Delaying arrays

Bachelor Project Report

Alfonso^2 Peterssen
Supervised by Dmitry Petrashko
{first.last}@epfl.ch
LAMP @ EPFL

1. Motivation

Current persistent array implementations provide a reasonable balance between reads and persistent updates; but supporting efficient updates comes at the cost of slowing down read operations.

Since reads are far more frequent than updates, slowing down all reads in exchange for speeding up a few updates seems like a bad compromise.

State-of-the-art persistent arrays (RRB-Vectors) [Stucki et al. 2015] are heavily optimized for the common use cases, namely sequential access and updates. To achieve blazing-fast performance these optimizations wrap additional mutable state behind the immutable interface.

Even with all these optimizations there's still a wide gap between the faux constant-time random access and native arrays performance. An alternative approach to support persistent updates was developed to address this inefficient trade-off; a lazy, persistent array, which evolves and seamlessly adapts to the use pattern. Most notably it provides **real amortized constant-time random access**, while maintaining **constant-time amortized persistent updates**.

2. Persistent re-sizable arrays

A very specific implementation of persistent re-sizable arrays was chosen for this project, having an ideal mix of complexities and trade-offs. The final data-structure: Delaying-array, is just a higher-level abstraction which can be $plugged$ to different implementations under the hood.

2.1 Structure

A sequence of n elements, is stored in several $chunks$, basically, arrays of powers-of-two elements, following the binary representation of the total number of elements: n. The number of $chunks$ needed equals the number of bits in the binary representation of n. The chunks are stored in a $lg(n) - sized$ array, which is an extra level of indirection. This idea is nothing new, similar structured data-structures exists in the functional world [Bagwell 2002] as well as non-persistent (imperative) variants [Brodnik et al. 1999].

This representation allows constant-time random access on architectures providing fast COUNT-LEADING-ZEROS and/or COUNT-TRAILING-ZEROS instructions.

Below is the basic index calculation routine,

```
def apply(idx: Int): Int = {
  val hb = 32 − Integer.numberOfLeadingZeros(idx)
  val mask = (1 << hb) − 1
  val maskedSum = arraysTotalSize & mask
  var arrayId = 0
  var elemId = 0
  if (maskedSum > idx) {
    arrayId = hb − 1
    elemId = idx − (arraysTotalSize & (mask >>> 1))
  } else {
    val clearedIdx = arraysTotalSize & ˜mask
    arrayId = Integer.numberOfTrailingZeros(clearedIdx)
    elemId = idx − maskedSum
  }

  /* return */ arrays(arrayId)(elemId)
}
```

The index calculation cost is dwarfed by the two memory accesses (see last line); but, due to locality, it's very likely that the first memory access is cached in a read intensive scenario, vastly boosting performance.

Unlike current Vector or RRB-Vector implementations, that claims a **faux constant-time random ac-**

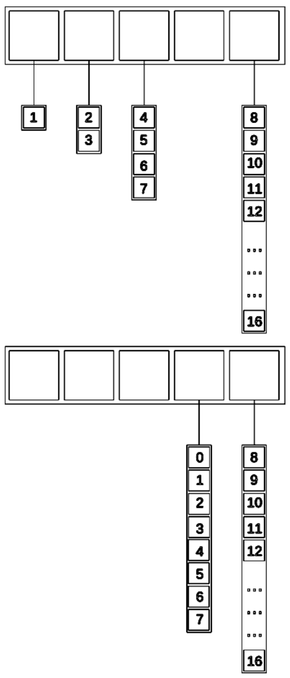

Figure 1. Re-shape after prepending 0, in a resizable array of size 23.

cess, this impementation provides a **real constant-time random access**, completely independent of the number or elements.

2.2 Additional operations

$prepend$, and it's counterpart $tail$ (pop head) are very popular in the functional world. In this case, it's possible to implement these operations in a rather efficient manner, through simple re-shapes.

The aggregated complexity of n prepend operations, is surprisingly just $O(nlgn)$, we can conclude that single-element $prepend$ and $tail$ have $O(lgn)$ amortized complexity.

It's worth mentioning that these operations based on re-shapes are very efficient in practice. Benchmarks show that the performance matches it's counterparts: Vector and RRB-Vector.

Single-element $updates$ still have linear-time complexity, and $appends$ are just not supported, those operations are implemented quite efficiently in the upper level of abstraction, namely Delaying-arrays.

2.3 Lazy updates

Applying a single-element update have linear-time complexity, since a whole $chunk$ must be copied. But instead of eagerly applying updates, one by one, they can be accumulated, and applied all at once, which also have linear-time complexity.

The remaining problem is that reads must be kept in sync with the accumulated updates, so, those grouped updates must be checked on every read, which can seriously slow down reads.

[Okasaki 1999] provides techniques to analyze and improve amortized complexity, the key idea is to keep track and quantify the slow down, and once the slow down surpasses a certain threshold, apply all updates, making reads constant-time again; amortizing the slow down.

2.4 Lazy updates implementation

Several ways were considered, a promising one, mainly because its simplicity consists of storing chained lists of updates. Every $chunk$ keeps a single-linked list of updates, where the head is the most-recent one.

When reading an element from a certain $chunk$, the list of accumulated updates must be traversed, looking for the specified index. In order to quantify the cost of this traversal, the concept of $coins$ is introduced.

Whenever an element of the updates list is checked, a $coin$ is spent. When the number of coins exceeds the size of the $chunk$, all updates are applied. This amortizes the cost of updates and makes future reads constant-time.

Looking for recently updated indices is cheap in terms of $coins$, since only a few recent updates are checked; this works nicely with the fact that rencently updated elements are usually the most frequently accessed.

The trade-off consist of improving performance at the expense of memory, in the worst case the memory needed will be proportional to the number of reads and updates; but this will only happen when persistency is heavily (abu)used (deep nesting).

Accumulated updates must be checked even if there's no update for the searched index; this can be improved, at the expense of memory, by using a Bloom filter.

Pending updates are also applied on re-shapes, since memory will be anyways copied.

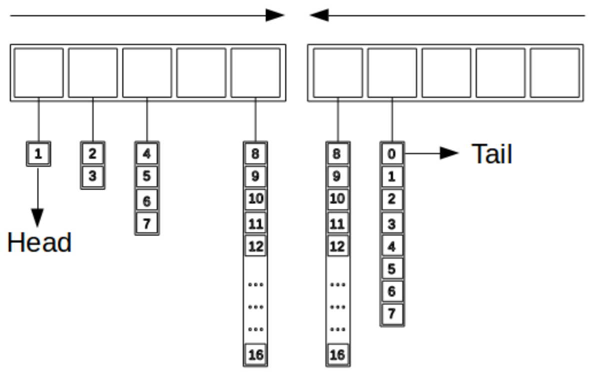

Figure 2. Delaying array's left and right components. The right component is reversed.

3. Delaying arrays

3.1 Structure

A well-known trick, widely used to implement double-ended queues (deques) is used to support *append* operations out of *prepend*-only re-sizable arrays.

3.2 Support for *append*

One consequence of the described structure is that *append* operations can be simply translated into a *prepend* to the right component, keeping the $O(lgn)$ amortized complexity.

3.3 Re-shapes an balance

Keeping a balance between both, left and right components is not needed. The cost of keeping a balance shadows the constant factor (2) that could be saved for just very few operations.

4. Optimizations

4.1 Caching prepends

Prepends are expensive due to frequent re-shapes, which are memory intesive. To improve performance, instead of performing a re-shape on every prepend operation, a cache is added to re-sizable arrays. This cache stores at most 16 elements of the head, when it becomes full it performs a re-shape, merging a *chunk* of 16 elements toghether into the re-sizable array. With this simple optimization, prepends become pretty efficient, reaching equivalent performance to it's counterparts, despite having a worse (by a constant factor) complexity.

4.2 Imperative constructs (ab)use

Scala constructs add a considerable overhead. For performance tuning, Java-like loops and traversals were used, leading to code duplication. Several optimizations were considered but only the less intrusive were implemented, maintaining the readability and structure of the code as much as possible.

5. Benchmarks

A set of microbenchmarks was developed to ease comparison against existing implementations, namely Vector, RRB-Vector and plain arrays.

The Java Microbenchmark Harness (JMH) was chosen over any other fancy benchmarking tools because it's stability and predictability, it also makes very explicit how to properly ensure these properties when using the tool [Shipilev 2014].

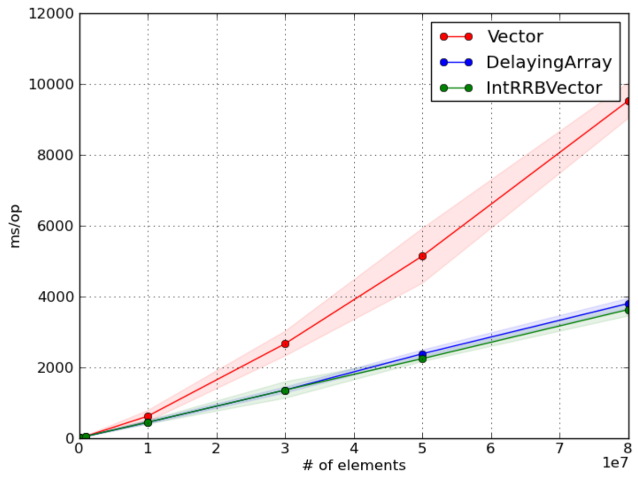

Figure 3. n single-element *appends*. Surprisingly Delaying-array matches IntRRBVector despite having a higher-constant factor.

6. Conclusions

Performance-wise, Delaying-arrays are really fast, most notably for large sequences. Read performance improves with usage at the cost of memory. It proved to be a feasible alternative to existing data-structures for non-online applications.

7. Drawbacks

Delaying-arrays are memory hungry and assumes that memory is an abundant resource. Contrary to intuition, random updates perform better than sequential ones.

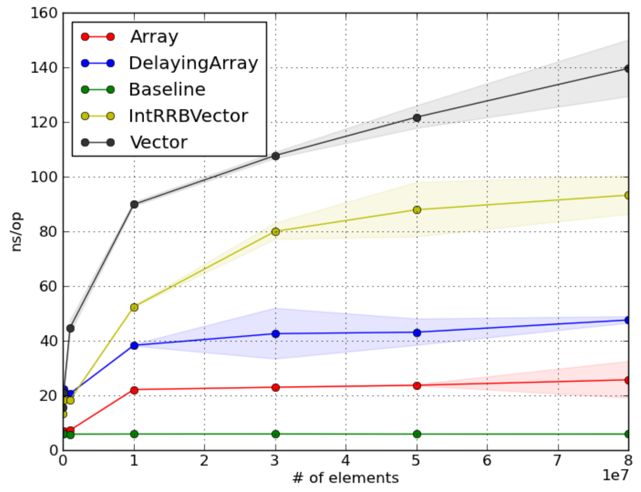

Figure 4. Random-access. Here the **real constant-time access** bests current implementations by a huge margin. Still, Vector and RRB-Vector perform better on small arrays.

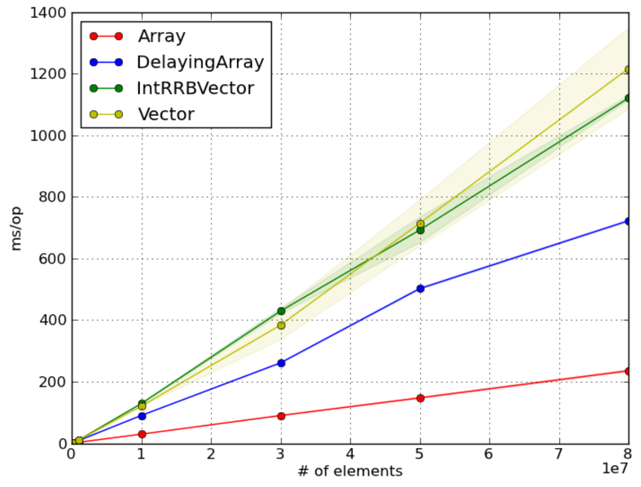

Figure 5. n sequential accesses. Once again, real constant-time complexity gives superior performance, still, RRB-Vector performs really well.

It's still impractical for small sequences, since there's a considerable overhead due to index computation.

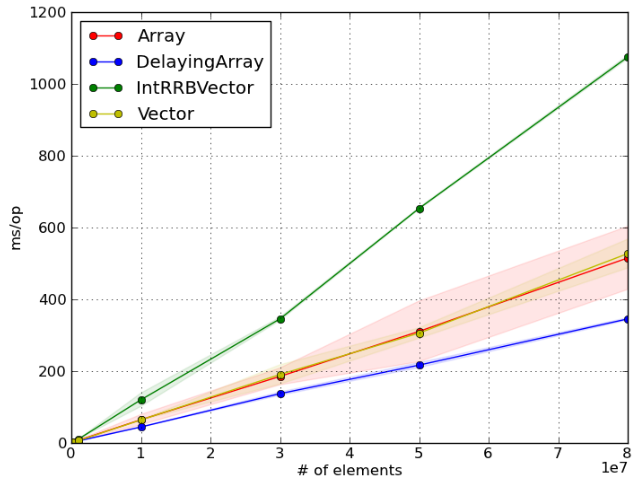

Figure 6. $.foreach$: This benchmark yields confusing results, a quick inspection reveals that these are derived from particular conditions. RRB-Vector $.foreach$ implementation does not take advantage of the RRB-Vector structure to improve traversal, and most surprisingly, native arrays have a huge overhead. This is not a win for Delaying-arrays, it's more of a warning about weaknesses on the other implementations.

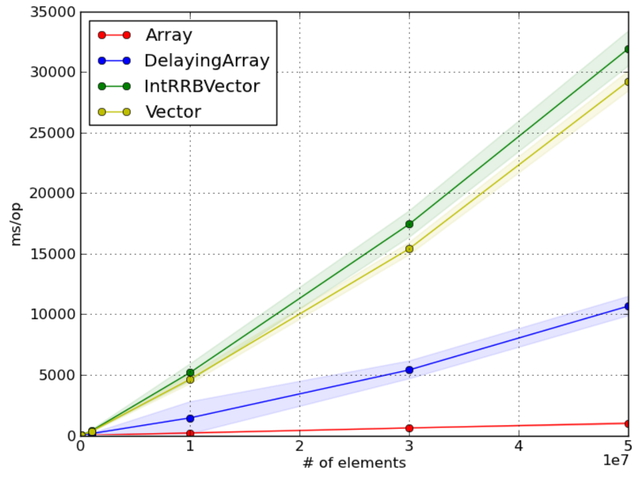

Figure 7. Random single-element updates. Delaying-arrays performs better since, unlike Vector and RRB-Vector, $update$ complexity is independent of the number of elements.

References

P. Bagwell. Fast functional lists, hash-lists, deques and variable length arrays. In *In Implementation of Functional Languages, 14th International Workshop*, page 34, 2002.

A. Brodnik, S. Carlsson, E. D. Demaine, J. I. Munro, and R. Sedgewick. Resizable arrays in optimal time and space. In *Proceedings of the 6th International Workshop on Algorithms and Data Structures*, WADS '99, pages 37–48, London, UK, UK, 1999. Springer-Verlag. ISBN 3-540-66279-0. URL `http://dl.acm.org/citation.cfm?id=645932.673194`.

C. Okasaki. *Purely Functional Data Structures*. Cambridge University Press, New York, NY, USA, 1999. ISBN 0521663504.

A. Shipilev. Nanotrusting the nanotime, 2014. URL `http://shipilev.net/blog/2014/nanotrusting-nanotime/`.

N. Stucki, T. Rompf, V. Ureche, and P. Bagwell. Rrb vector: A practical general purpose immutable sequence. *SIGPLAN Not.*, 50(9):342–354, Aug. 2015. ISSN 0362-1340. doi: 10.1145/2858949.2784739. URL `http://doi.acm.org/10.1145/2858949.2784739`.