**Compas'2020** : Parallélisme / Architecture/ Système/ Temps Réel
MILC - Lyon, France, du 30 juin au 3 juillet 2020

# On optimizing scalar self-rebalancing trees

Paul Iannetta†, Laure Gonnord† et Lionel Morel‡

† Univ Lyon, Université Claude Bernard Lyon 1
CNRS, ENS de Lyon, Inria,
LIP, F-69342, LYON Cedex 07, France
prénom.nom@ens-lyon.fr

‡ Univ Grenoble Alpes, CEA,
F-38000 Grenoble, France
lionel.morel@cea.fr

---

**Résumé**
Balanced trees are pervasive and very often found in databases or other systems which are built around querying non-static data. In this paper, we show that trees implemented as a collection of pointers shows bad data locality, poor cache performance and suffer from a lack of parallelism opportunities. We propose an alternative implementation based on arrays. Both implementations appear to be equivalently efficient time-wise. This new layout exposes new parallelism opportunities which can be then exploited by an optimizing polyhedral compiler.

**Mots-clés :** arbres équilibrés, parallélisme, modèle polyédrique, calcul haute performance

---

## 1. Introduction

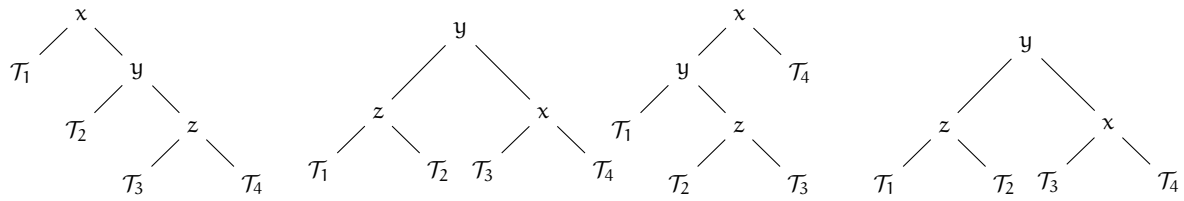
Trees, especially balanced trees, are pervasive and are often found behind data strutures which need to be frequently queried such as sets, maps or dictionaries. Hence, they are often found in traditional databases as T-Trees [12] (a kind of balanced tree built on AVL trees [1, 11]) or B-Trees [11]. The growing need of analyzing large amount of data, gathered from the internet and stored in gigantic databases require harnessing the computing power of high performance parallel machines at their fullest. Improving the processing of trees is part of this endeavor and, a first step in this direction has been made by Blelloch et al. [2, 13, 14] who investigated the benefits of bulk operations to increase parallelism. In this paper, we claim that there is still room for optimizations addressing better cache locality. Indeed, traditional approaches implement trees as collections of pointers. We show that such implementations demonstrate poor data locality, leading to bad performances. Then we explore an alternative representation of those trees with the goal of providing better memory locality, while enabling fine-grained parallelism. Our approach here is to build a memory layout with good properties with respect to the current state of the art compilers and their optimization schemes. This memory layout uses an array instead of a collection of pointers. This induces deep changes to the underlying mechanisms and their complexity (see Section 3). Our goal is to make that this changes in complexity is amortized by better data locality and compiler support for our programs. In particular, not only we will exhibit better opportunities for vectorization, but also we plan to use *polyhedral-model based optimizing compilers* as backend. The polyhedral model [7, 8, 10] is a framework which aims

at increasing the code locality of affine loop by rescheduling their instructions using various methods such as tiling and pipelining. As far as we know, the polyhedral model has never been considered to address programs using complex data structures relying on pointers such as trees, apart from the work of Paul Feautrier and Albert Cohen [5, 6, 9] which uses algebraic languages to describe the *iteration space* over trees. However, unlike Feautrier and Cohen, our approach does not extend the polyhedral model to tree like data structures. Rather, we try and make fit trees into arrays and see to what extent the operations like insertion, find or deletion can be written so as to fall within the reach of the polyhedral model. Our work focuses on standard operations and their parallelization opportunities before tackling bulk operations which is left for the future.
The rest of the paper is organized as follows. Section 2 recalls some background knowledge on AVL trees and the different ways of storing a tree as an array. Section 3 explains in depth our approach and Section 4 presents the optimizations that can be performed and to what extent our redesigned operations fit the polyhedral model. Finally, Section 5 concludes and discusses plans for future work.

## 2. Background

### 2.1. AVL Trees

An AVL tree [1, 11] is a binary search tree such that both of its children are AVL trees and that the absolute difference of theirs heights is strictly less that two. Those trees support the same operations as standard binary search trees (*insert*, *delete* and *find*). However, in order to keep those trees balanced when inserting or removing an element, they also provide a mechanism called *rotation* (see Figure 1 for simple or double rotation examples). An insertion needs at most one rotation to preserve the balance while a deletion may require as much as $\mathcal{O}(\lg n)$ rotations. Nevertheless, since a rotation is an $\mathcal{O}(1)$ operation, and that both insertion and deletion inspect the balance ratio of $\mathcal{O}(\lg n)$ nodes, the complextity of both operations is $\mathcal{O}(\lg n)$. It is important to note that this will not be the case anymore when AVL trees will be stored as arrays because rotations will not be constant time operations anymore. The *find* operation is also $\mathcal{O}(\lg n)$ due to the balanced nature of AVL trees.




(a) A binary tree T (b) T after L-rotation (c) A binary tree T′ (d) T′ after LR-rotation

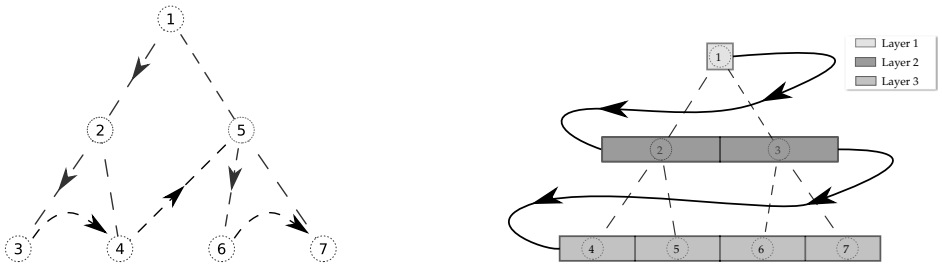
Figure 1: Left (L) rotation applied on to the unbalanced tree T, and a left-right (LR) rotation applied to the unbalanced tree T′

### 2.2. AVL trees as arrays

There are two natural categories of tree traversals: breadth-first and depth-first traversals. Each of them induces a linear order which can be used to index the elements of a tree. This order is not unique and depends on the order in which the children are visited.
**Depth-first numbering.** The *depth-first traversal* of a tree starts from its root, then recursively visits its left child before recursively visiting its right child. This traversal induces an order on the elements of the tree which can then be used to arrange the elements as an array. However,

there is no cheap way to recompute the structure of the tree from this numbering. This is a huge limitation because, in order to perform insertions, deletions, or searching for an element the structure of the tree is needed. A way to keep the structure of the tree is to store two additional arrays: one with the indexes of each node's father and the another with the indexes of each node's right child. Nevertheless, such book keeping is pretty expensive.

**Breadth-first numbering.** The *breadth-first traversal* of a tree starts from its root, then it visits each node at distance 1 of the root, then all nodes at distance 2 of the root, and so on until all nodes have been visited. Again, this traversal induces an order on the elements. This time, however, the structure of the array can be easily conserved if we also keep free holes for unused nodes. This way, the numbering induces layers where the $i^{th}$ layer is $2^i$-wide. The main advantage of this layered representation is to provide an easy way to recover the tree structure from array indices.




(a) Depth-first Numbering (b) Breadth-first Numbering

Figure 2: Array Numberings

We propose to evaluate in the rest of the paper how this new *breadth-first* memory layout can expose optimization opportunities, in particular during the construction of the tree where rotations are the crucial operations.

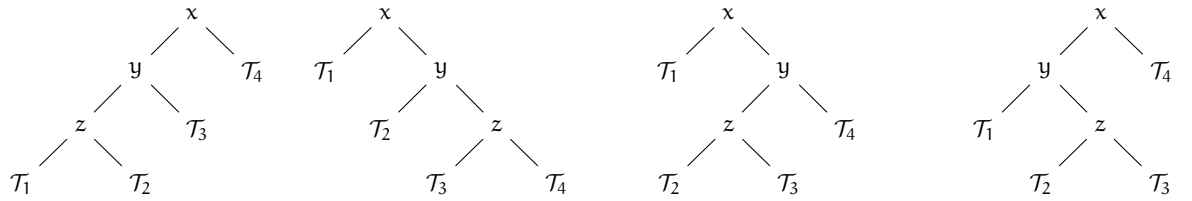
## 3. Rotations on breadth first arrays

Traditionally, a rotation is a cheap operation which: 1. moves around two pointers and 2. updates the information about the heights and the balance ratio of the nodes. However, when trees are internally represented as arrays, rotations become much more expensive because, now, part of the array has to be actually moved from one memory location to another. Hence, the cost of a rotation in the worst case becomes $\mathcal{O}(n)$. This section describes each operation (left and right rotation, but also left-right and right-left rotations) as a sequence of low-level operations on *breadth-first array*, namely *shifts* and *pulls*. On the other hand, the next section will explore how those low-level operations can be more efficient.

### 3.1. Low-level operations on breadth-first arrays

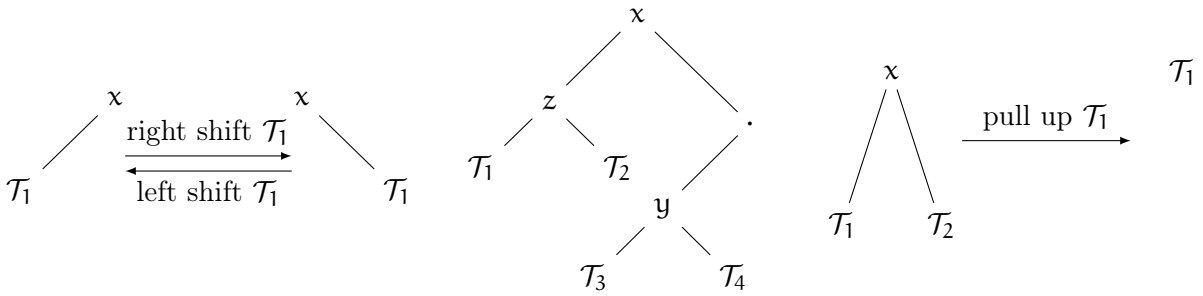
As presented in Section 2.2, breadth-first arrays provide a convenient index scheme which allows to view the array as a collection of layers. The next paragraphs will explain the action of rotations on those layers. First, we present a collection of low-level operations (*shifts* and *pulls*), then, we describe how those low-level operations can be combined to implement rotations.
It should be noted that those low-level operations can be applied on all kinds of breadth-first arrays, by themselves they do not preserve the balancing property of AVL. The combinations presented in Section 3.2 do preserve the balancing property.

**Left and right shifts (Figure 4a).** A *shift* moves a subtree at a certain depth to the left or to the right. The tree is moved such that it is still on the same depth. If we move left the left-most




(a) Needs a L-rotation (b) Needs a R-rotation (c) Needs a LR-rotation (d) Needs a RL-rotation

Figure 3: Unbalanced trees




(a) Left and Right Shifts (b) Pull-down Left (c) Pull-up

Figure 4: Low-level operations

subtree or move right the right-most subtree of a depth then this subtree is removed from the tree. The following picture might suggest that this operation is an involution. Nevertheless, this is not the case. First, the values which are moved overwrite previous values if any. Second, a subtree which is moved out of the tree it is forever lost.

**Pull up (Figure 4c).** A *pull up* takes a subtree and graft it in place of its father. This is a destructive operation in the sense that the father's node as well as one of his children's tree is overwritten. This operation can't be performed on the root because it does not have a father.

**Left and right pulls down (Figure 4b).** A *pull down* takes a subtree and graft it at the place of its right (in the case of a right pull down) or left (in the case of a left pull down) children. Conceptually this operation is not destructive, however, in practice, if the array which is used to represent the tree is of fixed size it is. This can be avoided by padding the tree with an empty layer at the bottom.

### 3.2. Rotations as sequences of low-level operations

Now that the elementary operations have been defined, we can use them to implement rotations as is explained in the following table.

| Rotation<br>Initial Configuration | | Right<br>Figure 3a | Left<br>Figure 3b | Right-left<br>Figure 3c | Left-right<br>Figure 3d |
|---|---|---|---|---|---|
| Steps | 1. | pull down $\mathcal{T}_4$ | pull down $\mathcal{T}_1$ | pull down $\mathcal{T}_1$ | pull down $\mathcal{T}_4$ |
| | 2. | shift right $\mathcal{T}_3$ | shift left $\mathcal{T}_2$ | shift left $\mathcal{T}_2$ | shift right $\mathcal{T}_3$ |
| | 3. | pull up $z$ | pull up $z$ | pull up $\mathcal{T}_2$ | pull up $\mathcal{T}_2$ |
| | 4. | relabel $x, y, z$ | relabel $x, y, z$ | relabel $x, y, z$ | relabel $x, y, z$ |

Unlike the depth-first representation, this time the rotations are not a single in-memory shift but several independent in-memory shifts which are all performed on the same array. The independence of those shifts will be explored in more details in Section 4.

### 3.3. Performance analysis

The goal here is to present and analyze the benchmarks, summarizing the time and the percentage of cache misses during the creation of random trees. The experiments have been done on a machine equipped with an Intel® Core™ i5-5300U CPU @ 2.30GHz with 3072KB of cache. Each test has been run 10 times. The following table records the mean of the results across those 10 calls.

| | | avl-tree | | avl-bf | |
|---|---|---|---|---|---|
| size | density | time (s) | cache-misses (%) | time (s) | cache-misses (%) |
| 64 | 0.55 | 0.001812 | 45 | 0.001659 | 41 |
| 512 | 0.33 | 0.001955 | 48 | 0.001792 | 43 |
| 1024 | 0.33 | 0.001754 | 48 | 0.001943 | 43 |
| 65536 | 0.13 | 0.031252 | 17 | 0.047713 | 36 |
| 524288 | 0.17 | 0.517095 | 46 | 0.648161 | 60 |
| 1048576 | 0.13 | 1.192258 | 47 | 1.509468 | 63 |
| 2097152 | 0.06 | 3.027440 | 47 | 3.779821 | 63 |

As can be seen from the above results neither the tree implementation, nor the array implementation are very good with respect to cache utilization. The number of cache misses is not constant for the tree implementation as may suggest the table. The huge number of cache misses in the array implementation is mostly due to the fact that the addresses are not aligned. The density of breadth first arrays is also a huge concern as the more elements they have, the sparser they become.
However, an interesting fact is that both require almost the time to construct an AVL tree from scratch. For this reason, we decided to investigate the opportunities that could be brought by a more efficient array implementation.

## 4. Optimizing breadth-first arrays

The previous section showed that the usual operations on AVL trees can be performed as a combination of low-level operations on breadth-first arrays. This section will focus on the internal mechanisms of those low-level operations and explore how they could be made more efficient. Most of what is presented focuses on the shift operations, optimizations relative to pulls are only sketched.

### 4.1. Shifts

A shift is an operation which moves internally a subtree to the left or to the right, erasing any previous data and zeroing the source location. Figure 5b illustrates how data moves when the subtree whose root is 2 (the striped region on the figure) is moved to the right into the subtree whose root is 3 (the checkerboard-like region on the figure). Shifting a subtree only modifies the source and destination region, that is why node 1 on the figure remains untouched. It can also be seen that the source region does not overlap with the destination region, this can be hinted to the compiler by using `memcpy` which provides an interface with restrict pointers since C99. The code listing in Figure 5a[1] makes use of that property.
Another point that should be noted is that layers can be moved independently from one another. More precisely, each node can be moved independently of the others. This means that it

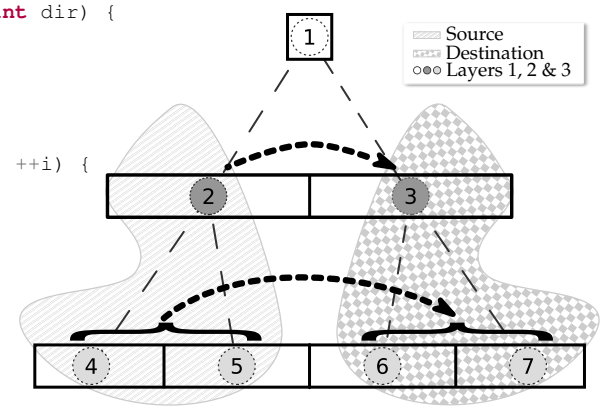
[1] This implementation cannot be used to shift a subtree out of an array. `idx` is the index of the root of the subtree to be shifted, `elems` is a breadth-first array, `len` its length, and `dir` is the shift direction (0 is left and 1 is right.)

```
void _shift (int idx, int * elems, size_t len, int dir) {
  int start_lvl = _greatest_bit_pos(idx + 1);
  int end_lvl   = _greatest_bit_pos(len - 1);
  /* 0 = left -> -1 ; 1 = right -> 1 */
  int dir_coef  = 2 * dir - 1;

  for (int i = 0 ; i < end_lvl - start_lvl + 1 ; ++i) {
    int size  = (1 << i) * (sizeof *elems);
    int src   = (1 << i) * (idx + 1) - 1;
    int dest  = src + dir_coef * depth;

    memcpy(elems + dest, elems + src, size);
    memset(elems + src, 0, size);
  }
}
```

(a) Implementation of the shift operation




(b) Shift inner mechanism

Figure 5: Shift internals

is possible to massively parallelize the shift mechanism. Indeed, the **for** is a parallel for and `memcpy` and `memset` can be distributed over multiple threads as well as vectorized.
While it is possible to add openmp *#pragma* manually our end goal is that an optimizing compiler should be able to detect itself the parallelization opportunities, as well as the vectorization opportunities and to harness them.
When we want to try to expose the inner parallelism of a code mainly using arrays, we cannot help but think of the polyhedral model [7, 8, 10]. However, our code does not quite fit the model because the iteration variable of the **for** loop does not follow a linear pattern. PLUTO [3, 4] a polyhedral loop parallelizer is able to detect and parallelize `memcpy` and `memset`[2] but fails to optimize the parallel for.
Another room for improvement comes from the fact the data could be moved in small chunks whose size should depend on the features of the processor such as its cache size and the size of the registers used by its vector unit.

### 4.2. Pulls

Like *shifts*, *pulls* also offer parallelism and pipelining opportunities. However, unlike for *shifts*, data is moving from one layer to another and the order of operations becomes important. This leads to more complex moving patterns that will be studied in detail in the future.

## 5. Conclusion

In order to address the parallelism issues of self-rebalancing trees, we proposed to change the memory layout to a collection of pointers to an array. Time-wise, the new memory layout performs almost the same as the traditional implementation based on a collection of pointers. We also pointed out that the cost of rotations is higher with this layout but it is possible to alleviate this cost since those can be massively parallelized. In the future, we plan to improve on this work on several directions : first of all, we are working on *breath-first* arrays compression to improve cache performance. Second, the very predictable data layout of these arrays should enable us to push the limits of polyhedral-based compiler optimisations so that to automatically perform bulk operations.

[2] We have rewritten these two functions as loops operating on single elements, as PLUTO is unable to deal with function calls.

**Bibliographie**

1. Adel'son-Vel'skii (G. M.) et Landis (E. M.). – An algorithm for organization of information. *Dokladi Akademia Nauk SSSR*, vol. 146, n2, April 1962, pp. 263–266.
2. Blelloch (G. E.), Ferizovic (D.) et Sun (Y.). – Just join for parallel ordered sets. – In *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures*, *SPAA '16*, p. 253–264, New York, NY, USA, 2016. Association for Computing Machinery.
3. Bondhugula (U.), Baskaran (M.), Krishnamoorthy (S.), Ramanujam (J.), Rountev (A.) et Sadayappan (P.). – Automatic transformations for communication-minimized parallelization and locality optimization in the polyhedral model. – In *International Conference on Compiler Construction (ETAPS CC)*, April 2008.
4. Bondhugula (U.), Hartono (A.), Ramanujam (J.) et Sadayappan (P.). – A practical automatic polyhedral parallelizer and locality optimizer. – In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, *PLDI '08*, pp. 101–113, New York, NY, USA, 2008. ACM.
5. Cohen (A.). – Analyse de flot de données pour programmes récursifs à l'aide de langages algébriques. *Technique et Science Informatiques*, 1999.
6. Cohen (A.). – *Program Analysis and Transformation: From the Polytope Model to Formal Languages*. – Theses, Université de Versailles-Saint Quentin en Yvelines, December 1999.
7. Feautrier (P.). – Some efficient solutions to the affine scheduling problem, I, one-dimensional time. *International Journal of Parallel Programming*, vol. 21, n5, October 1992, pp. 313–348.
8. Feautrier (P.). – Some efficient solutions to the affine scheduling problem, II, multi-dimensional time. *International Journal of Parallel Programming*, vol. 21, n6, December 1992, pp. 389–420.
9. Feautrier (P.). – A parallelization framework for recursive tree programs. – In Pritchard (D.) et Reeve (J.) (édité par), *Euro-Par'98 Parallel Processing*, pp. 470–479, Berlin, Heidelberg, 1998. Springer Berlin Heidelberg.
10. Feautrier (P.). – *Encyclopedia of Parallel Computing*, chap. Polyhedron Model, pp. 1581–1592. – Springer, 2011.
11. Gonnet (G. H.) et Baeza-Yates (R.). – *Handbook of Algorithms and Data Structures in Pascal and C*. – Addison-Wesley Pub (Sd), 1991, 2nd édition.
12. Lehman (T. J.) et Carey (M. J.). – A study of index structures for main memory database management systems. – In *Proceedings of the 12th International Conference on Very Large Data Bases*, *VLDB '86*, p. 294–303, San Francisco, CA, USA, 1986. Morgan Kaufmann Publishers Inc.
13. Sun (Y.) et Blelloch (G.). – Implementing parallel and concurrent tree structures. – In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*, *PPoPP '19*, p. 447–450, New York, NY, USA, 2019. Association for Computing Machinery.
14. Sun (Y.), Ferizovic (D.) et Belloch (G. E.). – Pam: Parallel augmented maps. *SIGPLAN Not.*, vol. 53, n1, February 2018, p. 290–304.