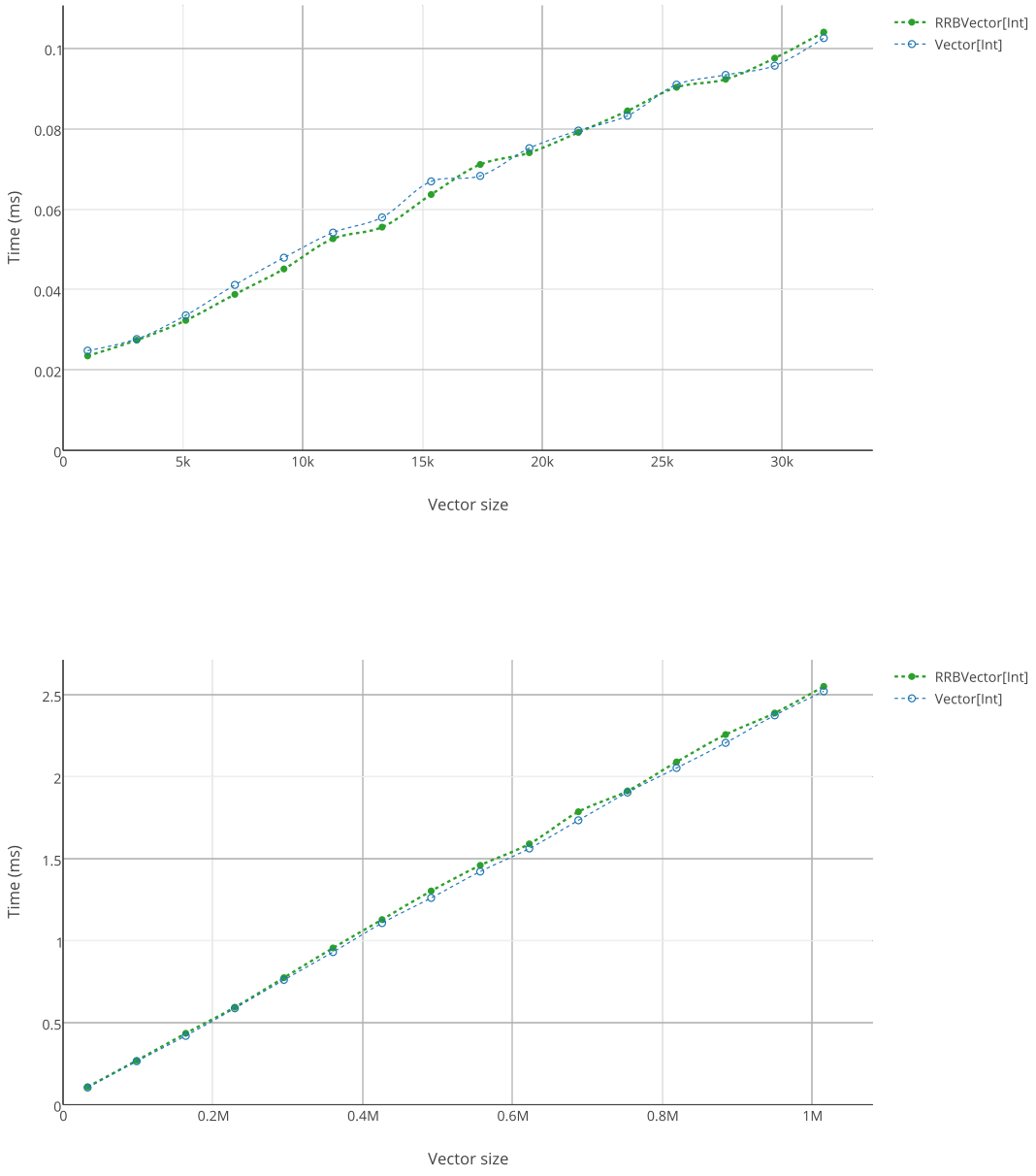


FIGURE 4.13: Execution time to build a vector of a given size.

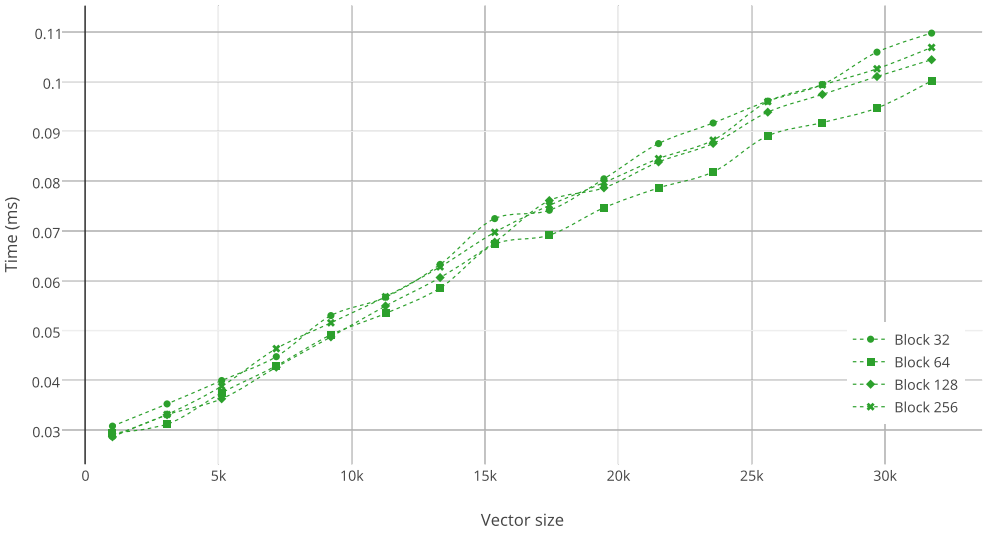


FIGURE 4.14: Execution time to build a vector of a given size. Comparing performances for different block sizes.

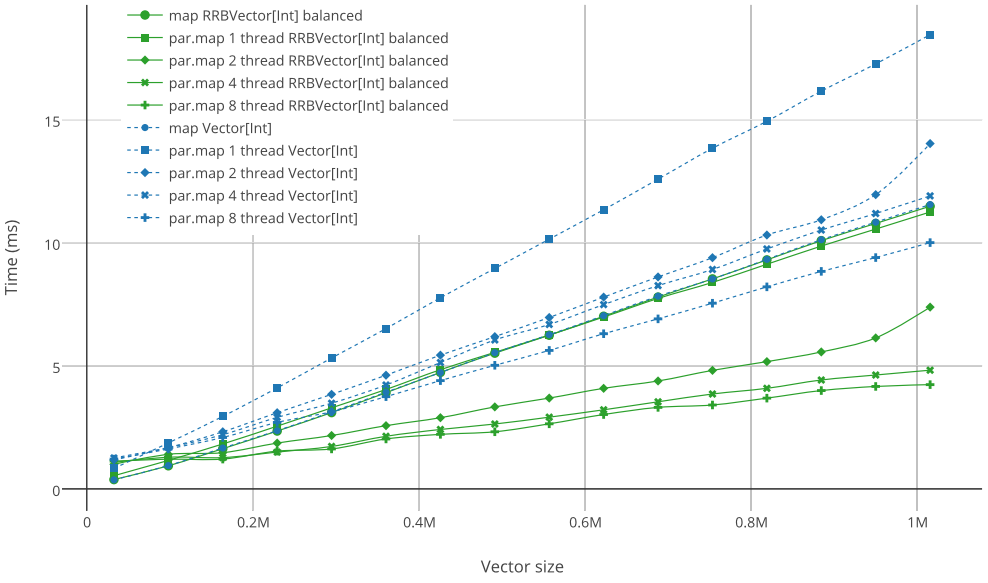


FIGURE 4.15: Benchmark on map and parallel map using the function (x=>x) to show the difference time used in the framework. This time represents the time spent in the splitters and combiners of the parallel collection (iterator and builder for the sequential version).

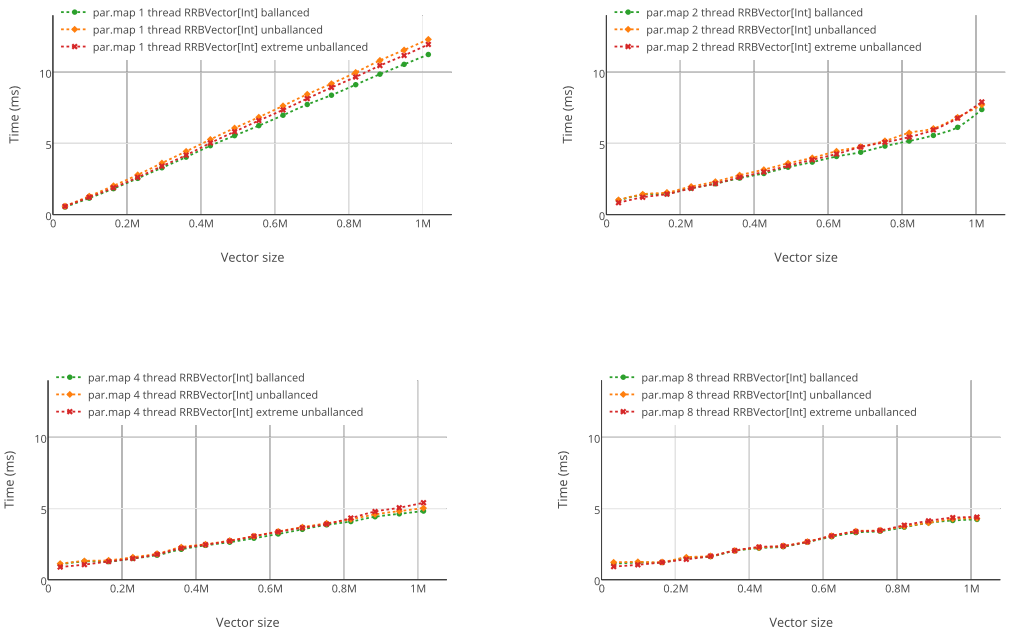
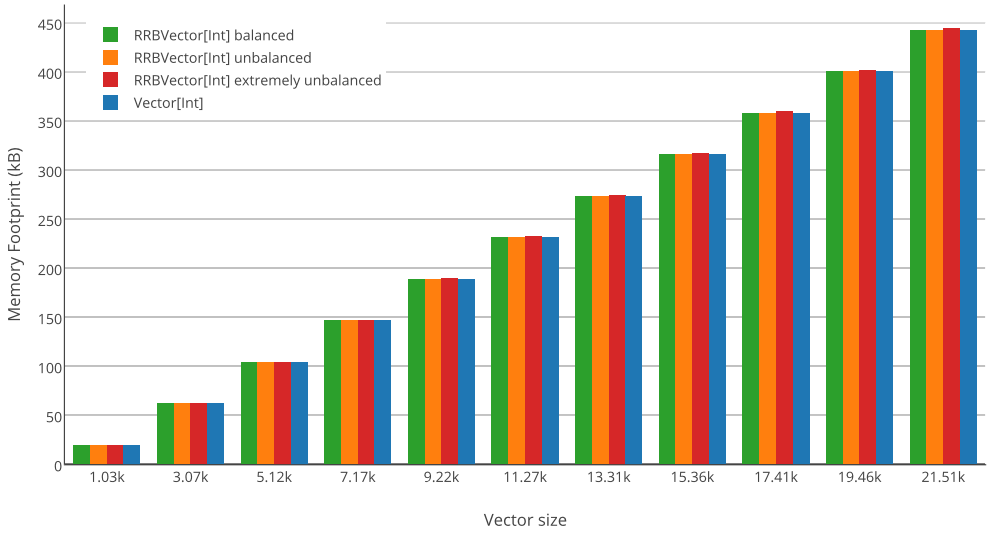


FIGURE 4.16: Benchmark on map and parallel map using the function (x=>x) to show the difference time used in the framework. This time represents the time spent in the splitters and combiners of the parallel collection.



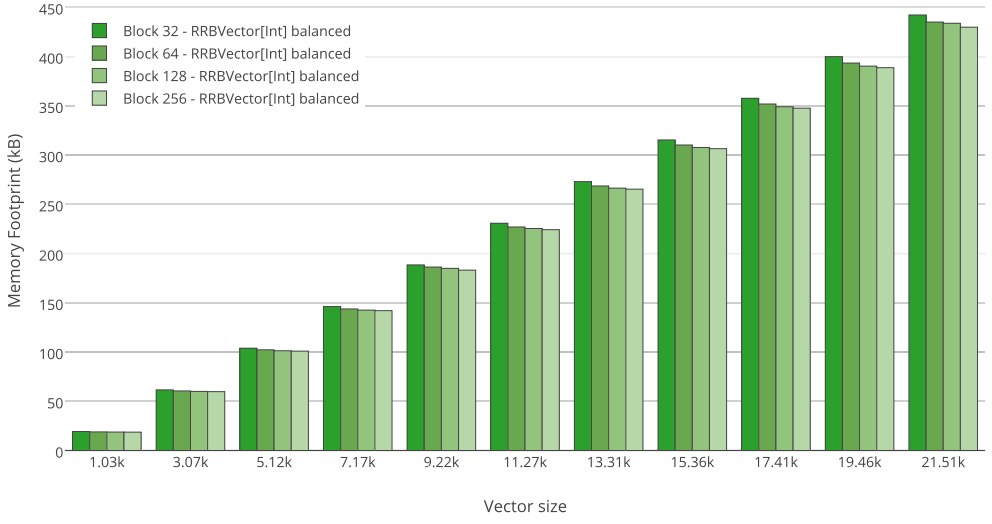


FIGURE 4.17: Memory Footprint

Chapter 5

Testing

5.1 Teststing correctness

5.1.1 Unit tests

TODO

5.1.2 Invariant Assertions

TODO

I

Chapter 6

Related Work

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, .