Persistence for the Masses: RRB-Vectors in a Systems Language

JUAN PEDRO BOLÍVAR PUENTE

Relaxed Radix Balanced Trees (RRB-Trees) is one of the latest members in a family of persistent tree based data-structures that combine wide branching factors with simple and relatively flat structures. Like the battle-tested immutable sequences of Clojure and Scala, they have effectively constant lookup and updates, good cache utilization, but also logarithmic concatenation and slicing. Our goal is to bring the benefits of functional data structures to the discipline of systems programming via generic yet efficient immutable vectors supporting transient batch updates. We describe a C++ implementation that can be integrated in the runtime of higher level languages with a C core (Lisps like Guile or Racket, but also Python or Ruby), thus widening the access to these persistent data structures.

In this work we propose (1) an *Embedding RRB-Tree* (ERRB-Tree) data structure that efficiently stores arbitrary unboxed types, (2) a technique for implementing tree operations independent of optimizations for a more compact representation of the tree, (3) a *policy-based* design to support multiple memory management and reclamation mechanisms (including automatic garbage collection and reference counting), (4) a model of *transience* based on move-semantics and reference counting, and (5) a definition of transience for confluent *meld* operations. Combining these techniques a performance comparable to that of mutable arrays can be achieved in many situations, while using the data structure in a functional way.

CCS Concepts: • **Software and its engineering → Data types and structures**; *Functional languages*; • **General and reference** → Performance;

Additional Key Words and Phrases: Data Structures, Immutable, Confluently, Persistent, Vectors, Radix-Balanced, Transients, Memory Management, Garbage Collection, Reference Counting, Design Patterns, Policy-Based Design, Move Semantics, C++

ACM Reference Format:
Juan Pedro Bolívar Puente. 2017. Persistence for the Masses: RRB-Vectors in a Systems Language. *Proc. ACM Program. Lang.* 1, ICFP, Article 16 (September 2017), 28 pages.
https://doi.org/10.1145/3110260

1 INTRODUCTION

Immutability enables safe lock-free communication between concurrent programs. *Persistence* facilitates reasoning about change and is a fundamental to higher level reactive and interactive systems. A growing development community is increasingly interested in these properties, motivated by the horizontal scaling of our computing power, and the increased expectations that a wider and more diverse consumer base are putting on user interfaces. Many traditionally object-oriented programming languages are turning to a multi-paradigm approach embracing core functional programming concepts. It is the data structures that enable immutability and persistence at the scale of real world production systems.

2475-1421/2017/9-ART16
https://doi.org/10.1145/3110260

1.1 Challenge

Implementations of persistent data structures exist for various languages, both functional and otherwise. However, few attempts have been made to implement them in a language without a managed runtime or pervasive garbage collection. There are good motivations to try though. First, the systems programming community is adopting many techniques and principles from functional programming, as shown by Rust [Matsakis and Klock 2014] and the latest developments in recent C++ standards. Second, a sufficiently general and efficient implementation could be integrated in the runtime of higher level languages (Lisps like Guile or Racket, but also Python or Ruby come to mind), allowing a wider audience to enjoy the benefits of these data structures. Doing so poses various challenges.

(1) Efficient persistent data structures require garbage collection. Without automatic garbage collection provided by the runtime, a reference counting reclamation scheme may be used. Doing so efficiently is challenging. Furthermore, when integrated in a runtime, it should be possible to leverage the garbage collector it provides, if one exists.
(2) Most immutable data structures are designed to store boxed values—i.e. pointers to objects allocated in the free store.[1] Many performance critical applications require embedding the values in the data structure for further cache locality.
(3) An immutable interface may not always interact well with other components of a language that is not fundamentally functional. Furthermore, performance oriented systems developers might want to *temporarily* escape immutability when implementing transactional batches of updates to a data structure.

1.2 Contributions

We describe an implementation of RRB-Tree based vectors with transience in C++. We overcome these challenges making the following contributions:

(1) The *Embedding RRB-Tree* (ERRB-Tree) data structure that efficiently stores arbitrary unboxed types (§ 3).
(2) A tree traversal technique based on mutually recursive higher order position/visitors. It can be used to implement tree operations independently of optimizations that achieve a more compact representation of the tree (§ 4).
(3) A *policy-based* design to support multiple memory management and reclamation mechanisms (including tracing garbage collection and reference counting) (§ 5).
(4) A model of *transience* based on reference counting (§ 6.2) and move-semantics (§ 6.3). These optimize updates by sometimes performing them in-place even when the containers are used in a functional style, making the performance profile is depend on the actual dynamic amount of persistence in the system.
(5) A definition of transience for all RRB-tree operations, including confluent *meld* operations (§ 6.4).
(6) An evaluation of the state of the art of Radix Balanced vectors, comparing it to various configurations of our implementation. This includes a discussion of the effects of reference counting, challenging the established assumptions on its suitability for the implementation of immutable data structures (§ 7).

Our implementation is libre software and freely available online.[2]

[1] Some implementations do embed basic numeric types, but since they have sizes similar to that of a pointer, they do not need to adapt the core algorithms.
[2] Immer: https://sinusoid.es/immer

1.3 Related Work

1.3.1 Persistent Data Structures. Persistent data structures were originally studied as mutable data structures [Driscoll et al. 1986; Kaplan 2005]. Such data structures where the contents of two or more values can be merged in sub-linear time and space by sharing structure with all inputs are called *confluently persistent* [Collette et al. 2012; Fiat and Kaplan 2001]. Persistent data structures gained special momentum with Okasaki's [1999] seminal work on purely functional data structures. These are immutable and achieve efficient persistence via *structural sharing*. However, these have not reached mainstream programmers because of their small-node structures amortized via lazy evaluation, making them hard to implement efficiently in popular languages and modern architectures.

1.3.2 Cache Efficient Immutability. Phil Bagwell has done extensive work in building data structures with efficient cache utilization, including VLists [2002], Array Mapped Tries [2000] and Hash Tries [2001]. The latter two inspired the foundations of Clojure's [Hickey 2008] ubiquitous immutable vectors and maps. These quickly gained popular attention and various implementations are currently used in production in Scala,[3] JavaScript,[4] Python,[5] and Java.[6] [Steindorfer and Vinju 2015, 2016]

1.3.3 Relaxed Radix Balanced Trees (RRB-Trees). Bagwell and Rompf [2011] first proposed relaxing the Radix Balanced Tree structure in order to make the vectors from Clojure and Scala confluently persistent supporting $O(log(n))$ concatenation. L'orange's [2014] masters thesis introduced *transience* for some operations, except concatenation and slicing, and it is one of the most thorough and intelligible descriptions of the inner workings of this data structure. Stucki et al. [2015] further developed the data structure adding amortized $O(1)$ updates via *displays*. RRB-Trees are often compared to other catenable sequences with efficient random access, such as Ropes [Boehm et al. 1995][7], Finger Trees [Hinze and Paterson 2006], and Chunked Sequences [Acar et al. 2014].

1.3.4 RRB-Trees in Systems Programming. L'orange [2014] implemented RRB-Trees in C, but they rely on Boehm and Weiser's [1988] conservative garbage collector and do not support unboxed types nor confluent transience. In parallel to this work, two other implementations of immutable vectors in C++ have been published.[8] These track garbage using atomic reference counts but do not support relaxed trees, transients, efficient embedding or customizable memory management.

2 BACKGROUND

2.1 Radix Balanced Trees

Radix Balanced Trees are inspired by Array Mapped Tries [Bagwell 2000] and were first introduced in the implementation of immutable vectors in Clojure [Hickey 2008]. They provide a good balance between structural sharing, random access complexity, and efficient cache utilization.

[3]Scala Standard Immutable Collection Library: http://www.scala-lang.org/docu/files/collections-api
[4]Immutable.js: https://github.com/facebook/immutable-js, Mori: https://github.com/swannodette/mori, Collectable: https://github.com/frptools/collectable
[5]Pyrsistent: https://github.com/tobgu/pyrsistent
[6]Cyclops: https://github.com/aol/cyclops-react/
[7]Sean Parent claims in his talk "Inheritance Is The Base Class Of All Evil" that Ropes are used inside Photoshop to make their data model persistent. That talk has drawn the attention of a big part of the C++ community towards immutability and value semantics, and is one source of inspiration for this work. https://channel9.msdn.com/Events/GoingNative/2013/Inheritance-Is-The-Base-Class-of-Evil
[8]Steady: https://github.com/marcusz/steady, Immutable++ https://github.com/rsms/immutable-cpp

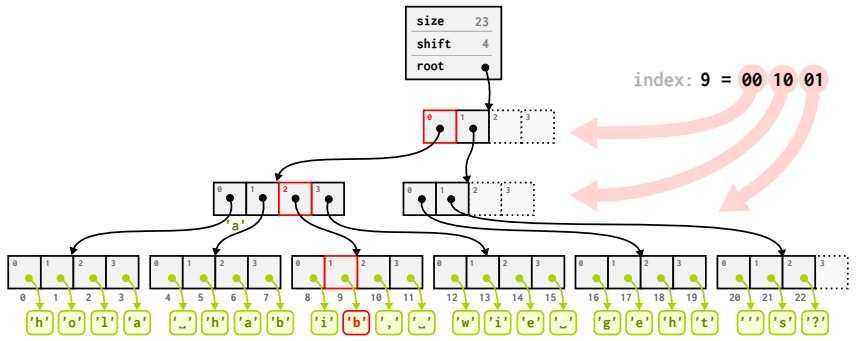


Fig. 1. A $B = 2$ Radix Balanced Tree containing the 23 element string: `"hola␣habib,␣wie␣geht's?"`. The access path down to index 9 is shown in red.

As illustrated in figure 1, this is a tree structure where every inner node and leaf has $M = 2^B$ slots (branches or elements respectively), where B are the *branching bits* that characterize the tree. The rightmost path of the tree may contain nodes with less than M slots. Every other node has exactly M slots and is considered *full*.

2.1.1 Radix Search. We may locate the vector element at index i in a radix balanced tree t by traversing down the tree, where for every subtree t' with height $h(t')$, we descend to its child at offset $\left\lfloor i/M^{h(t')} \right\rfloor \bmod M$ (*radix search*). Since M is a power of two, we can simply divide the vector element index in $h(t)$ groups of B bits and use each to navigate at each level of the tree. A common optimization is to keep track of the value $\text{shift}(t) = B \times h(t)$ by storing it at the root. This denotes the depth of the tree while avoiding multiplications in tree traversals, using comparatively cheap bit-wise operations instead. Interestingly, an indexing mechanism very similar to radix search is commonly used to map virtual memory addresses to hardware addresses using multi level page tables [Drepper 2008].[9]

A Radix Balanced Tree can also be considered a *trie*. If we think of numbers represented in base M as strings with an alphabet of size M, an immutable vector of size n is a trie containing every key in the range $[0, n)$ if we add left padding to the keys so they are evenly sized. This trie has a depth $h(T) = \log_M(n)$ and thus lookup runs in logarithmic time.

2.1.2 Branching Factor. We partially reproduced the results from [Bagwell and Rompf 2011; Hickey 2008; L'orange 2014; Stucki et al. 2015] claiming that $M = 32$ ($B = 5$) is a sensible choice on modern architectures.[10] With such high branching factor elements are stored in contiguous blocks spanning a few cache lines and thus the CPU caches are used effectively and the structure can be iterated fast. Furthermore, such tree containing 2^{32} has only depth of 7, this is, it contains every element that is addressable using 32 bit integers yet search elements in only 7 steps. When dealing with lots of data, other factors (like working set size vs cache size) have an impact orders of magnitude larger than the depth of the tree. For this reason, in practice, it is useful to think

[9] In fact, the way Linux implements virtual memory via copy-on-write can be thought of as massive persistent vector, where memory pages are leaf nodes, and page tables are inner nodes, and each `fork()` creates a new version of the data structure. So we have the operating system managing memory as massive persistent vectors at one end, and at the other end language users build their programs out of small persistent vectors and hash-tables. We could fill the sandwich adding a persistent vector based memory allocator/garbage collector, fulfilling the wild fantasies of the authors—maybe a future work proposition?

[10] Informal experiments using our implementation show that the values 4, 5 or 6 all sensible, having different impact on different operations. The optimal choice depends on the particular hardware, workload, and memory management strategy.

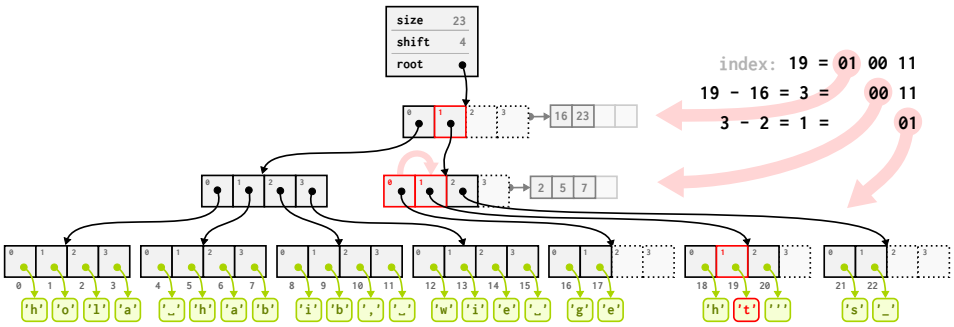


Fig. 2. A $B = 2$ RRB-Tree containing the 23 element string: `"hola␣habib,␣wie␣geht's?"`. The access path down to index 19 is shown in red. At level 2, an extra step is taken in order to find the right subtree.

of radix search complexity as constant. As such, it is often advertised that Radix Balanced based vectors support *effectively constant* random access.

2.2 Relaxed Radix Balanced Trees

Regular Radix Balanced Trees provide effectively constant random access, updates, appends, and right slicing (`take`). However concatenation is proportional to the size of the right argument, and left slicing (`drop`) is proportional to the number of remaining elements. This is caused by the strict balancing required for radix search. By *relaxing* this requirement, Relaxed Radix Balanced Trees (RRB-Trees) support $O(log(n))$ concatenation, left-slicing, and insertion at random locations [Bagwell and Rompf 2011; L'orange 2014; Stucki et al. 2015].

Regular Radix Balanced Trees are a strict subset of valid RRB-Trees. RRB-Trees introduce a new kind of node, *relaxed inner nodes*, whose children are allowed to contain less elements than those required by radix search. These nodes have an array with the accumulated sizes of its subtrees as illustrated in figure 2.

Relaxed Radix Search sometimes needs to step through the size array to look up the actual subtree containing the searched element. When it descends to a regular subtree, it falls back to regular radix search—regular subtrees can not contain relaxed subtrees. A rebalancing concatenation operation can be defined in $O(\log(n))$ that bounds the amount of extra search steps required such that random access remains effectively constant in the resulting structure (albeit with a higher constant).

This data structure provides a very good compromise for big vectors in which insertions at arbitrary positions are frequent. It also opens new opportunities for concurrent and parallel programming. Operations that change the size of the vector, such as *filtering*, can be parallelized by processing the vector in chunks and then concatenating the results [Prokopec 2014; Stucki et al. 2015].

2.3 Optimizations

2.3.1 Transience. By default, persistence is achieved in RRB-Trees via immutability. Because data is never modified in-place, updates involve copying the whole path down to the updated element. While this is desired most of the time, it is overkill when a function produces many intermediate versions that are immediately discarded.

Clojure proposes a pair of polymorphic functions (`transient v`) and (`persistent! t`). The first one returns a mutable (i.e. *transient*) version of the immutable collection `v`, for which mutating operations do exist. The second returns an immutable snapshot of the transient collection `t`

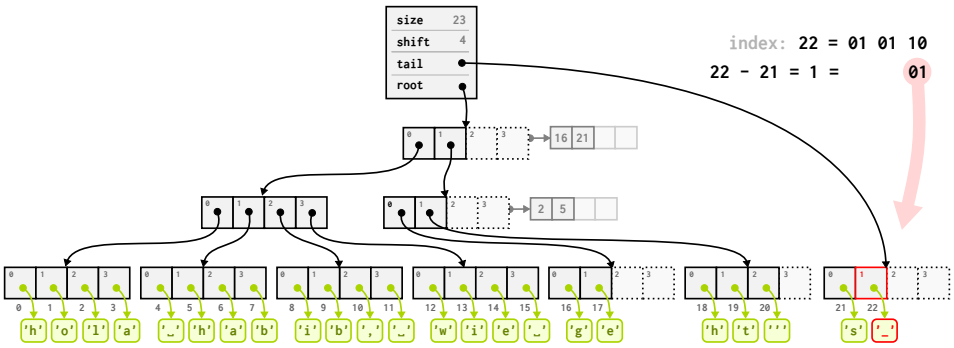


Fig. 3. A $B = 2$ RRB-Tree with off-tree tail containing the 23 element string: `"hola␣habib,␣wie␣geht's?"`. The access path down to index 22 is shown in red. Note that since the element lies in the tail, the tree does not need to be traversed.

and invalidates `t`. Both operations are $O(1)$. Updates on a transient collection are done in-place whenever possible, doing copy-on-write to ensure the immutability of the adopted contents. We further discuss transience in § 6.

2.3.2 Off-tree Tail. When using immutable *vectors*, users may expect fast appends, as it is the case for mutable vectors. The most common optimization when implementing Radix Balanced Trees is to keep the rightmost leaf off-tree and point to it directly from the root, as shown in figure 3. In most cases only the tail needs to be updated to append a new element. Once every M appends the tail gets full—it is inserted into the tree and a new tail is created. Appends become significantly faster this way and our implementation includes this optimization.

2.3.3 Left Shadow. We saw that the rightmost path may contain nodes with less than M slots. A way to think about this is to consider that every Radix Balanced Tree t does virtually contain $M^{h(t)}$ elements. When the *size* of the vector is not a power of M, the rightmost $M^{h(t)} - size$ elements are `null`. The representation is *compressed* by not storing all these empty rightmost branches in memory. This is the *right shadow* of the tree.

Instead of projecting our vector elements in the indices $[0, size)$ of the tree, we could as well use some other $[first, last)$ range, thus creating another shadow on the left. In this way, it is possible to implement effectively $O(1)$ `drop` and prepends with regular Radix Balanced Trees. This optimization is implemented in Scala [Stucki et al. 2015]. We chose not to implement this optimization, but we are considering it for future work.

2.3.4 Display. The *display* is a generalization of the off-tree tail mechanism that leverages the spatiotemporal locality of updates. The core idea is to establish a vector element as the *focus*. The whole path down to this index (the *display*) is kept off-tree and stored directly at the root. It was found that by using an exclusive-or (`xor`) operation between two indexes it is easy to find their common ancestor in the tree. Updates close to the focus become faster because only the sub-path after the common ancestor needs to be updated. Vector manipulations change the focus to the index where the update occurs. In this way, sequential updates become *amortized* $O(1)$. We experimented with this optimization for a while, but decided not to include it in our final implementation, because:

(1) The root node becomes bigger because it stores a whole path. This means that if we store it on the stack or unboxed in some container, it becomes more expensive to copy it: both

because of the bigger size, but also because the reference counts of all nodes in the display need to be touched.
(2) Implementation complexity is increased. Some assumptions are invalidated (e.g. now some references inside the tree are `null` when they belong to the display) adding further conditional checks and overhead to operations that do not benefit from the display.
(3) Transients are an alternative way of improving the peformance of sequential updates. While less general and *pure*, they provide better performance when applicable (§ 7.2.4).

3 EMBEDDING UNBOXED VALUES

3.1 Problems with Boxed Values

All evaluated RRB-Tree literature and implementations assume boxed values.[11] This is, the container does not manage the memory where the actual element values are stored. Instead, the values are located in separate objects (i.e. *boxes*) in the free store. The leaf nodes of the RRB-Tree just store *pointers* (i.e. references) to these objects, not the object themselves. This degrades performance in many situations:

(1) The elements are stored in separate objects that are potentially distant in memory. The extra indirection and the lack of locality result in suboptimal usage of the CPU caches, causing slower accesses.
(2) Every free store object requires extra memory usage, because of the extra pointer, but also because of the cost of boxing itself.[12]

3.2 Naive Unboxing

In system languages supporting manual memory management, containers often embed the contained values directly in the data structure, thus tackling the problems above.

A naive way of embedding values in RRB-Trees would be to store the contained objects directly at the leaves. In C++ we could represent these leaves as standard arrays—i.e. `T[M]` or `std::array<T, M>` for an element type `T`. However, this approach has one fundamental flaw: the size of the leaf nodes is now proportional to the size of `T` and normally different from the size of the inner nodes.

The established result that $B = 5$ provides efficient updates does no longer hold. It does not suffice to compute an alternative better B for each different `sizeof(T)`. Note that for user defined types, the size of T is unbounded—they may write types as big as they want. For bigger types, we may try to compensate lowering the branching factor. But doing so we also increase the depth of the tree, damaging random access performance. For very big types, we end up with the degenerate case $B = 1$ (a binary tree) where the benefits of the RRB search are completely lost.

3.3 Embedding Radix Balanced Trees

We propose a variation called *Embedding (Relaxed) Radix Balanced Trees* or ERRB-Trees.

3.3.1 Definition. An ERRB-Tree is characterized by *two* constants, the *branching bits* B and the *leaf branching bits* B_l, that relate to the branching factors $M = 2^B$ and $M_l = 2^{B_l}$ respectively. Its structure is similar to that of a standard RRB-Tree, but while inner nodes contain up to M slots, leaf nodes now contain up to M_l slots. Figure 4 shows an example of such tree and how addressing works in this case.

[11] Some implementations, like Clojure, have partial support for unboxed primitive types. However, in most architectures the size of primitive types is similar to the size of pointers, thus not rising the problems discussed here.
[12] Depending on the implementation, every free store object might have associated free list pointers, object size, GC mark/lock flags, padding due to bucketing or fragmentation, etc.

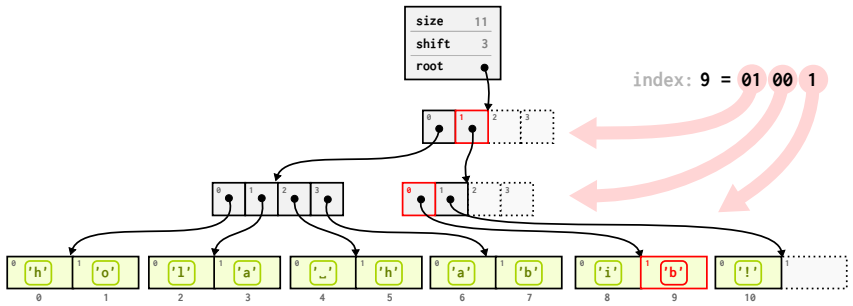


Fig. 4. A $B = 2, B_l = 1$ ERB-Tree containing the 11 element string: "hola␣habib!". The access path down to index 9 is shown in red. Every element is twice the size of a pointer (we could imagine we are storing a UTF-32 string in a 16-bit architecture) and the contained objects are embedded directly in the leaves.

Under the new structure, looking for vector element at index i, the offset in the leaf array is $\lfloor i/M_l \rfloor$. Thus, the shift(t) is now defined as:

$$\text{shift}(t) = \begin{cases} B_l + B \times (h(t) - 1) & \text{if } h(t) > 0 \\ 0 & \text{if } h(t) = 0 \end{cases} \tag{1}$$

Special care has to be taken to accomodate the base case of recursive tree traversals to this new *shift* definition. Most algorithms are otherwise very similar to those of the original RRB-Trees. Listings 3.1a and 3.1b compare potential C++ definitions of naive and embedding RRB-Trees and their respective random access operations.

3.3.2 Choosing B_l. The question remains: what are the best values for B and B_l? Intuitively, we expect the answer to depend on $sizeof_T$. Experimentally we can show that $B = 5$ remains valid, but B_l should be chosen such that leaf nodes are similar in size to inner nodes. The size of inner nodes depends only on B, since the size of a data pointer ($sizeof_*$) is usually fixed for a given CPU architecture. For a given B and T, we may derive the branching bits at the leaves as:

$$B'_l(B, T) = \left\lfloor \log_2 \left(\frac{sizeof_* \times 2^B}{sizeof_T} \right) \right\rfloor \tag{2}$$

Note that the choice to floor the result might seem rather arbitrary. This ensures that $sizeof_T \times 2^{B_l} \leq sizeof_* \times 2^B$, this is, that leaf nodes are at most as big as inner nodes. This property enables reusing buffers used to store all kinds of vector nodes across different value types (§ 5.3).

Also, we see that $B'_l = 0$ when $sizeof_T > sizeof_* \times 2^{B-1}$. In this case, leaf nodes contain only one element and no bits are used to address into it. Such ERRB-Tree is equivalent to a RRB-Tree with the same B (off-tail optimizations not considered).

In our C++ implementation, both B and B_l can be customized by passing template arguments. This allows the user to optimize the data structure for architectures not considered by the authors, or to reproduce the results here described. Otherwise, they default to $B = 5$ and $B_l = B'_l(B, T)$. The latter is derived at compile time when the containers is instantiated for a given element type T.

4 MINIMIZING METADATA

4.1 Incidental Metadata

A naive implementation of RRB-Trees stores the following two pieces of metadata in every node:

```
template <int B> constexpr auto M = 1u << B;
template <int B> constexpr auto mask = M<B> - 1;

template <typename T, int B>
struct rbtree {
  // ...
  const T& operator[] (size_t i) const {
    return get_(i, root, shift);
  }

private:
  union node {
    node* inner[M<B>];
    T leaf[M<B>];
  };

  node* root;
  int shift;
  size_t size;

  const T& get_(size_t i, const node* n, int s) const {
    return s == 0
      ? n->leaf[i & mask<B>]
      : get_(i, n->inner[(i >> s) & mask<B>], s - B);
  }
};
```

(a) Naive Radix Balanced Tree

```
template <int B> constexpr auto M = 1u << B;
template <int B> constexpr auto mask = M<B> - 1;

template <typename T, int B, int BL>
struct erbtree {
  // ...
  const T& operator[] (size_t i) const {
    return get_(i, root, shift);
  }

private:
  union node {
    node* inner[M<B>];
    T leaf[M<BL>];
  };

  node* root;
  int shift;
  size_t size;

  const T& get_(size_t i, const node* n, int s) const {
    return s == BL - B
      ? n->leaf[i & mask<BL>]
      : get_(i, n->inner[(i >> s) & mask<B>], s - B);
  }
};
```

(b) Embedding Radix Balanced Tree

Listing 3.1. A potential definition Naive and Embedding RRB-Trees

(1) The *type* of node. This is, whether it is a leaf or inner node (regular or relaxed). This may happen implicitly—i.e. it is added by the compiler when using runtime polymorphism, in the form of a pointer to a v-table, type tag, or alike.
(2) The *number of slots* in the node. In many languages this is added implicitly too—for example, Java arrays provide an `length` member.[13]

4.2 Fundamental Metadata

For relaxed inner nodes, some kind of metadata is unavoidable. This is so because (1) they may have a variable number of sub-nodes, and (2) they have to be distinguished from regular inner nodes. An inner node thus has a member that may point to an object containing the number of children and the cumulative size of each subtree. This pointer is `null` in regular inner nodes. In most implementations, this pointer is stored at the end of the children array. We store it before the children array—experimentally we saw that it makes no significant performance difference[14] but it makes the position fixed and independent of the number of elements.

4.3 Removing Incidental Metadata

4.3.1 Type Information. No other information about the type of node is fundamentally required, because we can distinguish inner from leaf nodes by their distance to the root relative to the *shift*. But as long as no type metadata exists, *nodes do not know how to destroy themselves*. This means that

[13]Whether this adds an overhead depends on the implementation. Very often, the free store needs to know the size of the node anyways, but this information is lost in abstraction in languages like C or C++.
[14]Has a tiny advantage for some operations, a tiny disadvantage for others.

no generic reference counting mechanism can be used.[15] However, by doing reference counting manually as part of the algorithms, we achieve further performance gains by avoiding redundant operations.

4.3.2 Slot Counts. Most implementations still do store the number of slots either directly or indirectly. In most cases, they use the array `length`. When transients are supported, mutable nodes keep room for extra slots, so the `length` attribute is not accurate. In that case, implementations may set the extra slots to `null`. But both the array length and `null` markers are redundant to the extent that we can derive the number of slots in a node from other information.

Remember that following radix search (§ 2.1.1), we may define a function that, for a given height h, computes the *offset* of the slot containing the vector index i:

$$\text{offset}(i, h) = \left\lfloor \frac{i}{M^h} \right\rfloor \bmod M \tag{3}$$

In a regular tree of size $s > 0$, we can then derive the number of slots in a node t at height $h(t)$ that contains the vector index i:

$$\text{slots}(t) = \begin{cases} M & \text{if } i < M^{h(t)} \times \text{offset}(s-1, h(t)+1) \\ \text{offset}(s-1, h(t)) + 1 & \text{otherwise} \end{cases} \tag{4}$$

This is, when a node is in the rightmost path its number of slots is the offset past the last element of the vector, otherwise the node is full. This computation is slow in comparison to just querying the number of elements for the array. But it can be implemented efficiently by leveraging the contextual information we have during the traversal. For example, in a `push_back()` operation that appends a new element we *know* that we are traversing the rightmost branch. Thus, no conditional is needed and the slot count can be computed using fast bitwise operations. For operations that traverse the tree towards arbitrary indexes of a vector, such an `update(i, v)` that changes the value of the i-th element to v, we can separate the traversal of full and rightmost nodes as in listing 4.1.

```
erbtree update(size_t i, T v) const {
  return { update_(root, shift, i, v), shift, size };
}

const node*
update_full_(const node* n, int s, size_t i, T v) const {
  if (s == B - BL) {
    auto slot = i & mask<BL>;
    auto newn = copy_leaf_node(n, M<BL>);
    newn->leaf[slot] = v;
    return newn;
  } else {
    auto newn = copy_inner_node(n, M<B>);
    auto slot = i & mask<BL>;
    newn->leaf[slot] = update_full_(n, s - B, i, v);
    return newn;
  }
}
```

```
const node*
update_(const node* n, int s, size_t i, T v) const {
  if (s == B - BL) {
    auto count = ((size - 1) >> s) & mask<BL> + 1;
    auto slot = i & mask<BL>;
    auto newn = copy_leaf_node(n, count);
    newn->leaf[slot] = v;
    return newn;
  } else {
    auto count = ((size - 1) >> s) & mask<B> + 1;
    auto slot = i & mask<B>;
    auto newn = copy_inner_node(n, count);
    newn->leaf[slot] = i < count - 1
      ? update_full_(n, s - B, i, v)
      : update_(n, s - B, i, v);
    return newn;
  }
}
```

Listing 4.1. An `update()` ERB-Tree operation without type or slot count metadata

[15] Like `std::shared_ptr` or the faster `boost::intrusive_ptr`.

While efficient, this code suffers from significant duplication. The `do_update` and `do_update_full` functions are identical excepting for (1) the computation for the slot count, and (2) the the recursive call down to the next child. The problem gets worse when we introduce relaxed nodes in RRB-Trees and binary operations like concatenation. To solve this problem we introduce the notion of *positions*.

4.4 Position Based Traversals

A *position* is an object that contains a reference to a node alongside the metadata needed to do useful things with it. It is not stored in the tree, but created on demand during traversals. A *position* supports a series of higher order operations to visit its children. These operations use the context they have to instantiate the most efficient position type for the child, and pass this instance to a visiting operation that it took as argument. Via mutual recursion between positions and visitors, any kind of traversal can be described.

A visitor may be defined as one generic function that is applied to any kind of node. Recursive traversals terminate because higher order operations for leaf nodes do nothing. However, most useful visitors are implemented as two or three functions that distinguish leaf or inner nodes (regular or relaxed).

Listing 4.2 shows how the `update()` operation may be implemented in terms of the position framework. Appendix A shows a possible implementation of a position framework supporting this kind of operation. Note that this vistor is defined for Embedding RRB-Trees, but only *positions* and the node copying operations are concerned with the details of embedding and relaxation.

```
struct update_op {};

errbtree update(size_t i, T v) {
  auto newr = visit_maybe_relaxed(root, update_op{}, i, v);
  return { newr, shift, size };
}

template <typename Pos, typename T>
auto visit_regular(update_op op, Pos pos, size_t i, T v) {
  auto newn = copy_regular(pos.node, pos.count());
  newn->inner[pos.offset(i)] = pos.towards(op, i, v);
  return newn;
}

template <typename Pos, typename T>
auto visit_relaxed(update_op op, Pos pos, size_t i, T v) {
  auto newn = copy_relaxed(pos.node);
  newn->inner[pos.offset(i)] = pos.towards(op, i, v);
  return newn;
}

template <typename Pos, typename T>
auto visit_leaf(update_op op, Pos pos, size_t i, T v) {
  auto newn = copy_leaf(pos.node, pos.count());
  newn->inner[pos.offset(i)] = v;
  return newn;
}
```

Listing 4.2. The `update()` (E)RRB-Tree operation using *positions*.

All the bit wizardry is hidden in the positions, and the traversal can be optimized without changing the visitor. The *operation* is not concerned anymore with the details of how to navigate through the tree. Instead, it focuses on what it needs to do to each node in order to produce a new structure. No combinatorial explosion happens between the types of nodes and the types of positions. When implemented in this way, changing an RRB-Tree into an ERRB-Tree structure is simple.

This code is also as performant as the hand-written traversal without positions. Note that all dispatching is done statically. The recursive visitor takes the positions as a template argument, and it is instantiated *at compile time* for all possible positions in the traversal. When instantiating the visitor in listing 4.2 against the position framework in appendix A, a call graph as shown in figure 5 is produced. All definitions are visible to the compiler allowing inlining and other optimizations.

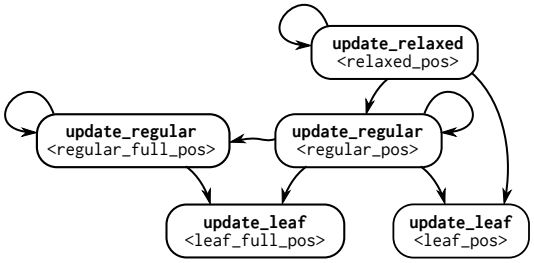


Fig. 5. Instantiated call graph for the `update()` operation using *positions* over an (E)RRB-Tree.

5 MEMORY MANAGEMENT

A garbage collection mechanism is required for immutable persistent data structures to be effective. As shown, when an operation updates the data structure, new nodes are allocated to copy the changed parts of the data structure, but the new value also references parts that did not change. Eventually, when old values are not used anymore, a node may loose any references to it and its memory should be recycled.

5.1 Garbage Collection in C++

C++ does not provide any pervasive garbage collection mechanism by default. These three mechanisms are often used instead:

(1) *Reference counting*. This is the most common technique. It can be implemented using C++ constructor/destructors and its determinism plays well with the rest of the language. The standard library provides an opt-in generic reference counting pointer type (`shared_ptr`).
(2) *Uncooperative garbage collection*. A common implementation is Boehm's conservative garbage collector [Boehm and Weiser 1988]. Composing GC-based objects into objects that use `malloc` is problematic because the GC can not see the `malloc`-ated memory.[16] Specially troublesome is the fact that there is no way to efficiently and deterministically call destructors of C++ objects. Still, since C++11 the standard allows such garbage collectors via an optional API [Boehm et al. 2008].
(3) *Cooperative garbage collection*. This is a common solution when writing C extensions to higher level languages. For example, GNU Guile provides an API for custom C types to cooperate in the mark phase of the GC [Galassi et al. 2002]. Racket [Flatt and PLT 2010] supports a compacting GC but defines an API[17] with sufficient semantics for C extensions to cooperate [Rafkind et al. 2009]. Python[18] or Ruby [Grimmer et al. 2015] are two other examples with similar approaches.

5.2 Policy Based Design

We use *policy based design* [Alexandrescu 2001] to support all these mechanisms without implementing the data structure multiple times. A *policy* is a type modelling a *concept* that factors some *aspects* of the behavior of some other type. A type takes in policies as template arguments that are bound and monomorphized at compile time, allowing the compiler to inline calls or remove empty

[16] Workarounds exist for this problem. With `libgc` one can use `GC_malloc_uncollectable` instead of `std::malloc` where memory is managed manually.
[17] http://docs.racket-lang.org/inside/im_memoryalloc.html
[18] https://docs.python.org/3/extending/

objects[19]. It is sometimes thought of as a compile-time version of the *strategy pattern* [Gamma et al. 1995]. A good factoring into policies enables extensibility *without a performance penalty* and it is thus useful to enable the user make their own trade-offs when configuring performance sensitive aspects of some type. Listing 5.1 shows two policies for reference counting as implemented in our system.

```
struct refcount_policy {
  mutable std::atomic<int> rc{1};
  void inc()
  { rc.fetch_add(1, std::memory_order_relaxed); }
  bool dec()
  { return 1 == rc.fetch_sub(1, std::memory_order_acq_rel); }
  void dec_unsafe()
  { rc.fetch_sub(1, std::memory_order_relaxed); }
  bool unique() const
  { return rc.load(std::memory_order_acquire) == 1; }
};
```

```
struct no_refcount_policy {
  void inc() {}
  bool dec() { return false; }
  void dec_unsafe() {}
  bool unique()
  { return false; }
};
```

Listing 5.1. Two reference counting policies. The `refcount_policy` enables thread-safe reference counting via an atomic integer count. The `no_reference_counting` is a no-op policy to be used when some other garbage collection is available. Our system provides an additional `unsafe_refcount_policy` for single-threaded systems.

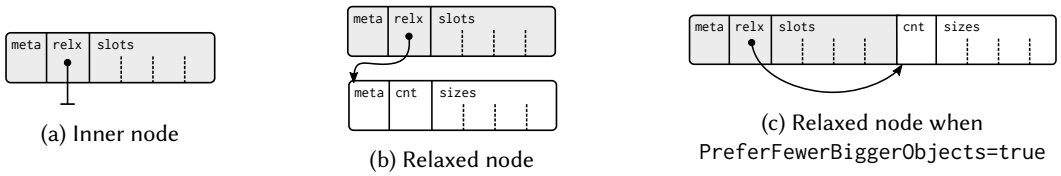
5.3 Memory Policies

In our system, users can customize various aspects of the memory layout and management by passing a `memory_policy<...>` as one of the template arguments of the container. A memory policy takes the following arguments:

(1) A `HeapPolicy` specifies how memory should be allocated and released. It allows, for example, to exchange standard `malloc` with `libgc` [Boehm and Weiser 1988].
Furthermore, our heap policies use memory allocator layering [Berger et al. 2001] to be able to define memory allocators out of composable building blocks. We provide heaps adaptors to enable both global and thread-local free-lists.
(2) A `RefcountPolicy` like the ones in listing 5.1.
(3) A `TransiencePolicy` used to implement transience when reference counting is not available (§ 6.1.1).
(4) A `prefer_fewer_bigger_objects` flag determining the layout of inner nodes as shown in figure 6.
(5) A `use_transient_rvalues` flag determining whether *r-values* should be considered transients (§ 6.3).

Note that not all parameters are completely independent—this is, there are some combinations of policies that do not make sense. For example, using a `libgc_heap` only a `no_refcount_policy` and a `gc_transience_policy` should be used. Likewise, `no_transience_policy` should be used whenever a reference counting mechanism is enabled. Some of these dependencies are captured by meta-functions that default-initialize the unspecified arguments of the memory policy.

[19]When combine with techniques like *empty based optimization*



(a) Inner node (b) Relaxed node (c) Relaxed node when `PreferFewerBiggerObjects=true`

Fig. 6. Inner node layouts. Regular inner nodes do not store the size of the subtrees, thus the `relx` pointer is `null`. A relaxed node may or may not have the size array allocated in the same memory object. In the former case, the size array needs its own `meta` to track reference counts or transient ownership. In the latter case, the size array can not be shared between nodes, making the `update()` slower, but improving traversals and reducing allocations in other update operations.

6 TRANSIENCE

6.1 Background

While RRB-Trees perform very well for a *persistent* data structure, they are suboptimal when persistence is not required. This happens when writing pure functions that perform multiple updates to a data structure but only return the last version. From the point of the view of the caller, the function is a *transaction* and we can only observe the accumulated effects of the whole operation. We may persist its inputs and outputs, but intermediate results produced inside the function are necessarily forgotten when it returns. An example of such suboptimal operation is shown in listing 6.1a, which defines a `iota(v, f, l)` that appends all integers in the range $[f, l)$ to the immutable vector v.

```
vector<int> iota(vector<int> v, int f, int l) {
  for (auto i = f; i < l; ++i)
    v = v.push_back(i);
  return v;
}
```

(a) Suboptimal `iota()`

```
vector<int> iota(vector<int> v, int f, int l) {
  auto t = v.transient();
  for (auto i = f; i < l; ++i)
    t.push_back(i);
  return t.persistent();
}
```

(b) Efficient `iota()` using transients

Listing 6.1. Populating an immutable vector

Clojure solves this problem by introducing the notion of *transients*. A *transient container* can be constructed in $O(1)$ from its persistent counterpart by merely copying a reference to its internal data. However, the transient has different semantics, such that update operations *invalidate* the container supplied as argument. This allows transient operations to sometimes update the data-structure in-place, stealing the passed in tree and mutating its node objects. Still, the transient has to preserve the immutability of the original persistent value whose contents it adopted at the beginning of the transaction.

6.1.1 Copy on Write. To be able to track which nodes it can mutate in place, the transient is associated with a globally unique identifier that is generated at the beginning of the transaction. We then proceed using a copy-on-write strategy. In a transient update operation, before mutating a node in place, we check whether the node is tagged with the current transient identifier—i.e. we check if the transient *owns* the node. Only if it owns the node is it allowed to modify the node in-place—otherwise a new copy is made, tagged with the transient identifier. Update operations

do this for every node in the path down to the affected leaves, thus making sure the mutations have no visible effects outside of the transient. A transient can be converted back into a persistent value, invalidating the transient. Thanks to this invalidation, the operation can be performed in $O(1)$ without cleaning up the tags—a new transient is going to have a different identifier, so it is impossible for newer transients to mutate those nodes that are remain tagged with the identifier of the finished transient. In our system, the *transience policies* (see appendix B for an implementation) describe how these identifiers are created and compared.

6.1.2 Interface. In Clojure, transient operations are not *referentially transparent*, even though functions that only use them internally can be so.[20] However, transients can be modelled in a pure language supporting affine types [L'orange 2014; Walker 2005].

In C++ it feels natural to encapsulate transients behind an explicitly mutable interface as exemplified in listing 6.1b. This makes the invalidation of the previous state more obvious, hopefully lowering chances for programming mistakes. Also this interface is compatible with generic standard library algorithms and components which expect mutable containers. In this way, persistent vectors become a first class citizen of the language. In section § 6.3 we present an alternative interface that does not sacrifice a functional style.

6.2 Reference Counting

The ownership tracking mechanism described above is redundant when combined with reference counting. Instead of generating transient identifiers, we can make a node t eligible for in-place mutation when its reference count is 1 and every node in the path up to the root also has a reference count of 1. This means that the path is only referenced by the transient and the effects of modifying it in-place are not visible outside of the transient.

The two ownership tracking rules can be combined into one. Algorithm 6.1 shows the general structure of a copy-on-write update operation that can operate under either mechanism. Each of the rules are implemented via policies, and they can be disabled by passing in a no-op policy that the compiler will inline away. Normally, only one of these two policies should be enabled for a given instance—using both mechanisms at the same time is harmless albeit inefficient and redundant.

6.3 Move Semantics

6.3.1 R-value References. C++ implements the notion of *move semantics* via *r-value references* [Hinnant et al. 2004]. Typically a variable can be bound to a either (1) a *value* (type of the form `T`), this is, an new object whose lifetime is determined by its lexical scope, or (2) a *reference* (type of the form `T&`), this is, an alias to some other existing object. In C++11 (3) *r-value references* were introduced (type of the form `T&&`), this is, a special reference that can only alias anonymous objects—temporaries, like the object containing the result of an expression.

R-value references are useful when combined with overloading. For some operations, it is common to provide two overloads, one that takes in an r-value reference of some type (`T&&`), and another one that takes in read-only references of that type (`const T&`). In general the latter overload will be used, but when some anonymous object is passed as argument the former takes precedence. The `T&&` overload can often be implemented more efficiently because, since the reference is known to be bound to an anonymous object—this is, to an object that that does not have any other alias and thus is invisible outside of this scope. This allows to *steal* parts of the internal representation of the argument in the process of producing its own results. This is, while in a `const T&` overload

[20] In Clojure, there are *runtime checks* protect against using using an invalidated transient or sharing a transient across multiple threads.

```
function ensure-mutable(node, mutating, id)
    mutating ← mutating ∧ (unique(node) ∨ owner(node) = id)
    if ¬mutating then
        node ← copy-node-with-owner(node, id)
    end if
    return (node, mutating)
end function

function transient-update(node, id, mutating, ...)
    (node, mutating) ← ensure-mutable(node, mutating, id)
    r ← transient-update(node, id, mutating, ...)
    update-in-place(node, r)
    return node
end function
```

Algorithm 6.1. Structure of a transient operation. The *unique*(*node*) expression tests whether a node's reference count is 1, and *owner*(node) returns the identifier associated to the transient owning the *node*.

the passed in objects need to be deeply *copied* when deriving new results from the passed in values, the `T&&` *moves* the passed in contents into the results, hence the name *move-semantics* for this general approach.

A truly anonymous *r-value* could in theory only be used *once*, thus forming a special kind of *affine type*. Affine types where used in the Rust programming language to model fully type safe move-semantics that unlike C++ are not based in r-value references [Matsakis and Klock 2014].

In C++ r-values references are not truly unique aliases because (1) the destructor is unconditionally called at the end of its lifetime, even after it was *moved*, and (2) the programmer can also cast a named variable (an *l-value*) to an r-value using the `std::move()` function. Even though an object is left in an unspecified state after being moved from, it is valid in the sense that the destructor must still succeed. The compiler will not check that a named object that has been moved from is not used again. Furthermore, it is common to define the assignment operator such that moved-from objects can be reassigned to give them a specified and known state.

6.3.2 R-value Transients. These semantics allow us to say that an r-value of a persistent container is a *transient r-value*. For every operation in the container, an overload for r-value references is provided that optimizes updates using the transient rules. Listing 6.2 show two examples where transient r-values are manipulated.

```
auto v = vector<int>()
  .push_back(7)
  .push_back(42)
  .push_back(101)
  .push_back(30);
```

(a) Chaining r-value temporaries

```
vector<int> iota(vector<int> v, int f, int l) {
  for (auto i = f; i < l; ++i)
    v = std::move(v).push_back(i);
  return v;
}
```

(b) Casting l-values to r-values

Listing 6.2. Manipulating r-value transients. On the left, every push_back call is applied to temporaries and thus the transient rules are used to perform in-place updates without any intervention from the programmer. On the right, by explicitly moving v into the push_back call, this implementation of `iota` has the same performance as that in listing 6.1b.

The applicability of this technique is not exclusive to C++. Notably, Rust is a systems programming language that includes type safe move semantics (*borrowing*) modelled after affine types, but

its lack of overloading means that the programmer would always need to explicitly note which version (transient or persistent) of an operation to pick—even though the compiler would most often protect them from picking the wrong one. We dream of a language that provides the transparency of overloaded r-value references (like C++), while removing the burden of explicitly moving variables when they are last used by using stronger and more type safe scoping rules (like Rust).

6.3.3 Ownership Tracking. Since l-values can be converted into r-values, we still need to track node ownership to ensure we do not modify aliased nodes. When using identifier based ownership tracking, we would need to generate a new identifier whenever we alias a persistent vector (in C++ terms, when copy it). This means that in-place modifications would happen only from the second operation that is applied *consecutively* on a r-value transient chain. Only programs with lots of batch updates would benefit from this—for most programs, we speculate that the cost of generating new identifiers all the time out-weights the potential gains.

However, reference counting keeps an accurate count on the aliasing of a node all the time *anyways*. No extra runtime cost is required to optimize updates on r-values, with potentially very high gains. Our implementation uses r-value transience whenever reference counting is enabled. This can be controlled by the user via the `use_transient_rvalues` switch of the `memory_policy` (§ 5.3).

6.3.4 Costs. To enable in-place push back, the rightmost path of the tree needs to keep enough space allocated in the nodes to write the new data. In the worst case, this has a $M \times \log_M(n)$ memory overhead. While this is negligible for non-trivial vectors, it might have an impact on an application that uses lots of small vectors where $n \ll M$. A solution could be to use an exponential growth mechanism for rightmost nodes, similar to that used for non persistent vectors. Sometimes the programmer may as well juse use immutable arrays when $n < M$.

6.4 RRB-Tree Transience

6.4.1 Background. Transients were first introduced in Clojure for regular Radix Balanced Tree based vectors. Later, when they implemented RRB-Trees[21] they only added support for transient `push`, `take`, and `update`. They do not store the owner id in the size array of relaxed nodes. Thus, the size array needs to be copied unconditionally when updating an aliased node, even when the sizes of the sub-trees do not actually change.

L'orange [2014] proposes storing the owner id in the size array, which increases the level of structural sharing after an `update` operation. Furthermore, when possible, they convert an editable regular inner node into a relaxed node by assigning a size array in-place. However, their implementation only supports transient `push` and `update`.

We implemented transient versions of all RRB-Tree operations, using either of the previous approaches depending on whether the size array is embedded or not (see figure 6). For most operations, a transient alternative can be designed by systematically applying the structure described in algorithm 6.1. However, *concatenation* requires special treatment.

6.4.2 Confluent Transients. Borrowing the terminology from Fiat and Kaplan [2001], concatenation is a *meld* operation—i.e. an operation that *combines* at least two persistent values into one. RRB-Trees is *confluently persistent* because it supports a meld operation without copying all the data from either argument. Let α_τ be the type of a persistent data structure supporting a melding operation

$$\oplus : \alpha_\tau \times \alpha_\tau \rightarrow \alpha_\tau \tag{5}$$

[21] `core.rrb-tree`: https://github.com/clojure/core.rrb-vector

we can define the following *three* corresponding transient operations when combined with its transient $\alpha^!_\tau$

$$\begin{aligned} \oplus^!_l &: \alpha^!_\tau \times \alpha_\tau \to \alpha^!_\tau \\ \oplus^!_r &: \alpha_\tau \times \alpha^!_\tau \to \alpha^!_\tau \\ \oplus^!_b &: \alpha^!_\tau \times \alpha^!_\tau \to \alpha^!_\tau \end{aligned} \tag{6}$$

These are produced by taking a transient in either or both arguments. These operations naturally map to an r-value based interface, but it can also be modelled using mutable l-values, as shown in listing 6.3. Note that in the l-value version the last two methods are fundamentally the same, provided for symmetry.

```
struct vector {
  // ...
  friend vector&& operator+(vector const& lhs, vector&& rhs);
  friend vector&& operator+(vector&& lhs, vector const& rhs);
  friend vector&& operator+(vector&& lhs, vector&& rhs);
};
```

(a) Using r-value transients

```
struct vector_transient {
  // ...
  void append(vector const& rhs);
  void prepend(vector const& rhs);
  void append(vector_transient&& rhs);
  void prepend(vector_transient&& rhs);
};
```

(b) Using mutable transient types

Listing 6.3. Modeling transient concatenation in C++

6.4.3 Transient Concatenation. Implementing transient concatenation does not require one to implement the three meld transient functions separately. The trick is to carry three owner identifiers, that of the left side, that of the right side, and that of the new center nodes. When a node is created or adopted it is assigned the center identifier, which is the identifier of the resulting tree. The center identifier should be that of the left tree for $\oplus^!_l$ and that of the right tree for $\oplus^!_r$. For $\oplus^!_b$, we can choose either side based on some heuristic—we propose taking that of the bigger tree. Algorithm 6.2 provides a general skeleton for transient meld operations.

Note that transient concatenation is only barely faster than normal concatenation. The meat of the algorithm happens in the rebalancing step. In the transient version, objects are reused when possible (saving allocations) but the data needs to be moved or copied around anyways. In order to reuse nodes we add complexity (more branching, recursion parameteres, etc.) that expends most cycles that we save by avoiding allocations.

Furthermore, implementing this operation with manual reference counting is particularly tricky because we destroy the old tree as we go, making it hard to keep an account of which nodes to `inc()` or `dec()`. Note that with reference counting we can instead keep a free-list of recently deallocated objects (and we do with the default *memory policy*)—we save allocations yet keep a simpler recursion, often being *faster*. Also, with reference counting, the transient concatenation algorithm is not required to keep an account of transient ownership. Thus, when using reference counting, our implementation concatenates small trees in-place when only the tail is affected, but resorts to the persistent concatenation algorithm for the general case.

However, we do enable transient concatenation when automatic garbage collection and id-based ownership are used. Otherwise we would not tag the produced and adopted nodes with the identifier of the new owning transient. Failing to do so pessimizes updates later performed on the resulting transient value.

```
function ENSURE-MUTABLE-WITH-NEW-ID(node, mutating, id, new-id)
    mutating ← mutating ∧ (unique(node) ∨ owner(node) = id)
    if ¬mutating then
        node ← COPY-NODE-WITH-OWNER(node, new-id)
    else
        owner(node) ← new-id
    end if
    return (node, mutating)
end function

function TRANSIENT-MELD( node_l, id_l, mutating_l, node_r, id_r, mutating_r, id_c, ...)
    (node_l, mutating_l) ← ENSURE-MUTABLE-WITH-NEW-ID(node_l, mutating_l, id_l, id_c)
    (node_r, mutating_r) ← ENSURE-MUTABLE-WITH-NEW-ID(node_r, mutating_r, id_r, id_c)
    node_c ← TRANSIENT-MELD( node_l, id_l, mutating_l, node_r, id_r, mutating_r, id_c, ...)
    node'_c ← MELD-IN-PLACE(node_l, node_c, node_r)
    return node'_c
end function
```

$$a_l \oplus^!_l a_r = \text{TRANSIENT-MELD}(root(a_l), \top, id(a_l), root(a_r), \bot, id_{noone}, id(a_l))$$
$$a_l \oplus^!_r a_r = \text{TRANSIENT-MELD}(root(a_l), \bot, id_{noone}, root(a_r), \top, id(a_l), id(a_l))$$
$$a_l \oplus^!_b a_r = \text{TRANSIENT-MELD}(root(a_l), \top, id(a_l), root(a_r), \top, id(a_r), id(choose(a_l, a_r)))$$

Algorithm 6.2. Structure of a transient meld operation

7 EVALUATION

7.1 Methodology

We evaluated various implementations (table 1) by running several benchmarks in a specific system (tables 2 and 3). We run each benchmark for three *problem sizes N* (normally, this is the size of the vector). For practical reasons, we take less samples of bigger problem sizes (table 4).

We run C/C++ benchmarks using the `Nonius` framework,[22] Clojure benchmarks using `Criterium`,[23] Python using `PyTest.Benchmark`,[24] and Scala using `ScalaMeter`.[25] The two first frameworks are based on the Haskell `Criterion` framework[26] which introduces interesting statistical bootstrapping methods for the detection of outliers. The rest also do some form of outlier detection. All of them do appropriate measurement of the clock precision and run each benchmark enough times per sample to obtain significant results. The JVM based frameworks take care of minimizing the impact of the garbage collector in the measurements, as well as ensuring that the code is properly JIT-compiled [Georges et al. 2007]. In C++, we manually trigger a full `libgc` collection before each benchmark to avoid remaining garbage from previous benchmarks impacting the performance. We also considered disabling the garbage collector during the measurement, but this is impracticable for big problem sizes.

7.2 Results

The benchmark results are presented in tables 5 to 8.

[22] https://nonius.io/
[23] https://github.com/hugoduncan/criterium/
[24] https://github.com/ionelmc/pytest-benchmark
[25] http://scalameter.github.io/
[26] http://hackage.haskell.org/package/criterion

Table 1. Evaluated implementations

`ours/gc`	Our ERRB-Tree implementation with tracing garbage collection using `libgc`.
`ours/safe`	Our ERRB-Tree implementation with thread safe reference counting using atomic counters. Memory is allocated using standard `malloc`, stacked under a global free list of up to 1024 objects (using lock-free synchronization with atomic pointers), stacked under a thread local free list of up to 1024 objects (not synchronized). In some benchmarks we disable the free list and name it `ours/basic`.
`ours/unsafe`	Our ERRB-Tree implementation with thread unsafe reference counting. Memory is allocated using `malloc`, stacked under a global free list of up to 1024 objects (not synchronized).
`ours/py`	Python bindings for our ERRB-Tree implementation written directly against the Python C interface[a]. These provide a Python `Vector` class able to hold any dynamically typed Python object. It uses `Py_Malloc` to allocate internal nodes, thread unsafe reference counting to track these, and collaborates with the Python garbage collector to trace the contained objects.
`librrb`	The C implementation[d] by L'orange [2014] (`librrb`). It uses `libgc` for garbage collection and contains boxed objects. In our benchmarks, when storing integers, we just reinterpret these as pointers to store them unboxed—this should give more comparable results.
`clojure`	Clojure standard vectors implementing tail optimized RB-Trees (`clojure`). It is written in Java.
`clojure.rrb`	The Clojure implementation of RRB-Trees[e]. It is written in Clojure.
`scala`	Scala standard vectors implementing RB-Trees with display.
`scala.rrb`	Scala implementation[f] of RRB-Trees with display by Stucki et al. [2015].
`pyrsystent`	A Python implementation[g] of RB-Trees. They provide both an implementation in C by default, but also an implementation in Python for systems where C modules are not available.

[a] We also have experimented with using the C++ frameworks Boost.Python[b] and PyBind11[c] but these add too much overhead. [b] http://www.boost.org/doc/libs/release/libs/python [c] https://github.com/pybind/pybind11
[d] https://github.com/hyPiRion/c-rrb [e] https://github.com/clojure/core.rrb-vector
[f] https://github.com/nicolasstucki/scala-rrb-vector [g] https://github.com/tobgu/pyrsistent

Table 2. System hardware

Processor	Intel Core i5-460M (64bit)
Frequency	2.53 GHz
L1 Cache (per core)	2 × 32 KB (8-way assoc.)
L2 Cache (per core)	256 KB (8-way assoc.)
L3 Cache	3MB (12-way assoc.)
RAM	4 GB DDR3 (1,066 MHz)

Table 3. System software

OS	Linux 4.9.0 (Debian)
Compiler	gcc 6.3.0
Java	openjdk 1.8.0-121
Python	cpython 2.7.3
Scala	scala 2.11.11
Clojure	clojure 1.8.0

Table 4. Problem sizes

Size	N	Samples
Small (S)	10^3	100
Medium (M)	10^5	20
Large (L)	10^7	3

7.2.1 Abstraction Cost. Our implementation uses various abstraction mechanisms (§ 4.4, § 5.2). We argued that these are zero-cost abstractions—or may even incur a negative cost when used to remove metadata. We can evaluate this by comparing our implementation with `librrb` and `pyrsistent`, since both are written in C using similar optimizations (off-tree tail). Our Python bindings are faster than `pyrsistent` in all benchmarks. Our implementation (when combined with `libgc`) is faster than `librrb` in most benchmarks excepting two.

`librrb` supports faster transient random updates (table 7) and shows a speedup of around 20% because their implementation does not support exceptions (it is plain C after all) nor recovers gracefully from memory exhaustion. Their update function is thus just a simple loop while, in order to be exception safe, our implementation uses non-tail recursion, executing all potentially failing operations first and only then mutating the tree.

Table 5. Access benchmarks. The sum of all values in a n element vector is computed. Elements are accessed either by sequential indexes (e.g. via `operator[]`), iterators, or via internal iteration (i.e. higher order *reduce* function).

		10ns | **S**	µs | **M**	100µs | **L**
INDEXING	ours	595	828	851
	relaxed ours	497	930	1573
	librrb	466	905	1268
	relaxed librrb	978	1280	3302
	ours/python	5412	5973	7052
	pyrsistent	5794	6189	7131
	scala	9437	1546	1739
	scala.rrb	9500	1167	1360
ITERATION	ours	120	126	144
	relaxed ours	88	113	221
	scala	17368	772	562
	scala.rrb	17879	629	559
FOLDING	ours	29	36	75
	relaxed ours	32	38	109
	scala	13904	1033	943
	scala.rrb	14835	1049	941
	clojure	5122	5578	6902
	clojure.rrb	14626	19264	23533
	std::vector	19	21	56
	std::list	210	465	581

Table 6. Append benchmarks. A n element vector is produced by sequentially appending n elements. In the transient version a mutable interface is used.

		µs | **S**	100µs | **M**	10ms | **L**
PERSISTENT	ours/basic	136	140	172
	ours/safe	71	76	98
	ours/unsafe	31	30	34
	ours/gc	48	73	139
	librrb	76	106	184
	ours/python	214	221	238
	pyrsistent	332	313	338
	scala	143	65	159
	scala.rrb	175	64	151
	clojure	107	117	—
	clojure.rrb	246	257	—
TRANSIENT	ours/basic	7	7	8
	ours/safe	6	6	8
	ours/unsafe	5	6	7
	ours/gc	5	5	6
	librrb	9	9	12
	clojure	65	66	—
	clojure.rrb	84	85	—
	std::vector	4	3	8
	std::list	56	56	58

Table 7. Update benchmarks. Every element in a n element vector is updated using sequential indexes. In the transient version, a mutable interface is used without destroying the initial value.

		µs | **S**	100µs | **M**	10ms | **L**
PERSISTENT	ours/basic	459	1385	2269
	ours/safe	327	1200	2004
	ours/unsafe	76	350	667
	ours/gc	251	409	1226
	relaxed ours/basic	519	1549	2514
	relaxed ours/safe	380	1335	2249
	relaxed ours/unsafe	135	445	891
	relaxed ours/gc	182	441	828
	librrb	288	551	1049
	relaxed librrb	316	615	1238
	ours/python	419	732	994
	pyrsistent	605	884	1140
	scala	136	64	144
	scala.rrb	103	68	158
	clojure	359	626	—
	clojure.rrb	542	883	—
TRANSIENT	ours/basic	14	31	41
	ours/safe	14	31	40
	ours/unsafe	15	31	40
	ours/gc	17	38	51
	relaxed ours/basic	15	31	40
	relaxed ours/safe	14	31	40
	relaxed ours/unsafe	15	31	41
	relaxed ours/gc	17	38	50
	librrb	13	21	28
	relaxed librrb	19	25	40
	clojure	114	149	—
	clojure.rrb	270	340	—
	std::vector	1	3	5

Table 8. Concate benchmarks. A n element vector is produced by concatenating 10 equally sized vectors.

		µs | **S**	µs | **M**	µs | **L**
PERSISTENT	ours/basic	7	30	159
	ours/safe	5	26	206
	ours/unsafe	3	16	158
	ours/gc	4	16	122
	librrb	8	21	103
	scala.rrb	58	221	857
	clojure.rrb	1133	492	—
TRANSIENT	ours/gc	4	15	112

Table 8 (L column) shows an example in which `librrb` does faster concatenation, while our implementation seems faster in all others. Note that we are comparing vectors of normal `int` values. These are 32 bit in size while pointers are 64 bit wide, thus $B_l = 6$ because of embedding. The resulting relaxed structures are not the same and in this particular instance we happen to need more rebalancing at the leaves. When repeating the benchmark controlling for B_l, our implementation provides a consistent 50% speedup because it uses positions to avoid allocating auxiliary center nodes.

7.2.2 Abstraction Suitability. One of our main goals was to offer sufficiently customizable memory management such that our implementation could be integrated in other language runtimes. In a few hours we had an initial integration with the Python, supporting cooperative garbage collection and allocating memory in the idiomatic ways suggested by the interpreter documentation. These bindings are already faster than `pyrsistent`, a C implementation manually tailored against the Python interpreter that implements similar optimizations: it keeps the tail off-tree, uses single threaded reference counting, and it keeps a free list of recently released nodes.

7.2.3 Embedding Effectiveness. While embedding provides some advantage in most operations, its benefits are most evident when accessing the elements. In our implementation we provide three ways of accessing the elements: (1) querying an element by index, (2) using iterators (3) using `reduce` (i.e. folding). The three methods are compared in table 5.

When using the *reduce* method, our implementation is only 50% slower than a fully contiguous `std::vector`. This is an excellent result, considering how efficient contiguous arrays are. That method uses *internal iteration*—this is, it traverses the data-structure once, applying a given operation on the elements. Using templates, the operation can be inlined in the traversal.

External iteration adds some overhead. At every step the iterator needs to check if it is done running through the current leaf, and when it is, it needs to traverse down the tree to find the next leaf. The problem of external iteration over hierarchical data structures is further discussed by Austern [2000]. Still, thanks to embedding, iterating over an ERRB-Vector is a few times faster than iterating over a `std::list`.

Comparing the access performance across languages is hard to do in a fair way, because other languages have unrelated costs due to some other features. For example, Clojure's dynamism taxes dealing with basic types[27]. Still, in spite of the best efforts of the JVM JIT, folding C++ EERB-Trees seems orders of magnitude faster than folding in any of the Java based languages, and random access is still about twice as fast.

7.2.4 Transience Effectiveness. Looking at tables 7 and 6 we see that transient updates are an order of magnitude faster than their immutable counterparts.

In fact, for large data sets appends are even faster in a transient ERRB-Vector than in a standard mutable vector (table 6, column L). We believe that this is due to cache utilization. Even though a standard mutable vector uses exponential growth to support amortized $O(1)$ appends, in the growth step it needs to copy all the data. When the vector does not fit in the cache it needs to load it from main memory. RRB-Vector appends only touches the rightmost path of the tree, which fits in L1 cache even for huge vectors.

Mutating transient operations are still an order of magnitude slower than on a `std::vector`. In this case, the RRB-Tree update cost is dominated by the lookup. Note that we deliberately tested ordered updates to put our implementation against the wall. We saw in informal experiments that when performing the updates in random order, a mutable vector slows down an order of magnitude

[27] Clojure supports monomorphic vectors for basic types using `(vector-of :type)`. We tried that method and, surprisingly, found it to actually be slower than generic vectors. Thus we decided not to include it in the presented data

for big data sets. Another way to bring RRB transient updates closer to those on mutable purely sequential data would be to provide mutable iterators.

Note that while Scala updates are very efficient compared to immutable updates on other implementations. This is because the *display* optimization (§ 2.3.4) achieves amortized $O(1)$ updates when applied sequentially. However, the display management adds some cost to other operations. *Transient* updates, although less general, are still significantly faster and can optimize non-local updates.

7.2.5 Considering Reference Counting. While reference counting is the most convenient garbage collection for the C++ developer, they are believed to be inadequate for the implementation of immutable data structures. For arbitrarily deep data structures (e.g. lists), they may overflow the stack when releasing a big object. This is not a problem for RRB-Trees. But an RRB update touches up to $M \times \log_M(n)$ reference counts. In a multi-threaded system these updates must be atomic, adding additional overhead. This is also relatively cache inefficient, because the reference counted object needs to be visited in order to update its count. These may also cause *shadow writes* to the immutable data neighbouring the reference count, limiting the parallelism of concurrent reads.

However, reference counting also opens opportunities. Because it reclaims the memory deterministically, we can put the freed nodes in a free list for future reuse. When performing batches of immutable updates, not only does this avoid calling the relatively slow `malloc`, but also reuses buffers that have been used recently, most probably paged in and in cache. Our benchmarks show that combining single-threaded reference counting with free lists (even though they are small) provide the best performance for *all* manipulation operations.

However, multi-threaded reference counting impacts immutable update performance, adding up to a $1.5X$–$2X$ overhead over using `libgc`.[28] For many use cases this might be tolerable, considering the gains across all other operations. But more importantly, reference counting enables transient updates on r-values (§ 6.3). When using move semantics in a disciplined manner across a whole system, how much data is copied during updates depends on the level of aliasing. In other words: this opens a continuum between *immutability* and *mutability*, where update performance is characterized by the actual *dynamic* amount of persistence in the system, even when the programmer uses only immutable interfaces.

We believe that for many realistic workloads this will provide a significant advantage to reference counting over automatic garbage collection. However, we found no way to design an unbiased benchmark to test this hypothesis. Still, our implementation provides a good framework to validate this assumption in concrete real world systems. Users can measure the impact of different memory management configurations in their system and pick the one that fits best.

8 CONCLUSION

We described an implementation of ERRB-Trees in C++. By storing unboxed values and supporting transient operations, its performance is comparable to that of mutable arrays while providing optional persistence, and logarithmic concatenation and slicing. Via generic programming and policy-based design, different memory management strategies can be used such that the data structures can be made available to higher level languages through their C runtimes. Also, when using reference counting and move semantics, all r-values become eligible for transient optimizations. This effectively blurs the boundaries between immutable and mutable values and enables better system wide performance without sacrificing a functional programming style.

[28] Informal experiments on a more recent Skylake Intel processor show that the gap actually increases in modern machines, up to a $3X$–$4X$ difference.

We showed that a systems programming language is suitable for implementing immutable data structures. We hope that this helps making these data structures accessible to a wider audience, enabling functional architectures to face the challenge of building highly concurrent and interactive systems.

9 FUTURE WORK

Associative containers. We would like to apply the methodology and techniques developed in this work to other persistent data structures. Specially interesting are other wide-node trie based random access containers, like HAMT [Bagwell 2001] and CHAMP [Steindorfer and Vinju 2015]. We anticipate that the relatively sparse nature of those data-structures (compared to RRB-Vectors) makes some optimizations more costly (§ 6.3.4) and alternatives need to be developed. Also, the radix-balanced structure could be used to implement persistent compile-time indexed hybrid structures like those in Boost.Fusion[29] or Hana[30]

Diffing and patching. Because of structural sharing, comparing persistent data structures is already relatively efficient. It is interesting to compute the differences between two versions to from a patch that can be used to reconstruct the more recent version from an older one. Applications include: serializing a history of transactions to disk or the network, efficiently updating user interfaces, or implementing version control [Demaine et al. 2010].

Applications. We shall explore how RRB-Vectors can be used to design novel architectures, beyond the obvious ones (e.g document as a value). For example, games often use flat data models with entities factored out horizontally into subsystems, with *components* stored in big per subsystem sequences—*data-oriented design* [Fabian 2013]. RRB-vectors could be used to design persistent high-performance in-memory data-bases for highly interactive systems.

[29] http://boost.org/libs/fusion
[30] http://boost.org/libs/hana

A A *POSITION* FRAMEWORK

```
// Simplified position framework to support the code
// in section 4 of this paper. For a more general
// impletation visit:
// -- https://github.com/arximboldi/immer

template <typename... Args>
auto visit_node(Args...args) {}
template <typename... Args>
auto visit_leaf(Args...args)
{ return visit_node(args...); }
template <typename... Args>
auto visit_inner(Args ...args)
{ return visit_node(args...); }
template <typename... Args>
auto visit_regular(Args ...args)
{ return visit_inner(args...); }
template <typename... Args>
auto visit_relaxed(Args ...args)
{ return visit_inner(args...); }

struct leaf_pos {
  node* node;
  size_t size;
  int offset(size_t i) { return i & mask<BL>; }
  int count() { return offset(size - 1) + 1; }
  template <typename Visitor, typename... Args>
  auto visit(Visitor v, Args... args)
  { return visit_leaf(v, *this, args...);}
};

struct leaf_full_pos {
  node* node;
  size_t size;
  int offset(size_t i) { return i & mask<BL>; }
  int count() { return M<BL>; }
  template <typename Visitor, typename... Args>
  auto visit(Visitor v, Args... args)
  { return visit_leaf(v, *this, args...);}
};

struct regular_full_pos {
  node* node;
  size_t size;
  int shift;
  int offset(size_t i) { return (i >> shift) & M<B>; }
  int count() { return M<B>; }
  template <typename Visitor, typename... Args>
  auto towards(Visitor v, size_t i, Args... a) {
    auto next = node->inner[slot];
    return shift == BL
      ? leaf_full_pos{next, size}.visit(v,i,a...)
      : regular_full_pos{next, size, shift - B}
          .visit(v,i,a...);
  }
  template <typename Visitor, typename... Args>
  auto visit(Visitor v, Args... args)
  { return visit_regular(v, *this, args...);}
};

struct regular_pos {
  node* node;
  size_t size;
  int shift;
  int offset(size_t i) { return (i >> shift) & M<B>; }
  int count() { return offset(size - 1) + 1; }
  template <typename Visitor, typename... Args>
  auto towards(size_t i, Visitor v, Args... a) {
    auto slot = offset(i);
    auto next = node->inner[slot];
    return slot == offset(size - 1)
      ? (shift == BL
         ? leaf_pos{next, size}.visit(v,i,a...)
         : regular_pos{next, size, shift-B}.visit(v,i,a...))
      : (shift == BL
         ? leaf_full_pos{next, size}.visit(v,i,a...)
         : regular_full_pos{next, size, shift-B}.visit(v,i,a...));
  }
  template <typename Visitor, typename... Args>
  auto visit(Visitor v, Args... args)
  { return visit_regular(v, *this, args...);}
};

struct relaxed_pos {
  node* node;
  int shift;
  int offset(size_t i) {
    auto offset = i >> shift;
    auto r = node->relaxed;
    while (r->sizes[offset] <= i) ++offset;
    return offset;
  }
  int count() { return node->relaxed->count; }
  template <typename Visitor, typename... Args>
  auto towards(size_t i, Visitor v, Args... a) {
    auto slot = offset(i);
    auto size = node->relaxed->sizes[i];
    if (i) {
      auto prev_size = node->relaxed->sizes[i - 1];
      i -= prev_size;
      size -= prev_size;
    }
    auto next = node->inner[slot];
    return shift == BL
      ? leaf_pos{next, size}.visit(v, a...)
      : visit_maybe_relaxed(next, size, shift - B, v,i,a...);
  }
  template <typename Visitor, typename... Args>
  auto visit(Visitor v, Args... args)
  { return visit_regular(v, *this, args...);}
};
template <typename Visitor, typename... Args>
auto visit_maybe_relaxed(node* n, size_t sz, int s,
                         Visitor v, Args... as) {
  return n->relaxed
    ? relaxed_pos{n, sz, s, v}.visit(v, as...)
    : regular_pos{n, sz, s}.visit(v, as...) ;
}
```

B TRANSIENCE POLICIES

```
struct gc_transience_policy
{
  template <typename HeapPolicy>
  struct apply {
    struct type
    {
      using heap_ = typename HeapPolicy::type;
      using edit = void*;
      struct owner {
        edit token_ = heap_::allocate(1, norefs_tag{});
        operator edit () { return token_; }
        owner() {}
        owner(const owner&) {}
        owner(owner&& o) : token_{o.token_} {}
        owner& operator=(const owner&) { return *this; }
        owner& operator=(owner&& o) {
          token_ = o.token_;
          return *this;
        }
      };
      struct ownee {
        edit token_ = nullptr;
        ownee& operator=(edit e) {
          assert(token_ == nullptr);
          token_ = e;
          return *this;
        }
        bool can_mutate(edit t) const
        { return token_ == t; }
      };
    };
  };
};
```

```
struct no_transience_policy
{
  template <typename>
  struct apply {
    struct type
    {
      struct edit {};
      struct owner {
        operator edit () const { return {}; }
      };
      struct ownee {
        ownee& operator=(edit) { return *this; };
        bool can_mutate(edit) const { return false; }
      };
    };
  };
};
```

ACKNOWLEDGMENTS

We would like to thank the ICFP reviewers for their very valuable feedback. We are grateful to María Carrasco Rodríguez, Francisco Jerez Plata, Emilio Jesús Gallego Arias, Ryan Brown, Joaquín Valdivia, Javier Martínez Baena, Antonio Garrido Carrillo, and Raphael Dingé for dicussing these ideas, reviewing early drafts, and their encouragement towards the publication of this work.

REFERENCES

Umut A. Acar, Arthur Charguéraud, and Mike Rainey. 2014. *Theory and Practice of Chunked Sequences.* Springer Berlin Heidelberg, Berlin, Heidelberg, 25–36. https://doi.org/10.1007/978-3-662-44777-2_3

Andrei Alexandrescu. 2001. *Modern C++ design: generic programming and design patterns applied.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Matthew H. Austern. 2000. Segmented Iterators and Hierarchical Algorithms. In *Selected Papers from the International Seminar on Generic Programming*. Springer-Verlag, London, UK, UK, 80–90. http://dl.acm.org/citation.cfm?id=647373.724070

Phil Bagwell. 2000. *Fast And Space Efficient Trie Searches.* Technical Report.

Phil Bagwell. 2001. Ideal Hash Trees. *Es Grands Champs* 1195 (2001).

Phil Bagwell. 2002. Fast Functional Lists, Hash-Lists, Deques and Variable Length Arrays. In *In Implementation of Functional Languages, 14th International Workshop*. 34.

Philip Bagwell and Tiark Rompf. 2011. *RRB-Trees: Efficient Immutable Vectors.* Technical Report. EPFL.

Emery D. Berger, Benjamin G. Zorn, and Kathryn S. McKinley. 2001. Composing High-performance Memory Allocators. *SIGPLAN Not.* 36, 5 (May 2001), 114–124. https://doi.org/10.1145/381694.378821

Hans-Juergen Boehm and Mark Weiser. 1988. Garbage Collection in an Uncooperative Environment. *Softw., Pract. Exper.* 18, 9 (1988), 807–820. https://doi.org/10.1002/spe.4380180902

Hans-J. Boehm, Russ Atkinson, and Michael Plass. 1995. Ropes: An Alternative to Strings. *Softw. Pract. Exper.* 25, 12 (Dec. 1995), 1315–1330. https://doi.org/10.1002/spe.4380251203

H-J Boehm, M Spertus, and C Nelson. 2008. N2670: Minimal support for garbage collection and reachability-based leak detection (revised. (2008).

Sébastien Collette, John Iacono, and Stefan Langerman. 2012. Confluent Persistence Revisited. In *Proceedings of the Twenty-third Annual ACM-SIAM Symposium on Discrete Algorithms (SODA '12)*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 593–601. http://dl.acm.org/citation.cfm?id=2095116.2095166

Erik D. Demaine, Stefan Langerman, and Eric Price. 2010. Confluently Persistent Tries for Efficient Version Control. *Algorithmica* 57, 3 (July 2010), 462–483. https://doi.org/10.1007/s00453-008-9274-z

Ulrich Drepper. 2008. *What Every Programmer Should Know About Memory.* Technical Report. Red Hat. http://people.redhat.com/drepper/cpumemory.pdf

J R Driscoll, N Sarnak, D D Sleator, and R E Tarjan. 1986. Making Data Structures Persistent. In *Proceedings of the Eighteenth Annual ACM Symposium on Theory of Computing (STOC '86)*. ACM, New York, NY, USA, 109–121. https://doi.org/10.1145/12130.12142

Richard Fabian. 2013. Data-Oriented Design. (2013). http://www.dataorienteddesign.com/dodmain/dodmain.html

Amos Fiat and Haim Kaplan. 2001. Making Data Structures Confluently Persistent. In *Proceedings of the Twelfth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA '01)*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 537–546. http://dl.acm.org/citation.cfm?id=365411.365528

Matthew Flatt and PLT. 2010. *Reference: Racket.* Technical Report PLT-TR-2010-1. PLT Design Inc. https://racket-lang.org/tr1/.

Mark Galassi, Jim Blandy, Gary Houston, Tim Pierce, Neil Jerram, Martin Grabmüller, and Andy Wingo. 2002. Guile Reference Manual. (2002). https://www.gnu.org/software/guile/manual/guile.html

Erich Gamma, Richard Helm, Ralph E. Johnson, and John Vlissides. 1995. *Design Patterns. Elements of Reusable Object-Oriented Software.* Addison-Wesley.

Andy Georges, Dries Buytaert, and Lieven Eeckhout. 2007. Statistically Rigorous Java Performance Evaluation. *SIGPLAN Not.* 42, 10 (Oct. 2007), 57–76. https://doi.org/10.1145/1297105.1297033

Matthias Grimmer, Chris Seaton, Thomas Würthinger, and Hanspeter Mössenböck. 2015. Dynamically Composing Languages in a Modular Way: Supporting C Extensions for Dynamic Languages. In *Proceedings of the 14th International Conference on Modularity (MODULARITY 2015)*. ACM, New York, NY, USA, 1–13. https://doi.org/10.1145/2724525.2728790

Rich Hickey. 2008. The Clojure Programming Language. In *Proceedings of the 2008 Symposium on Dynamic Languages (DLS '08)*. ACM, New York, NY, USA. https://doi.org/10.1145/1408681.1408682

Howard E. Hinnant, David Abrahams, and Peter Dimov. 2004. *A Proposal to Add an Rvalue Reference to the C++ Language.* Technical Report N1690=04-0130. ISO JTC1/SC22/WG21 – C++ working group.

Ralf Hinze and Ross Paterson. 2006. Finger Trees: A Simple General-purpose Data Structure. *Journal of Functional Programming* 16, 2 (2006), 197–217.

Haim Kaplan. 2005. Persistent data structures. In *In Handbook On Data Structures And applications, CRC Press 2001, Dinesh Meht And Sarta Sahni (Editors) Boroujerdi, A., And Moret, B.M.E., "Persistency in Computational Geometry"; Proc. 7TH Canadian Conf. Comp. Geometry, Quebeq.* 241–246.

Jean Niklas L'orange. 2014. *Improving RRB-Tree Performance through Transience.* Master's thesis. Norwegian University of Science and Technology.

Nicholas D. Matsakis and Felix S. Klock, II. 2014. The Rust Language. *Ada Lett.* 34, 3 (Oct. 2014), 103–104. https://doi.org/10.1145/2692956.2663188

C. Okasaki. 1999. *Purely Functional Data Structures.* Cambridge University Press. https://books.google.de/books?id=SxPzSTcTalAC

Aleksandar Prokopec. 2014. *Data Structures and Algorithms for Data-Parallel Computing in a Managed Runtime.* Ph.D. Dissertation. IC, Lausanne. https://doi.org/10.5075/epfl-thesis-6264

Jon Rafkind, Adam Wick, John Regehr, and Matthew Flatt. 2009. Precise Garbage Collection for C. In *Proceedings of the 2009 International Symposium on Memory Management (ISMM '09)*. ACM, New York, NY, USA, 39–48. https://doi.org/10.1145/1542431.1542438

Michael J. Steindorfer and Jurgen J. Vinju. 2015. Optimizing Hash-array Mapped Tries for Fast and Lean Immutable JVM Collections. *SIGPLAN Not.* 50, 10 (Oct. 2015), 783–800. https://doi.org/10.1145/2858965.2814312

Michael J. Steindorfer and Jurgen J. Vinju. 2016. Towards a Software Product Line of Trie-based Collections. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences (GPCE 2016)*. ACM, New York, NY, USA, 168–172. https://doi.org/10.1145/2993236.2993251

Nicolas Stucki, Tiark Rompf, Vlad Ureche, and Phil Bagwell. 2015. RRB Vector: A Practical General Purpose Immutable Sequence. *SIGPLAN Not.* 50, 9 (Aug. 2015), 342–354. https://doi.org/10.1145/2858949.2784739

D Walker. 2005. Substructural type systems. In *In Advanced Topics in Types and Programming Languages*. The MIT Press.