RRB Vector: A Practical General Purpose Immutable Sequence

Nicolas Stucki[†] Tiark Rompf [‡] Vlad Ureche[†] Phil Bagwell [*]

[†]EPFL, Switzerland: {first.last}@epfl.ch
[‡]Purdue University, USA: {first}@purdue.edu

Abstract

State-of-the-art immutable collections have wildly differing performance characteristics across their operations, often forcing programmers to choose different collection implementations for each task. Thus, changes to the program can invalidate the choice of collections, making code evolution costly. It would be desirable to have a collection that performs well for a broad range of operations.

To this end, we present the `RRB-Vector`, an immutable sequence collection that offers good performance across a large number of sequential and parallel operations. The underlying innovations are: (1) the Relaxed-Radix-Balanced (RRB) tree structure, which allows efficient structural reorganization, and (2) an optimization that exploits spatio-temporal locality on the RRB data structure in order to offset the cost of traversing the tree.

In our benchmarks, the `RRB-Vector` speedup for parallel operations is lower bounded by 7× when executing on 4 CPUs of 8 cores each. The performance for discrete operations, such as appending on either end, or updating and removing elements, is consistently good and compares favorably to the most important immutable sequence collections in the literature and in use today. The memory footprint of `RRB-Vector` is on par with arrays and an order of magnitude less than competing collections.

Categories and Subject Descriptors E.1 [*Data Structures*]: Arrays; E.1 [*Data Structures*]: Trees; E.1 [*Data Structures*]: Lists, stacks, and queues

Keywords Data Structures, Immutable, Sequences, Arrays, Trees, Vectors, Radix-Balanced, Relaxed-Radix-Balanced

1. Introduction

In functional programs, immutable sequence data structures are used in two distinct ways:

- to perform **discrete operations**, such as accessing, updating, inserting or deleting random collection elements;
- for **bulk operations**, such as mapping a function over the entire collection, filtering using a predicate or grouping using a key function.

Bulk operations on immutable collections lend themselves to implicit parallelization. This allows the execution to proceed either sequentially, by traversing the collection one element at a time, or in parallel, by delegating parts of the collection to be traversed in different execution contexts and combining the intermediate results. Therefore, the bulk operations allow programs to scale to multiple cores without explicit coordination, thus lowering the burden on programmers.

Most state-of-the-art collection implementations are tailored to some specific operations, which are executed very fast, at the expense of the others, which are slow. For example, the ubiquitous `Cons` list is extremely efficient for prepending elements and accessing the head of the list, performing both operations in $\mathcal{O}(1)$ time. However, it has a linear $\mathcal{O}(n)$ cost for reading and updating random elements. And although sequential scanning is efficient, requiring $\mathcal{O}(1)$ time per element, it cannot benefit from parallel execution, since both splitting and combining take sequential $\mathcal{O}(n)$ time, cancelling out any gains from the parallel execution.

This non-uniform behavior across different operations forces programmers to carefully choose the collections they use based on the operations required by the task at hand. This ad-hoc choice also stifles code evolution, as new features often rely on different operations, forcing the programmers to revisit their choice of collections. Furthermore, having different collection choices for each module prevents good end-to-end performance, since data must be passed from one collection to another, adding overhead.

Instead of asking programmers to choose a collection which performs well for their needs, it would be much better to provide a default collection that performs well across a broad range of operations, both discrete and bulk. Having such a collection readily available would allow programmers to rely on it without worrying about performance, except in extremely critical places, and would encourage modules to standardize their interfaces around it.

To this end, we present the `RRB-Vector`, an immutable indexed sequence collection that inherits and improves the fast discrete operations of tree-based structures while supporting efficient parallel execution by providing fast split and combine primitives. The `RRB-Vector` is a good candidate for a default immutable collection, thanks to its good all-around performance, allowing programs to use it without the risk of running into unexpected linear or supralinear overheads.

Bulk data parallel operations on the `RRB-Vector` are executed with effectively-constant[1] sequential overheads thanks to the underlying wide Relaxed-Radix-Balanced (RRB) tree structure. The key property is the relaxed balancing requirement, which allows efficient structural changes without introducing extremely unbalanced states. Data parallel operations, such as `map`, are executed in three phases: (1) the `RRB-Vector` is split into chunks in an effectively-

[*] Phil Bagwell passed away on October 6, 2012. He made significant contributions to this work, and to the field of data structures in general.

ICFP'15, August 31 – September 2, 2015, Vancouver, BC, Canada
ACM. 978-1-4503-3669-7/15/08...$15.00
http://dx.doi.org/10.1145/2784731.2784739

[1] Proportional to a logarithm of the size with a large base. In practice our choice of index representation as signed integer limits to $log_{32}(2^{31}) + 1$ which corresponds to approximately 6.2 indirections.

constant sequential operation, (2) each execution context traverses one or more chunks, with an amortized-constant overhead per element and (3) the intermediate results are concatenated in a final effectively-constant sequential operation.

Discrete operations, such as appends on either side, updates and deletions are performed in amortized-constant time. This is achieved thanks to a lightweight fragmented representation of the tree that reduces the propagation of updates to branches, thus exploiting locality of operations. This provides an adapted and more efficient treatment compared to the widely-used tree structural sharing [11], thus lowering the asymptotic complexity from effectively constant to amortized constant. In the worst case, if operations are called in an adversarial manner, the behavior remains effectively constant and the additional overhead is limited to a range check and a single set of assignment operations.

We implemented the `RRB-Vector` data structure in Scala[2] and measured the performance of its most important operations. On 4 cores, bulk operation performance is at least 2.3× faster compared to sequential execution, scaling better with heavier workloads. Discrete operations take either amortized- or effective-constant time, with good constants: compared to mutable arrays, sequential reads are at most 2× slower, while random access is 2-3.5× slower.

We compare our `RRB-Vector` implementation to other immutable collections such as red-black trees, finger trees, copy-on-write arrays and the current `Vector` implementation in the Scala library. Overall, the `RRB-Vector` is at most 2.5× slower than the best collection for each operation and consistently delivers good performance across all benchmarks. The memory footprint of `RRB-Vector` is on-par with copy-on-write arrays and an order of magnitude better than red-black trees and finger trees.

We claim the following contributions:

- We present the Relaxed-Radix-Balanced (RRB) tree data structure and show how it enables the efficient splitting and concatenation operations necessary for data parallel operations (§3);
- We describe the additional structural optimizations that exploit spatio-temporal locality on RRB-Trees (§4);
- We discuss the technical details of our Scala `RRB-Vector` implementation (which is under consideration for inclusion in the Scala standard library as a replacement for the current `Vector` collection) in hope that other language implementers will benefit from our experience (§5);
- We evaluate the performance of our implementation and compare the results of 7 different core operations across 5 different immutable collections (§6).

2. Background: Vectors as Balanced Trees

In this section we present a base version of the immutable `Vector`, which is based on Radix-Balanced trees[3]. This simple version provides logarithmic complexities: $\mathcal{O}(log_m(n))$ on random accesses and $\mathcal{O}(m \cdot log_m(n))$ while updating, appending at either end, or splitting. The constant m is the branching factor (ideally a power of two).

2.1 Radix-Balanced Tree Structure

A Radix-Balanced tree is a shallow and complete (or perfect) m-ary tree located only in the leaves. The nodes have a fixed branching size m, and are either internal nodes linking to sub-trees or leaves containing elements. In practice the branching used is 32 [4, 37], but as we will later see, the node size can be any power of 2, allowing efficient radix-based implementations. Figure 1 shows

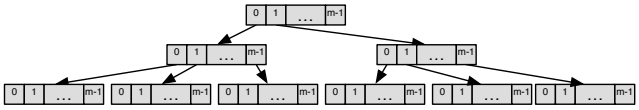

Figure 1. Radix-Balanced tree structure

this structure for m children on each node. Logically each node is a copy-on-write array that contains subtrees or elements.

Apart from the tree itself, a `Vector` keeps the tree height as a field, in order to improve performance. This height is upper bounded by $log_m(n-1)+1$ for nodes of m branches. For example, if m is 32, the tree becomes quite shallow and the complexity to traverse it from root to leaf is considered as effectively constant[4] when taking into account that the number of elements will never be larger than the maximum index representable with 32 bit signed integers, which corresponds to a maximum height of 7 levels[5].

Usually the number of elements in a `Vector` does not exactly match a full tree (m^i for some $i > 0$). To mark the start and end of the elements in the tree, the vector keeps these indices as fields. All subtrees on the left and right that are outside of the filled index range are represented by empty references.

2.2 Core Operations

Indexing Elements are fetched from the tree using radix search on the index. If the tree has a branching factor 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:

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

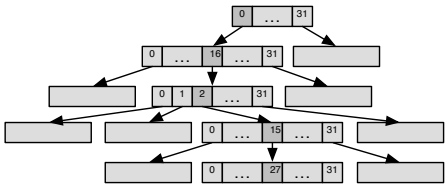

Figure 2. Accessing element at index 526843 in a tree of depth 5. Empty nodes represent collapsed subtrees.

This scheme can be generalized to any branching size m where $m = 2^i$ for $0 < i \leq 31$. The formula is:

$$(index >> (i \cdot L))\&(m-1)$$

It is also possible to generalize for other values of m using the modulo, division and power operations. In that case the formula would become $(index/(m^L)) \mod m$.

The base implementation of the indexing operation requires a single traversal from the root to the leaf containing the element, with the path taken defined by the index of the element and extracted using efficient bitwise operations. With this traversal of the tree, the complexity of the operation is $\mathcal{O}(log_m(n))$.

The same radix-based traversal of the tree is used in the rest of the operations to find the leaf corresponding to a given index. It can be optimized for the first and last leaf by removing computations to improve performance on operations on the ends.

[2] `https://github.com/nicolasstucki/scala-rrb-vector`

[3] Base structure for Relaxed-Radix-Balanced Vector.

[4] There exists a small enough constant bound (due to practical limitations).

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

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

```
val i = // bits in blocks of the index
val mask = (1 << i) - 1
def get(index: Int): A = {
  def getRadix(idx: Int, nd: Node, level: Int): A = {
    if (depth == 0) nd(idx & mask)
    else {
      val indexInLevel = (idx >> (level * i)) & mask
      getRadix(idx, nd(indexInLevel), level-1)
    }
  }
  getRadix(index, vectorRoot, vectorDepth)
}
```

Updating Since the structure is immutable, the `updated` operation has to recreate the entire path from the root to the element being updated. The leaf update creates a fresh copy of the leaf array with one updated element. Then, the parent of the leaf is also updated with the reference to the new leaf, then the parent's parent, and so on all the way up to the root.

```
def updated(index: Int, elem: A) = {
  def updatedNode(node: Node, level: Int): Node = {
    val indexInNode = // compute index
    val newNode = copy(node)
    if(level == 0) {
      newNode(indexInNode) = elem
    } else {
      newNode(indexInNode) =
        updatedNode(node(indexInNode), level-1)
    }
    newNode
  }
  new Vector(updatedNode(vectorRoot, vectorDepth),
    ...)
}
```

Therefore the complexity of this operation is $\mathcal{O}(m \cdot log_m(n))$, since it traverses and recreates $\mathcal{O}(log_m(n))$ nodes of size $\mathcal{O}(m)$. For example, if some leaf has all its elements updated from left to right, the branch will be copied as many times as there are updates. We will later explain how this can be optimized by allowing transient states that avoid re-creating the path to the root tree node with each update (described in §4).

Appending front and back The implementation of `appended` front/back has two cases, depending on the current state of the Radix-Balanced tree: If the first/last leaf is not full the element is inserted directly and all nodes of the first/last branch are copied. If the leaf is full we must find the lowest node in the last branch where there is still room left for a new branch. Then a new branch that only contains the new element is appended to it.

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

```
val m = // branching factor
def appended(elem: A, whr: Where): Vector[A] = {
  def appended(node: Node, level: Int) = {
    val indexInNode = // compute index based on
      start/end index
    if (level == 1)
      copyAndUpdate(node, indexInNode, elem)
    else
      copyAndUpdate(node, indexInNode,
              appended(node(indexInNode), level-1))
  }
  def newBranch(depth: Int): Node = {
    val newNode = Node.ofDim(m)
    val idx = whr match {
      case Frt => m-1
      case Bck => 0
    }
    newNode(idx) =
      if (depth == 1) elem
      else newBranch(depth-1)
    newNode
  }
  if (needNewRoot()) {
    val newRoot = whr match {
      case Frt => Node(newBranch(depth), root)
      case Bck => Node(root, newBranch(depth))
    }
    new Vector(newRoot, depth+1, ...)
  } else {
    new Vector(appendedFront(root, depth), depth, ...)
  }
}
```

In the code above, `isTreeFull` and `needNewRoot` are operations that compute the answer using efficient bitwise operations on the start/end index of the vector.

Since the algorithm traverses and creates new nodes from the root to a leaf, the complexity of the operation is $\mathcal{O}(m \cdot log_m(n))$. Like the `updated` operation, it can be optimized by keeping transient states of the immutable vector (described in §4).

Splitting The core operations to remove elements in a Radix-Balanced tree are the `take` and `drop` operations. They are used to implement many other operations such as `splitAt`, `tail`, `init` and others.

The `take` and `drop` operations are similar. The first step is traversing the tree down to the leaf where the cut will be done. Then the branch is copied and cleared on one side. The tree may become shallower during this operation, in which case some of the nodes on the top will be dropped instead of being copied. Finally, the start and end are adjusted according to the changes on the tree.

```
def take(index) = split(index, Right)
def drop(index) = split(index, Left)
def split(index: Int, removeSide: Side) = {
  def splitRec(node: Node, level: Int): (Node, Int) =
    {
    val indexInNode = // compute index
    if (level == 0) {
      (copyAndSplitNode(node, indexInNode,
       removeSide), 1)
    } else removeSide match {
      case Left if indexInNode == node.length - 1 =>
        splitedRec(node(indexInNode), level - 1)
      case Right if indexInNode == 0 =>
        splitedRec(node(indexInNode), level - 1)
      case _ =>
        val newNode = copyAndSplitNode(node,
          indexInNode, removeSide)
        val (newSubnode, depth) =
          splitedRec(node(indexInNode), level-1)
        newNode(indexInNode) = newSubnode
        (newNode, level)
    }
  }
  val (newRoot, newDepth) = splitRec(vectorRoot,
    vectorDepth)
  new Vector(newRoot, newDepth, ...)
}
```

The computational complexity of any split operation is $\mathcal{O}(m \cdot log_m(n))$ due to the traversal and copying of nodes on the branch where the cut index is located. $\mathcal{O}(log_m(n))$ for the traversal of the branch and then $\mathcal{O}(m \cdot log_m(n_2))$ for the creation of the new branch, where n_2 is the size of the new vector (with $0 \leq n_2 < n$).

3. Immutable Vectors as Relaxed Radix Trees

Relaxed-Radix-Balanced vectors use a new tree structure that extends the Radix-Balanced trees to allow fast concatenation of vectors without losing performance on other core operations [4]. Relaxing the vector consists in using a slightly unbalanced extension of the tree that combines balanced subparts. This vector still en-

sures the $log_m(n)$ bound on the height of the tree and on the operations presented in the previous section.

3.1 Relaxed-Radix-Balanced Tree Structure

The basic difference in the structure is that in an Relaxed-Radix-Balanced (or RRB) tree, we allow nodes that contain subtrees that are not completely full. As a consequence, the start and end index are no longer required, as the branches on the ends can be truncated. The structure of the RRB trees does not ensure by itself that the tree height is bounded by $log_m(n)$. This bound is maintained by each operation using an additional invariant on the tree balance. In our case the concatenation operation is the only one that can affect the inner structure (excluding ends) of the tree and as such it is the only one that needs to worry about this invariant.

Tree balance As the tree will not always be perfectly balanced, we define an additional invariant on the tree that will ensure an equivalent logarithmic bound on the height. We use the relation between the maximum and minimum branching factor m_{max} and m_{min} at each level. These give corresponding maximum height h_{max} and least height h_{min} needed to represent a given number of elements n. Then $h_{min} = log_{m_{max}}(n)$ and $h_{max} = log_{m_{min}}(n)$ or as $h_{min} = \frac{1}{lg(m_{max})} \cdot lg(n)$ and $h_{max} = \frac{1}{lg(m_{min})} \cdot lg(n)$. Trees that are better balanced will have a height ratio, $h_r = \frac{lg(m_{min})}{lg(m_{max})}$, that is closer to 1, perfect balance. In our tree we use $m_{max} = m_{min} + 1$ to make h_r as close to 1 as possible. In practice (using $m = 32$) in the worst case scenario there is an increase from around 6.2 to 6.26 in the maximum possible height (i.e. 7 levels in both cases).

Sizes metadata When one of these trees (or subtrees) is unbalanced, it is no longer possible to know the location of an index just by applying radix manipulation on it. To avoid losing the performance of traversing down the tree in such cases, each unbalanced node will keep metadata on the sizes of its subtrees. The sizes are kept in a separate[6] copy-on-write array as accumulated sizes. This way, they represent the location of the ranges of the indices in the current subtree. To avoid creating additional objects in memory, these sizes are attached at the end of the node. To have a homogeneous representation of nodes, the balanced subtrees have an empty reference attached at the end. For leaves, however, we make an exception: since they will always be balanced, they only contain the data elements but not the size metadata.

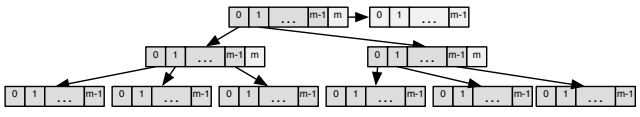

Figure 3. Relaxed radix balanced tree structure

3.2 Relaxed Core Operations

Algorithms for the relaxed version assume that the tree is unbalanced and use a relaxed version of the code for Radix-Balanced trees. But, as soon as a balanced subtree is encountered the more efficient radix based algorithm is used. We also favor the creation of balanced trees/subtrees when possible to improve performance on subsequent operations.

Indexing When the tree is relaxed it is not possible to compute the sub-indices directly from the index. By keeping the accumulated sizes in the node the computation of sub-indices becomes trivial. The sub-index is the same as the first index in the sizes array where $index < sizes[subIndex]$. The fastest way to find it is by using binary search to reduce the search space and when it is small enough to take advantage of cache lines and switch to linear search.

```
def getBranchIndex(sizes: Array[Int], indexInTree:
 Int): Int = {
  var (lo, hi) = (0, sizes.length)
  while (linearThreshold < hi - lo) {
    val mid = (hi + lo) / 2
    if (sizes(mid) <= indexInTree) lo = mid
    else hi = mid
  }
  while (sizes(lo) <= indexInTree) lo += 1
  lo
}
```

Note that to traverse the tree down to the leaf where the index is located, the sub-indices are computed from the sizes as long as the tree node is unbalanced. If the node is balanced, then the more efficient radix based method is used from there to the leaf, to avoid accessing the additional array in each level. In the worst case the complexity of indexing will become $\mathcal{O}(log_2(m) \cdot log_m(n))$ where $log_2(m)$ is a constant factor that is only added on unbalanced nodes.

```
def get(index: Int): A = {
  def getRadix(idx, Int, node: Node, depth: Int) = ...
  def get(idx: Int, node: Node, depth: Int) = {
    val sizes = // get sizes from node
    if(isUnbalanced(sizes)) {
      val branchIdx = getBranchIndex(sizes, idx)
      val subIdx = indexInTree-sizes(branchIdx)
      get(subIdx, node(branchIdx), depth-1)
    } else getRadix(idx, node, depth)
  }
  get(index, root, depth)
}
```

Updating and Appending For each one of these operations, the only fundamental difference with the Radix-Balanced tree is that when a node of a branch is updated the sizes must be updated with it (if needed). In the case of updating, the structure does not change and as such it always keeps the same sizes object reference. The traversal down the tree is done using the new abstraction used in the relaxed version of indexing.

In the case of appending to the back, an updated unbalanced node must increment the accumulated size of its last subtree by one. When a new branch is appended, a new size is appended to the sizes. The `newBranch` operation is simplified by using truncated nodes and letting the node on which it gets appended handle any index shifting required.

```
def appended(elem: A, whr: Where): Vector[A] = {
  ...
  def newBranch(depth: Int): Node = {
    val newNode = Node.ofDim(1)
    newNode(0) = if (depth == 1) elem else
      newBranch(depth-1)
    newNode
  }
  ...
}
```

In the case of appending front, an updated node must increment the accumulated size of each subtrees by one. When a new branch is appended, a 1 is appended on the front of the sizes and all other accumulated sizes are incremented by one.

The complexity of these operations is still $\mathcal{O}(m \cdot log_m(n))$, $log_2(m) \cdot log_m(n)$ for the traversal plus $m \cdot log_m(n)$ for the branch update or creation.

Splitting While splitting, the traversal down the tree is done using the relaxed version of indexing. The splitting operation just truncates the node on the left/right. In addition, when encountering an unbalanced node, the sizes are truncated and adjusted. The

[6] To be able to share them across different vectors. This is a common case when using `updated`.

complexity of this operation is still $\mathcal{O}(m \cdot log_m(n))$, $log_2(m) \cdot log_m(n)$ for the traversal plus $m \cdot log_m(n)$ for the branch update.

3.3 Concatenation

The concatenation algorithm used on RRB-Vectors is a slightly modified version of the one proposed in the RRB-Trees technical report [4]. This version favors nodes of size m over $m-1$ making the trees more balanced. With this approach, we sacrifice a bit of performance for concatenations but we gain performance on all other operations: better balancing implies higher chance of using fast radix operations on the trees.

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 nodes, each of them is rebalanced in order to ensure the $\mathcal{O}(log_m(n))$ bound on the height of the tree and avoid the degeneration of the structure. The RRB version of concatenation has a time complexity of $\mathcal{O}(m^2 \cdot log_m(n))$ where m is constant.

```
1  def concatenate(left: Vector[A], right: Vector[A]) =
     {
2     val newTree = mergedTrees(left.root, right.root)
3     val maxDepth = max(left.depth, right.depth)
4     if (newTree.hasSingleBranch)
5       new Vector(newTree.head, maxDepth)
6     else
7       new Vector(newTree, maxDepth+1)
8   }
9   def mergedTrees(left: Node, right: Node, depth: Int)
      = {
10    if (depth==1) {
11      mergedLeaves(left, right)
12    } else {
13      val merged =
14        if (depth==2) mergedLeaves(left.last,
            right.first)
15      else mergedTrees(left.last, right.first, depth-1)
16      mergeRebalance(left.init, merged, right.tail)
17    }
18  }
19  def mergedLeaves(left: Node, right: Node) = {
20    // create a balanced new tree of height 2
21    // with all elements in the nodes
22  }
```

The concatenation operation starts at the bottom of the branches by merging the leaves into a balanced tree of height 2 using `mergedLeaves`. 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 get dropped later. New sizes of nodes are computed each time a node is created based on sizes of children nodes.

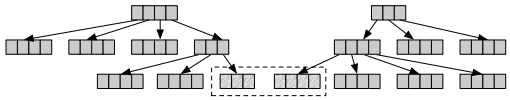

Figure 4. Concatenation example: Rebalancing level 0

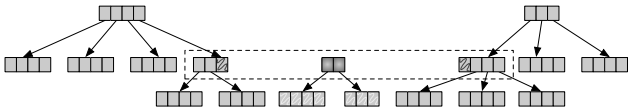

Figure 5. Concatenation example: Rebalancing level 1

The rebalancing algorithm has two proposed variants. The first consists of 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 rebalances the minimum amount of nodes that ensures the logarithmic bound. The first one leaves the tree better balanced, while the second is faster. As we aim to have good performance on all operations we use the first variant[7]. The following snippet of code shows a high level implementation for this first variant. Details for the second variant can be found in [4] in case that concatenation is prioritized over all other operations.

```
1  def mergeRebalance(left: Node, center: Node, right:
     Node) = {
2    // join all branches
3    val merged = left ++ centre ++ right
4    var newRoot = new ArrayBuilder
5    var newSubtree = new ArrayBuilder
6    var newNode = new ArrayBuilder
7    def checkSubtree() = {
8      if(newSubtree.length == m) {
9        newRoot += computeSizes(newSubtree.result())
10       newSubtree.clear()
11     }
12   }
13   for (subtree <- merged; node <-subtree) {
14     if(newNode.length == m) {
15       checkSubtree()
16       newSubtree += computeSizes(newNode.result())
17       newNode.clear()
18     }
19     newNode += node
20   }
21   checkSubtree()
22   newSubtree += computeSizes(newNode.result())
23   computeSizes(newRoot.result)
24 }
```

Figures 4, 5, 6 and 7 show a concrete step by step (level by level) example of the concatenation of two vectors. In the example, some of the subtrees were collapsed. This is not only to make the diagrams fit, but also to expose only the nodes that are referenced during the execution of the algorithm. Nodes with colors represent new nodes and changes, to help track them from figure to figure.

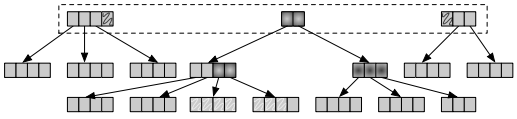

Figure 6. Concatenation example: Rebalancing level 2

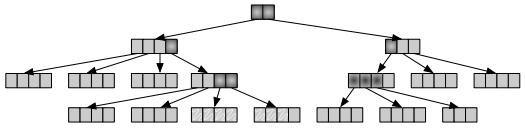

Figure 7. Concatenation example: Rebalancing level 3

The concatenation algorithm chosen for the RRB Vector is the one that is slower but that is better at rebalancing. The reason behind this decision is that with better balanced trees all other operations on the trees are more efficient. In fact, choosing the least efficient option does not need to be seen as a reduction in performance, because the improvement is in relation to the Relaxed-Balanced tree concatenation of linear complexity. An interesting consequence of this choice is that all trees (or subtrees) of size at most m^2 (the maximum size of a two level RRB tree) that were created by concatenation will be completely balanced.

It is important to have a smart rebalancing implementation, due to the m^2 elements that can possibly be accessed. The first crucial factor is the speed of copying the nodes. With an implementation that takes advantage of spatial locality by using arrays (§5.2), the amount of work required can be reduced to m fast node copies

[7] Performance of operations using the second variant was analyzed in [37].

rather than m^2 element copies. Another crucial but obvious implementation detail is to never duplicate a node if it does not change. This requires a small amount of additional logic and comes with a benefit on memory used and in good cases can reduce the number of node copies required, potentially reducing the effective work to $o(m * log_m(n))$ if there is a good alignment.

When improving the vector on locality (§4), concatenating a small vector using the concatenation algorithm is less efficient than appending directly on the other tree. That case is identified by a simple bound on the lengths, and then all elements from the smaller vector are appended to the larger one.

Other Operations Having efficient concatenation and spitting allows us to also implement several other operations that change the structure of the tree. Some of there operations are: inserting an element/vector in any position, deleting an element/subrange of the vector and patching/replacing part of the vector. The complexity of these operations are bounded by the complexity of the core operations used.

Parallelizing the Vector To parallelize operations we use the fork-join pool model from Java [18, 31]. In this model processing is achieved by splitting the work into smaller parts until they are deemed small enough to ensure good parallelism. This can be achieved using the efficient splitting of the RRB-Tree. For certain operations, like `map`, `filter` and `reduce`, the results obtained in parallel must be aggregated, such as concatenating the partial vectors produced by the parallel workers. The aggregation can occur in several steps, where partial results from different workers are aggregated in parallel, recursively, until a single result is produced. The overhead associated with the distribution of work is $\mathcal{O}(m^2 \cdot log_m(n))$.

4. Improvements on Operation Locality and Amortization of Costs

In this section we present different approaches aimed at improving the performance using explicit caching on the subtrees in a branch. This is a new generalization of the Clojure [15] (and current Scala) optimizations on their Radix-Balanced vectors. All optimizations we describe rely on the vector object keeping a set of fields that track the entire branch of the tree, from the root to a leaf. Figure 4 shows such an RRB-vector with the levels numbered starting from 0 at the bottom. The explicit caches focus on the nodes reaching from the root to the `tree0` leaf.

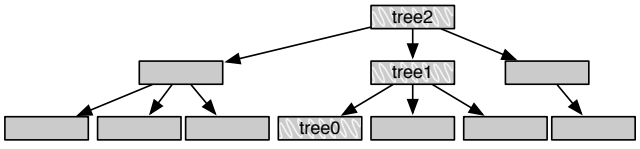

Figure 8. Branch trees in cache.

To know on which branch the vector is focused there is also a focus index field. It can be the index of any element in the current `tree0`. 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. Each method that creates a new vector must decide which focus to set.

These optimizations depend heavily on the radix operations for efficiency. To avoid losing these completely on unbalanced RRB trees we will only use these operation on a balanced subtree of the branch. The vector will keep extra meta data on the start and end index of this subtree as well as its height. In the case of a balanced RRB tree this covers the entire tree and will effectively only use the more efficient radix based operations.

4.1 Faster Access

One of the uses of the focused branch is as a direct access to 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 is needed, it can be fetched from the lowest common node of the two branches.

To know which is the level of the lowest common node in a vector of branching size m (where $m = 2^i$ and i is the number of bits in the sub-indices), only the focused index and the index being fetched are needed. The operation $index \veebar focus$ will return a number that is bounded to the maximum number of elements in a tree of that level. The actual level can be extracted with some 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. This is computed in $\mathcal{O}(log_m(n))$ without accesses to memory.

```
1  val i = // nubmer of bits of sub-indices
2  def lowestCommonLevel(idx: Int, focus: Int): Int = {
3    val xor = idx ^ focus
4    if (xor < (1<<(1*i)) ) 0
5    else if (xor < (1<<(2*i)) ) 1
6    else if (xor < (1<<(3*i)) ) 2
7    ...
8    else 5
9  }
```

When deciding which will be the focused branch of a new vector two heuristics are used: If there was an update operation on some branch where that operations could be used again, that branch is used as focus. If the first one can't be applied, the focus is set to the first element as this helps key collection operations (such as getting an iterator).

The vector iterator and builder use this to switch from one leaf to the next one with the minimal number of steps. In fact, this effectively amortizes out the cost of traversing the tree structure over the accesses in the leaves as each edge of the tree will only be accessed once. In the case of RRB tree iteration there is an additional abstraction for switching from one balanced subtree to the next one.

4.2 Amortizing Costs using Transient States

Transient states are the key to providing amortized constant-time appending, local updating and local splits. To achieve this, we decouple the tree by creating an equivalent tree that does not contain redundant edges on the current focused branch. The information missing in the edges of the tree is represented and can be reconstructed from the trees in the focused branch.

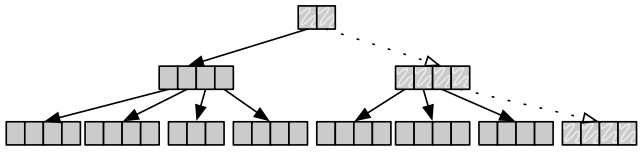

Figure 9. Transient tree with current focus branch marked in white and striped nulled edges.

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

When appending or updating consecutive indices, $\frac{m-1}{m}$ operations must only update the leaf, then $\frac{m-1}{m^2}$ need to update two lev-

els of the tree and so on. These operations will thus be amortized to constant time if they are executed in succession. This is due to the bound given by average number of node update per operation: $\sum_{k=1}^{\infty} \frac{k \cdot (m-1)}{m^k} = \frac{m}{m-1}$.

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 paying off after 3 consecutive operations. With 2 consecutive operations they are matched and with 1 there is a loss in performance.

Canonicalization The transient state aims to improve performance of some operations by amortizing costs. But, the transient state is not ideal for performance of other operations. For example an indexing operation on an unbalanced vector may lack the size information it requires to efficiently access certain indices. And an iterator relies on a canonical tree for performance. It is possible to implement these operations on a transient state, but this involves both code duplication and additional overhead on each call.

The solution we used involves converting the transient representation to a canonical one. This conversion, called canonicalization, is applied when an operation that requires the cannonical form is called on an instance of the immutable vector. The mutation of the vector is not visible from the outside and only happens at most once (Figure 10). This transformation only affects the nodes that are on the focused branch, as it copies each one (except the leaf) and links the trees. If the node is unbalanced, the size of the subtree in focus is inserted. This transformation could be seen as a lazy initialization of the current branch.

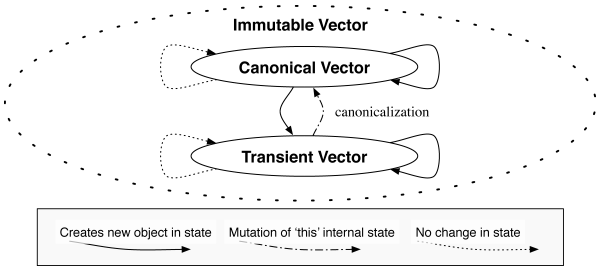

Figure 10. Objects states and effect of operations.

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

4.3 Comparison

In table 1 we show the difference in complexities between the Radix-Balanced Vector and the Relaxed-Radix-Balanced Vector. In table 2 we compare the RRB-Vector to other possible implementations. The operations in the table represent all different complexities that can be reached for the core operations. The operations are divided into four categories: (i) fetching, (ii) updating, (iii) inserting , and (iv) removing. In (i) there is the indexing or random access given an element index and the sequential scanning of elements. Category (ii) is divided into normal (or random) update and has a special case for updates that are done locally. The category (iii) is divided into building (with/without result size), concatenation and insertions of element (or vectors) and has a special case for insertions on ends (`appended` front/back and small `concat`). Removing (iv) is divided into splits (`split`, `take`, `drop`, ...), splits ends (`tail`/`init`, `drop(n)`/`take(n)` with n in the first/last leaf) and a general removal of elements in a range of indices.

Table 1. Comparison between the Radix and Relaxed Radix Vectors. In this table all aC have log_m worst case scenario.

	Radix-Balanced Vector	RRB Vector	With $m = 32$
indexing	log_m	log_m	eC
scanning	aC	aC	aC
update	$m \cdot log_m$	$m \cdot log_m$	eC
update local	aC	aC	aC
concat/insert	L	$m^2 \cdot log_m$	L v.s. eC
insert ends	aC	aC	aC
building	aC	aC	aC
split	$m \cdot log_m$	$m \cdot log_m$	eC
split ends	aC	aC	aC
remove	L	$m^2 \cdot log_m$	L v.s. eC

We use the notation eC as effective constant time when we have $log_m(n)$ complexities assuming that n will be bounded by the index integer representation and m is large enough. In our case `Int` is a 32-bit signed integer and $m = 32$, giving us the bound $log_m(n) < 7$ and hence argue that this is bounded by a constant. In table 1, aC (amortized constant) has a worst case scenario of log_m or $m \cdot log_m$, in other terms it has is eC in the worst case. In table 2, for the RRB Vector the aC has a worst case of eC, for COW Array aC has linear worst case and for the rest of aC-s have a worst case of log_2. C and L are constant and linear time respectively.

5. Implementation

5.1 Scala Indexed Sequences

Our implementation of the RRB-Vector[8] is based on the Scala Collection [26] `IndexedSeq`, which acts as a decorator exposing many predefined generic operations which are based on just a few core primitives. For example, most operations that involve the entire collection (such as `map`, `filter` and `foreach`) use iterators and builders. To improve performance, we overwrote several of the decorator operations to use the efficient vector primitives directly, without going through the `IndexedSeq` code.

Parallel RRB Vector The implementation of the parallel RRB Vector is a trivial wrapper over the sequential RRB Vector using the Scala Parallel Collections API [25, 33, 34]. This only requires the presence of the split and combine operations, which, in our case, are simply splitting an iterator on the tree and combining using concatenation. Both of these use the efficient core RRB tree operations. When splitting, we additionally have heuristics that yield a well aligned concatenation and create a balanced tree.

5.2 Arrays as Nodes

One of the aims of Scala Collections [25, 26, 33, 34] is the elimination of code duplication, and one of the mechanisms to achieve this

[8] Along with all other sequences we compare against.

Table 2. Comparisons with other data structures that could be used to implement indexed sequences.

	RRB Vector	COW Array	FingerTree	RedBlack Tree
indexing	eC	C	log_2	log_2
scanning	aC	C	aC	aC
update	eC	L	log_2	log_2
update local	aC	L	log_2	log_2
concat/insert	eC	L	log_2	L
insert ends	aC	L	aC	log_2
building	aC	C/aC	aC	log_2
split	eC	L	log_2	log_2
split ends	aC	L	aC	log_2
remove	eC	L	log_2	L

is the use of generic types [9, 22]. But this also has a drawback: the need to box primitive values in order for them to be stored in the collection. We implemented all our sequences in this context.

All nodes are stored in arrays of type `Array[AnyRef]`, since this allows us to quickly access elements (which are boxed anyway due to generics[9]) without dispatching on the primitive array type. A welcome side effect of this decision is that elements are already boxed when they are passed to the Vector, thus accessing and storing them does not incur any intermediate boxing/unboxing operations, which would add overhead. However, it is known that using the boxed representation for primitive types is inefficient when operating on the values themselves, so the sizes of unbalanced nodes are stored in `Array[Int]` objects, guaranteeing the most compact and efficient data representation.

Most of the memory used in the vector data structure will be composed of arrays. There are three key operations used on these arrays: creation, update and access. Since the arrays are used with copy-on-write semantics, actual update operations are only allowed when the array is initialized. This also implies that each time there is a modification on some part of an array, a new array must be created and the old elements must be copied.

The size of the array will affect the performance of the vector. With larger arrays in the nodes the access times will be reduced because the depth of the tree will decrease. But, on the other hand, increasing the size of the arrays will slow down the update operations, as they have to copy the entire array to execute the element update, due to the copy-on-write semantics.

For an individual reference to an RRB-Vector of size n and branching of m, the memory usage will composed by the arrays located in the leaves[10] and the ones that form the tree structure. In our case we save references and hence we need $\lceil \frac{n}{m} \rceil$ arrays of m references[11]. The structure requrires at least the references to the child nodes and in the worst case scenario an additional integer the size of each child. Going up level by level, the reference count decreases by a factor of m and hence the total is bounded by $\sum_{k=2}^{log_m(n)} \lceil \frac{n}{m^k} \rceil < \sum_{k=2}^{\infty} \lceil \frac{n}{m^k} \rceil \leq \frac{n+m}{m \cdot (m-1)}$ refrences. For the sizes of the nodes, given our choice of rebalancing algorithm, they will only appear on nodes that are of height 3 or larger and hence the sizes will be bounded by $\sum_{k=3}^{log_m(n)} \lceil \frac{n}{m^k} \rceil < \sum_{k=3}^{\infty} \lceil \frac{n}{m^k} \rceil \leq \frac{n+m}{m^2 \cdot (m-1)}$ integers.

5.3 Running on a JVM

In practice, Scala compiles to Java bytecode and executes on a Java Virtual Machine (JVM), where we used the Oracle Java SE distribution [29] as a reference. This imposes additional characteristics of performance that can't be evaluated on the algorithmic level alone, and ask for a more nuanced discussion.

One of the JVM components that directly affects vectors is the *garbage collector* (or GC). Vector operations tend to create a large number of `Array` objects, some of which are only necessary for a short time. These objects will use up memory and thus degrade overall performance as the GC is invoked more often. For this reason our code is optimized to avoid the redundant creation of intermediary objects, delaying the GC cycles and thus improving performance.

Instead of directly compiling bytecode to native code, the JVM uses a *just in time compilation* (JIT) mechanism in order to take advantage of run-time profiling information. At first it runs the compiled bytecode inside an interpreter and collects execution statistics (profiles). Later, once a method has executed enough times, it compiles it using the statistics to guide optimizations. The Vector code tries to gain performance by aligning with the JIT heuristics and hence taking advantage of its optimizations. The most important such optimization is inlining, which eliminates the overhead of calling a method and, furthermore, enables other optimizations to improve the inlined code. Critical parts of the Vector code are carefully designed to to match the heuristics of the JVM. In particular, a heuristic that arose commonly is that only methods of size less than 35 bytes are inlined, which meant we had to split the code into several methods to stay below this threshold.

6. Evaluation

6.1 Methodology

ScalaMeter [30] is used to measure performance of operations on different implementations of indexed sequences.

To have reproducible results with low error margins, ScalaMeter was configured on a per benchmark basis. Each test is run on 32 different JVM instances to average out badly allocated VMs. On each JVM, 32 measurements were taken and they were filtered using outlier elimination to remove those runs that where exceptionally different. This could happen if a more thorough garbage collection cycle occurs in a particular run, due to JIT compilation or if the operating system switches to a more important task during the benchmark process [12]. Before taking measurements, the JVM is warmed up by running the benchmark code several times, without taking the measurements into account. This allows us to measure the time after the JIT compilation has occurred, when the system is in a steady state.

There are three main directions in the performance comparisons. The first compares the Radix-Balanced vectors with well-balanced RRB Vectors, with the goal of having an equivalent performance, even if the RRB Vectors have an inherent additional overhead. The second axis shows the effects of unbalanced nodes on RRB-Tree. For this we compare the same perfect balanced vector with an extremely unbalanced vector. The later vector is generated by concatenating pseudo-random small vectors together. The amount of unbalanced nodes is in part affected by the size of the vector. The third axis is the comparison between vectors in general and other well known functional and/or immutable data structures used to implement sequences. We used a copy-on-write (COW) arrays, finger trees [16] (FingerTreeSeq[12]) and red black trees [13] (RedBlackSeq[13]).

6.2 Results

For the results of this sections, 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. The parallel vector split-combine was executed on a machine with 4 Intel(R) Xeon(R) Processors, of type E5-4640 @ 2.40GHz with 128GiB on RAM.

Iterating The benchmark in Figure 11 shows the time it takes to scan the whole sequence using a specialized iterator. Unsurprisingly, the results show that the best option is the array. But the vector is only 1-2× slower, closer to 1× in the most common cases. It is also possible to see that vectors are 7-15× faster than other deeper trees, mainly due to the reduction in indirections and increased locality.

Building The benchmark in Figure 12 shows the time it takes to build a sequence using a specialized builder. In general, the

[9] A limitation that could be circumvented by Miniboxing [40].

[10] Note that the memory used in the leaves is equivalent to the memory used for an array that contains all the elements.

[11] It could be any kind of data.

[12] Adapted version of `https://github.com/Sciss/FingerTree` where abstractions that did not involve sequences where removed.

[13] Adaptation of the standard Scala Collections RedBlackTree where keys are used as indices.

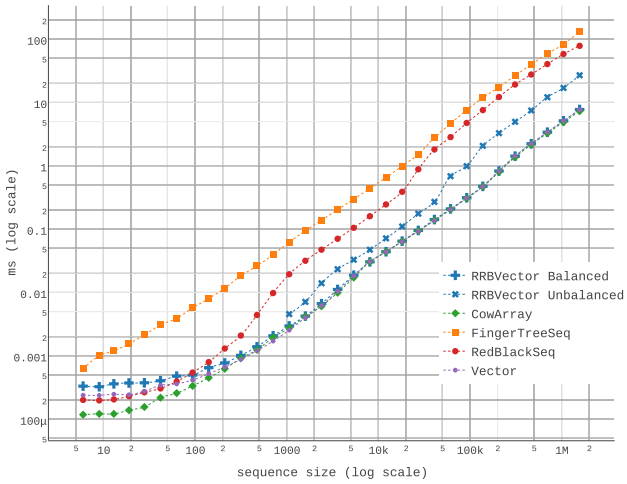

Figure 11. Iterating through the sequence

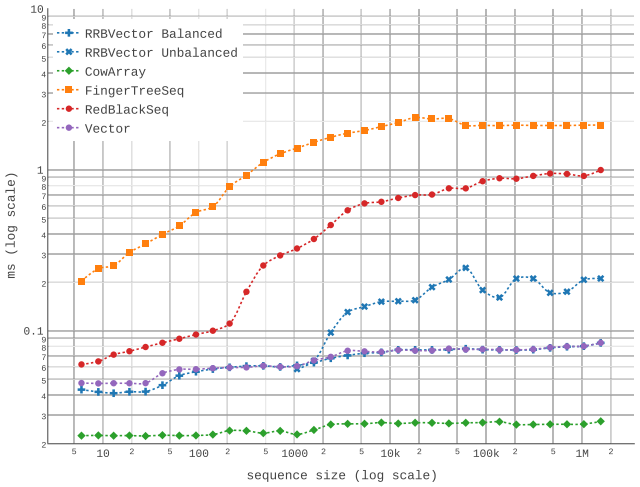

Figure 13. Accaccessing 10k consecutive indices

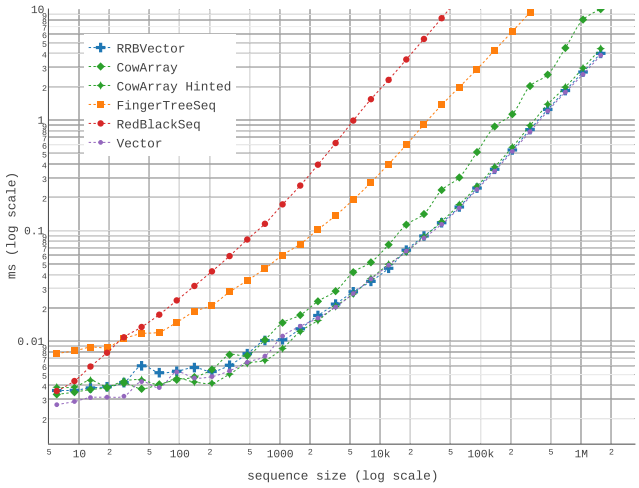

Figure 12. Building a sequence.

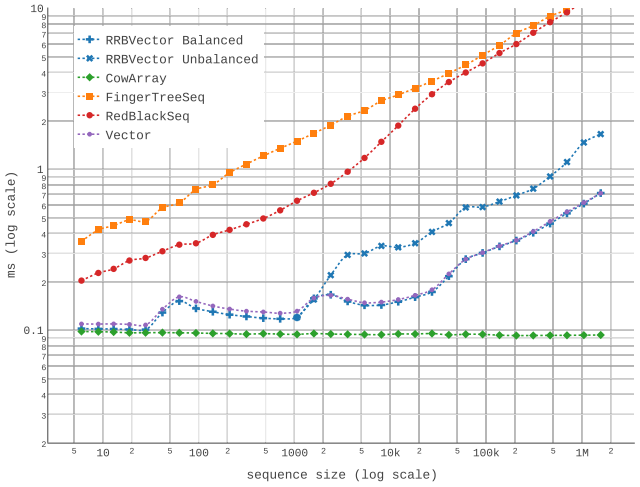

Figure 14. Accessing 10k random indices

builder for these sequences does not know the size of the resulting sequence. In the case of array builder there is the possibility of giving it a hint of the result size (`Hinted` in the benchmarks). In this case the vector wins against all other implementations. It is faster than other trees because they require re-balancing during the building, whereas the vector behaves more like an array building by allocating chunks of memory and filling them. Array building requires resizing of the array whenever it is filled or the result is returned, which implies a copy of the whole array. By contrast, the vector only requires a copy of the last branch when returned. This is the main reason the vector is able to outperform the array building process. Also, the standard array builder uses the hint as such and therefore still requires some copies of the array.

Indexing Figure 13 shows the time taken to access 10k elements in consecutive indices while Figure 14 shows the same for randomly chosen indices. From the algorithmic point of view they are exactly the same, the difference is in how the memory is kept in the processor caches. It shows that in either cases the vector access behaves effectively as constant time like the array, where the finger trees and red black trees degenerate with randomness. A vector of depth 3 is 2-3.5× slower than the array, the cost of accessing the arrays in the 3 levels of the branches.

Updating Figure 15 shows the time taken to update 10k elements in consecutive indices and Figure 16 shows the same for randomly chosen indices. In this case the array is clearly the worst option because it creates a new version and copies the contents with each update. The vector behaves effectively as having constant time while taking advantage of locality and degenerates slightly with randomness. The vector is 4.3× faster on local updates and 1-2.3× faster on random updates than the red black tree.

Concatenating Figures 17 and 18 show the time it takes to concatenate two sequences (two points of view of the same 3D plot). The two axes on the bottom represent the sizes of the LHS (left hand side) and RHS (right hand side) of the concatenation operation. It can be seen that the RRB Vector and finger trees are almost equivalent in performance (bottom planes). The array up to a result size of 4096 is able to concatenate faster thanks to locality, but then grows linearly with the result size (middle plane). The vector without efficient concatenation (on Radix-Balanced trees) behaves just like the array but with worse constant factors (top plane). The red black tree was omitted from this graph due its inefficient concatenation operation.

Appending Figures 19 and 20 show the time it takes to append 256 elements on by one. In the first case we append them

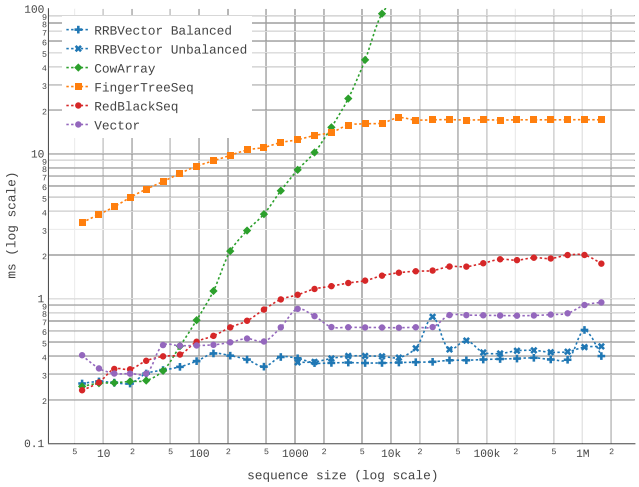

Figure 15. Updating on 10k consecutive indices

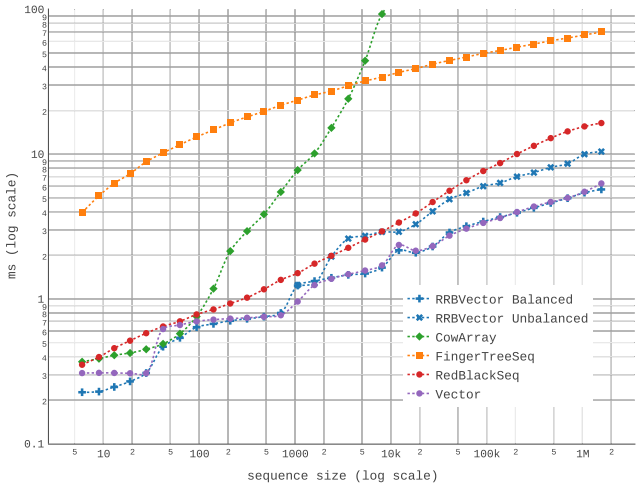

Figure 16. Updating on 10k random indices

to the front and in the second to the back of the sequence. The large number of elements was chosen in order show the amortized time of the operation on the vectors. In this case the array is clearly the worst option because it creates a new version and copies the contents with each append. The vector is around 2.5× slower than the finger trees, a structure that specifically focuses on these operations. The vector can be 1-2× faster than a red black tree.

Splitting Figures 21 and 22 show the time it takes to split a sequence on the left and on the right. We fixed the cut point to the middle of the sequence to be able to compare the time it takes to take or drop the same number of elements. It can be seen that splitting a vector is more efficient than other structures. Even more, the vector behaves with an effectively constant time.

Parallel Vector Split-combine Overhead The benchmarks in Figure 23 and 24 show the amount of overhead associated with the parallelization of the vector with and without efficient concatenation. They show the typical overhead of a parallel `map`, `filter` or other similar operations that create a new version of the sequence. The benchmark computes a `map` operation using the identity function, such that the execution time is dominated by the time it takes to split and combine the sequence rather than the function computations. As a base for comparison we used the sequential `map` on

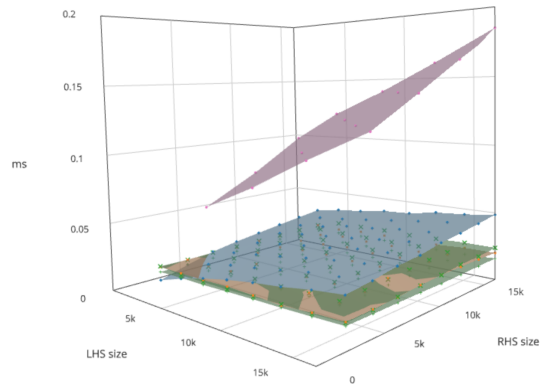

Figure 17. Concatenating two sequences (point of view 1). RRB Vector and Finger Tree are the planes at the bottom, COW Array is the plane in the middle and Vector is the plane on the top.

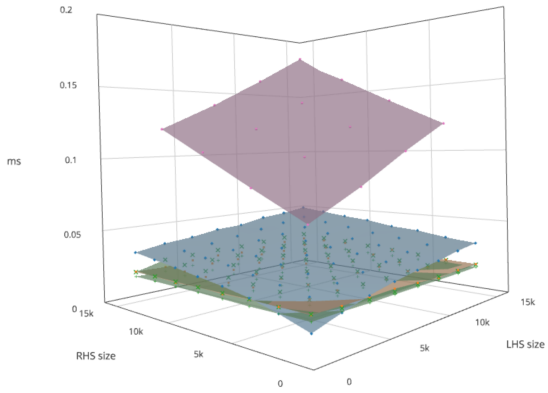

Figure 18. Concatenating two sequences (point of view 2). More informationg on the first point of view on figure 17.

both versions, where the results are identical. Then we parallelized it on fork-join thread pools of 1, 2, 4, 8, 16, 32 and 64 thread on a 64 threaded (32 cores) machine. Without concatenation, there is a loss of performance on when passing from sequential to parallel and although the performance increases with the addition of threads, even with 64 threads it's only slightly better than the sequential version. By contrast, with our new vector, the gain in performance starts with one thread in the pool (dedicated thread) and then increases. Giving a 1.55× increase with 2 threads, 2.46× for 4 thread, 3.52× for 8 thread, 4.60× for 16 thread, 5.52× for 32 thread (core limit) and 7.18× for 64 thread (hardware thread limit).

Memory Overhead Figure 25 shows the memory overhead of the data structures used in the benchmarks. This overhead is the additional space used in relation to the COW Array. The overhead of a vector is 17.5× smaller than the finger tree and 40× smaller than the red black trees.

7. Related Work

Related data structures There is a strong relation between RRB Trees and various data structures in the literature. Patricia tries [24] are one of the earliest documented uses of radix trees, performing lookups and updates one bit or character at a time. Wider radix trees were used to build space efficient sparse arrays, Array Mapped Tries (AMT) [1], and on top of that Hash Array Mapped Tries (HAMT) [2], which have been popularized and adapted to an

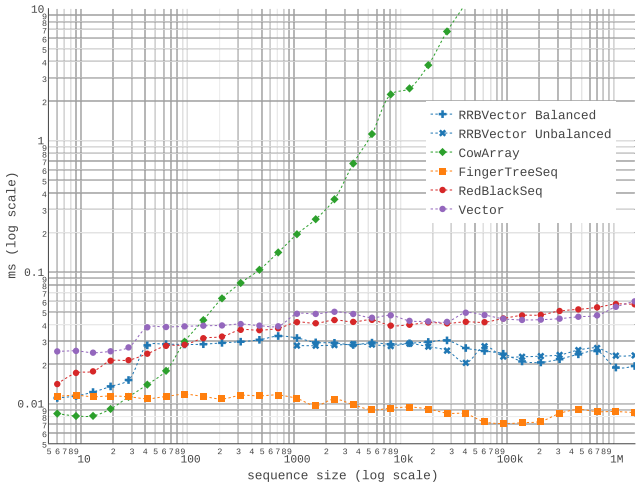

Figure 19. Appending front 256 elements one by one

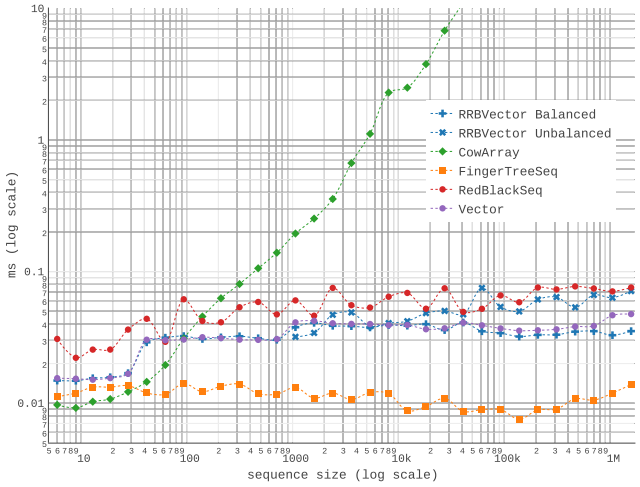

Figure 20. Appending back 256 elements one by one

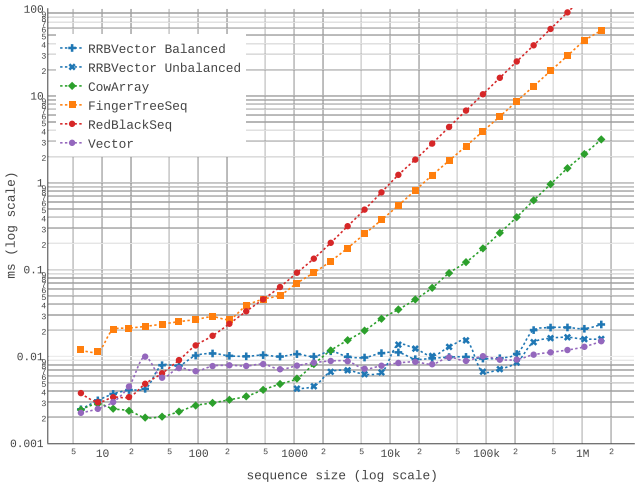

Figure 21. Taking the first half of the sequence

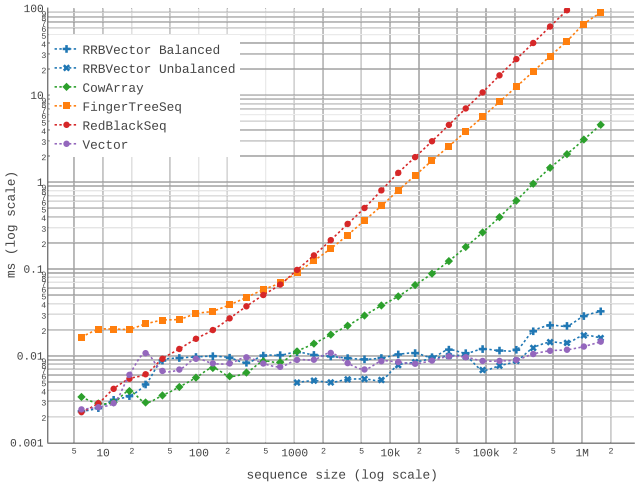

Figure 22. Dropping the first half of the sequence

immutable setting in Clojure [15]. Radix trees are also used as index structures for in-memory databases [19]. In databases B-Trees [6] are a ubiquitous index data structure that are similar to RRB Trees, in the sense that they are wide trees and allow a certain degree of slack to enable efficient merging. However the chosen trade-offs are different and B-Trees are not normally used as sequence data structures. Ropes [5] are a sequence data structure with efficient concat and rebalancing operations. Immutable sequence data structures include VLists [3], Finger Trees [16] and various kinds of random access lists [27].

Parallelism Parallel execution is achieved in the `RRB-Vector` by relying on the fork-join pools in Java [18, 31]. The vector is split into chunks which are processed in parallel. The overhead of splitting can be offset by using cooperative tasks [14, 21], but, in the case of `RB-Vector` the cost of splitting is much smaller compared to the cost of combining (assembling) the partial results returned by parallel execution contexts. This is where the RRB trees make a difference: by allowing efficient structural changes, it enables the concatenation to occur in effectively constant time, much better than the previous $\mathcal{O}(n)$ for the Scala `Vector`.

RRB Trees The core data structure was first described in a technical report [4] as a way to improve the concatenation of immutable vector like the ones found in Scala Collections [26] and Clojure [15]. Allowing the implementation of an additional wide range of efficient structural modification on the vectors. Later, a concrete version for Scala where in all optimization on locality are adapted to RRB vectors was implemented. This is the implementation used in this paper and presented with more technical details in [37] on the implementation in Scala. Another related project is [20], where more detailed mathematical proofs where shown for the RRB Trees and their operations. They also introduce a different approach on transience using semi-mutable vectors with efficient snapshot persistence and provide a C implementation.

Scala Library In Scala, most general purpose data structures are contained in Scala Collections [26]. This framework aims to reduce code duplication to a minimum using polymorphism, higher-order functions [8], higher kinded types [23], implicit parameters [28] and other language features. It also aims to simplify the integration of new data structures with a minimum of effort. In addition, the Scala Parallel Collection API [31–34] allows parallel collections to integrate seamlessly with the rest of the library. Behind the scenes, Scala Parallel Collections use the Java fork-join pools [18] as a backend for implicit parallelism in the data structure operations.

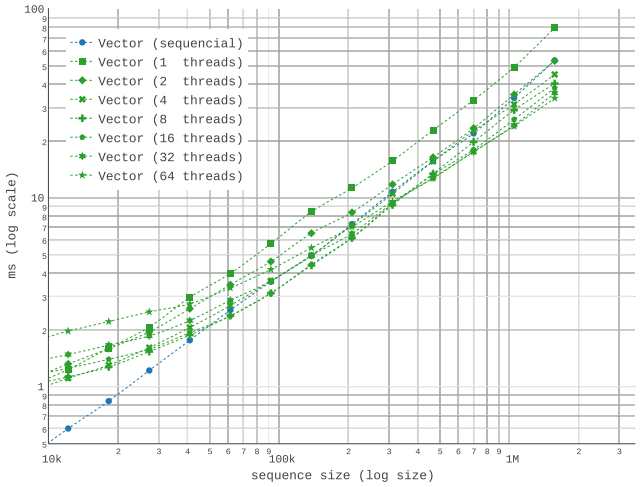

Figure 23. Parallel (non-RRB) Vector overhead on a `map` operation

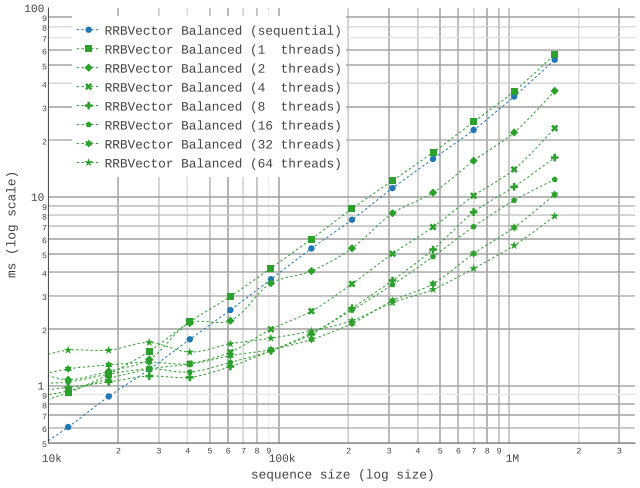

Figure 24. Parallel RRB Vector overhead on a `map` operation

Low level optimizations We took into account the capabilities of the VM to do escape analysis and inlining of the compiled bytecode. These are influenced by the concrete implementation of the Java Hotspot VM compilers (C1 and C2) [17, 29]. At the compiler level there are optimizations techniques that remove the cost of type generic abstractions such as specialization [10], Miniboxing [39, 40] or Scala Blitz [33]. This last one can go further do fusion on collection [7]. Additionally it is possible to use staging techniques [35, 36, 38] to further optimize the code.

Benchmarks Running code on a virtualized environment like the JVM where the factors that influence performance are not under our control and can vary from execution to execution complicates the benchmarking process [12]. In Scala there is a tool (ScalaMeter [41]) designed to overcome these issues.

8. Conclusions

In this paper we presented the `RRB-Vector`, an immutable sequence collection that offers good performance across a broad range of sequential and parallel operations. The underlying innovations are the Relaxed-Radix-Balanced (RRB) Tree structure, which allows efficient structural changes and an optimization that exploits

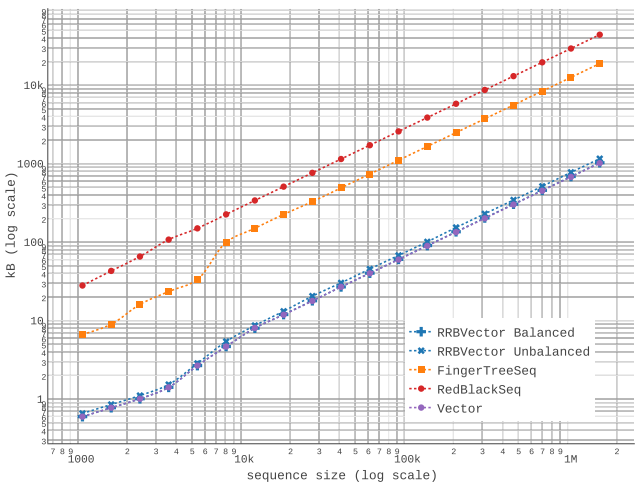

Figure 25. Memory overhead of the sequences in relation to arrays

spatio-temporal locality on the RRB data structure in order to offset the cost of navigating from the tree root to the leaves.

The `RRB-Vector` implementation in Scala speeds up bulk operation performance on 4 cores by at least 2.33× compared to sequential execution, scaling better with light workloads. Discrete operations take either amortized- or effective-constant time, with good constants: compared to mutable arrays, sequential reads are at most 2× slower, while random access is 2-3.5× slower. The implementation of the project is open-source[14] and is being considered for inclusion in the Scala standard library.

Acknowledgements

We would like to thank the ICFP reviewers for their feedback and suggestions, which allowed us to improve the paper both in terms of clarity and in terms of breadth. We are grateful to Martin Odersky for allowing the Master Thesis [37] that forms the base of this paper to be supervised in the LAMP laboratory at EPFL. We would also like to thank Vera Salvisberg for her thorough review of both the paper and the code, which led to remarkable improvements the quality of the final submission. Last but not least, we would like to thank Sébastien Doeraene for allowing Nicolas to dedicate time to improving this paper while he was working on Scala.js.

References

[1] P. Bagwell. Fast and Space-efficient Trie Searches. Technical report, EPFL, 2000.

[2] P. Bagwell. Ideal hash trees. Technical report, EPFL, 2001.

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

[4] P. Bagwell and T. Rompf. RRB-Trees: Efficient Immutable Vectors. Technical report, EPFL, 2011.

[5] H.-J. Boehm, R. Atkinson, and M. Plass. Ropes: An alternative to strings. *Software: Practice and Experience*, 25(12):1315–1330, 1995.

[6] D. Comer. The ubiquitous b-tree. *ACM Comput. Surv.*, 11(2):121–137, 1979.

[7] D. Coutts, R. Leshchinskiy, and D. Stewart. Stream fusion: From lists to streams to nothing at all. In *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming*, ICFP '07, pages 315–326, New York, NY, USA, 2007. ACM.

[8] I. Dragos. Optimizing Higher-Order Functions in Scala. In *ICOOOLPS*, 2008.

[14] `https://github.com/nicolasstucki/scala-rrb-vector`

[9] I. Dragos. *Compiling Scala for Performance*. PhD thesis, IC, 2010.

[10] I. Dragos and M. Odersky. Compiling Generics through User-directed Type Specialization. In *ICOO0LPS '09*. ACM, 2009.

[11] J. R. Driscoll, N. Sarnak, D. D. Sleator, and R. E. Tarjan. Making data structures persistent. *J. Comput. Syst. Sci.*, 38(1):86–124, Feb. 1989.

[12] A. Georges, D. Buytaert, and L. Eeckhout. Statistically rigorous java performance evaluation. In *Proceedings of the 22Nd Annual ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications*, OOPSLA '07, pages 57–76, New York, NY, USA, 2007. ACM.

[13] S. Hanke. The Performance of Concurrent Red-Black Tree Algorithms. In J. Vitter and C. Zaroliagis, editors, *Algorithm Engineering*, volume 1668 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 1999.

[14] M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming*. Apr. 2008.

[15] R. Hickey. The Clojure programming language, 2006.

[16] R. Hinze and R. Paterson. Finger Trees: A Simple General-purpose Data Structure. *J. Funct. Program.*, 16(2), 2006.

[17] T. Kotzmann, C. Wimmer, H. Mössenböck, T. Rodriguez, K. Russell, and D. Cox. Design of the Java HotSpot&Trade; Client Compiler for Java 6. *ACM Trans. Archit. Code Optim.*, 5(1), May 2008.

[18] D. Lea. A Java Fork/Join Framework. In *Proceedings of the ACM 2000 Conference on Java Grande*, JAVA '00, New York, NY, USA, 2000. ACM.

[19] V. Leis, A. Kemper, and T. Neumann. The adaptive radix tree: Artful indexing for main-memory databases. In C. S. Jensen, C. M. Jermaine, and X. Zhou, editors, *29th IEEE International Conference on Data Engineering, ICDE 2013, Brisbane, Australia, April 8-12, 2013*, pages 38–49. IEEE Computer Society, 2013.

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

[21] Moir and Shavit. Concurrent data structures. In Mehta and Sahni, editors, *Handbook of Data Structures and Applications, Chapman & Hall/CRC*. 2005.

[22] A. Moors. *Type Constructor Polymorphism for Scala: Theory and Practice (Type constructor polymorfisme voor Scala: theorie en praktijk)*. PhD thesis, Informatics Section, Department of Computer Science, Faculty of Engineering Science, May 2009. Joosen, Wouter and Piessens, Frank (supervisors).

[23] A. Moors, F. Piessens, and M. Odersky. Generics of a Higher Kind. *Acm Sigplan Notices*, 43, 2008.

[24] D. R. Morrison. PATRICIA-practical algorithm to retrieve information coded in alphanumeric. *J. ACM*, 15(4):514–534, Oct. 1968.

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

[26] M. Odersky and A. Moors. Fighting bit Rot with Types (Experience Report: Scala Collections). In R. Kannan and K. N. Kumar, editors, *IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science*, volume 4 of *Leibniz International Proceedings in Informatics (LIPIcs)*, Dagstuhl, Germany, 2009. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[27] C. Okasaki. *Purely Functional Data Structures*. Cambridge University Press, New York, NY, USA, 1998.

[28] B. C. d. S. Oliveira, A. Moors, and M. Odersky. Type Classes as Objects and Implicits. In *OOPSLA '10*. ACM, 2010.

[29] M. Paleczny, C. Vick, and C. Click. The java hotspottm server compiler. In *Proceedings of the 2001 Symposium on JavaTM Virtual Machine Research and Technology Symposium - Volume 1*, JVM'01, pages 1–1, Berkeley, CA, USA, 2001. USENIX Association.

[30] A. Prokopec. ScalaMeter. `https://scalameter.github.io/`.

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

[32] A. Prokopec and M. Odersky. Near optimal work-stealing tree scheduler for highly irregular data-parallel workloads. In *Languages and Compilers for Parallel Computing*, Lecture Notes in Computer Science, pages 55–86. Springer International Publishing, 2014.

[33] A. Prokopec, D. Petrashko, and M. Odersky. Efficient Lock-Free Work-stealing Iterators for Data-Parallel Collections. 2015.

[34] A. Prokopec, T. Rompf, P. Bagwell, and M. Odersky. On a generic parallel collection framework, 2011.

[35] T. Rompf and M. Odersky. Lightweight Modular Staging: A Pragmatic Approach to Runtime Code Generation and Compiled DSLs. *Communications Of The Acm*, 55, 2012.

[36] T. Rompf, A. K. Sujeeth, K. J. Brown, H. Lee, H. Chafi, and K. Olukotun. Surgical precision JIT compilers. In M. F. P. O'Boyle and K. Pingali, editors, *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, page 8. ACM, 2014.

[37] N. Stucki. Turning Relaxed Radix Balanced Vector from Theory into Practice for scala collections. Master's thesis, EPFL, 2015.

[38] W. Taha and T. Sheard. MetaML and Multi-Stage Programming with Explicit Annotations. In *Theoretical Computer Science*. ACM Press, 1999.

[39] V. Ureche, E. Burmako, and M. Odersky. Late data layout: Unifying data representation transformations. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '14, pages 397–416, New York, NY, USA, 2014. ACM.

[40] V. Ureche, C. Talau, and M. Odersky. Miniboxing: Improving the Speed to Code Size Tradeoff in Parametric Polymorphism Translations. In *OOPSLA'13*, OOPSLA '13, pages 73–92, New York, NY, USA, 2013. ACM.

[41] B. Venner, G. Berger, and C. C. Seng. Scalatest. `http://www.scalatest.org/`.