A New Memory Layout for Self-Rebalancing Trees

Paul Iannetta

To cite this version:

Paul Iannetta. A New Memory Layout for Self-Rebalancing Trees. CGO'21, Feb 2021, Séoul, South Korea. hal-03123491

HAL Id: hal-03123491

https://hal.science/hal-03123491

Submitted on 28 Jan 2021

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

A New Memory Layout for Self-Rebalancing Trees

Paul Iannetta*

Univ Lyon 1 UCBL
CNRS, ENS de Lyon, Inria,
LIP, F-69342, LYON Cedex 07, France

paul.iannetta@ens-lyon.fr

Abstract

In this paper, we show that trees implemented as a collection of pointers suffer from a lack of parallelism opportunities. We propose an alternative implementation based on arrays. Both implementations appear to be equivalently efficient time-wise. However, this new layout exposes new parallelism opportunities which can be then exploited by an optimizing compiler.

ACM Reference Format:

Paul Iannetta. 2021. A New Memory Layout for Self-Rebalancing Trees. In *Proceedings of xxxxxx (CGO'21).* ACM, New York, NY, USA, 3 pages.

1 Introduction

Trees are pervasive in systems which need to be frequently queried such as databases. They can take the from of T-Trees [11] (a kind of balanced tree built on AVL trees [1, 9]) or B-Trees [9]. Improving the processing of trees is a necessary endeavor. A first step in this direction has been made by Blelloch et al. [2, 12, 13] who investigated the benefits of bulk operations to increase parallelism. However, we claim that there is still room for optimizations in other directions, such as cache locality. Here, we show that traditional implementations which store trees as collection of pointers can be enhanced by using an array with a layout designed to improve data locality and performance. This new memory layout induces deep changes to the underlying mechanisms (see Section 3). Our goal is that the underlying changes in complexity gets amortized by better data locality and compiler optimizations. In particular, not only we exhibit better opportunities for vectorization, but also we plan to reuse the ideas behind the *polyhedral-model* [5, 6, 8] (a framework which aims at increasing the code locality of affine loops by rescheduling their instructions using various methods such as tiling and pipelining). As far as we know, the polyhedral model has never been used to address programs using complex data structures relying on pointers such as trees, apart from the work of Feautrier and Cohen [3, 4, 7] which uses algebraic languages to describe the *iteration space* over trees. Our approach does not extend directly the polyhedral model to tree-like data structures. Rather, it fits 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 or how ideas behind the polyhedral model can be reused in this context. Here we focus our presentation on the insertion operation and its parallelization opportunities. Bulk operations is left for future work.

*This work was partially funded by the French National Agency of Research in the CODAS Project (ANR-17-CE23-0004-01)

CGO'21, date, place
2021.

2 Breadth-first arrays

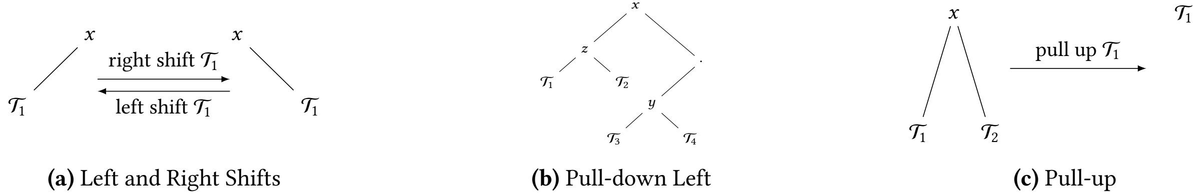
There are two natural categories of tree traversals: breadth-first and depth-first traversals. Both induce an ordering on the nodes which can be used to store the elements in an array. On one hand, the numbering based on the depth-first traversal provides no cheap way to recompute the structure of the tree which is needed to perform insertions, deletions and searches. On the other hand, the numbering induced by the breadth-first traversal can easily store the structure of the tree by keeping holes for unused nodes. This way, the numbering induces layers where the ith layer is 2^i-wide.

3 Rotations on breadth-first arrays

Traditionally, a tree-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 $O(n)$. 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* as well as their performance against a traditional implementation.

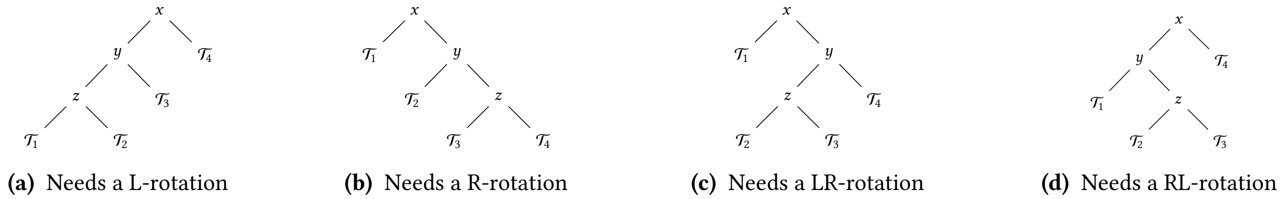
3.1 Low-level operations on breadth-first arrays

As presented in Section 2, breadth-first arrays provide a convenient index scheme which allows to view the array as a collection of layers.



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

Figure 1. Low-level operations



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

Figure 2. Unbalanced trees

Left and right shifts (Figure 1a). 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.

Pull up (Figure 1c). A *pull up* takes a subtree and graft it in place of its father.

Left and right pulls down (Figure 1b). 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.

Those low-level operations can be used to implement the rotation by following the steps in the following table.

	Right Figure 2a	Left Figure 2b	Right-left Figure 2c	Left-right Figure 2d
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

3.2 Performance analysis

The following table presents the results of the insertion of `size` random elements into an breadth-first-array-based AVL tree. Each experiment has been conducted 10 times and the table presents the average. The machine is an Intel® Core™ i5-5300U CPU @ 2.30GHz with 3072KB of cache.

		avl-tree		avl-bf	
size	d (%)	t (ms)	misses (%)	t (ms)	misses (%)
64	55	1.8	45	1.7	41
512	33	1.9	48	1.8	43
65536	13	31.2	17	47.7	36
524288	17	517.1	46	648.1	60
1048576	13	1192.3	47	1509.4	63
2097152	06	3027.4	47	3779.8	63

Both (unoptimized) implementations have similar running times. The number of cache-misses with avl-tree (tree based implementation) is not constant despite what the measure nots tend to show in the table, avl-bf (array based implementation) cache-misses increase steadily as the density (d) (the number of occupied nodes divided by the total number of cells including holes) decrease. However, avl-bf can be compacted to mitigate this problem. The compression process transform a sparse array into an array representing the densest binary search tree with the same elements, this can be done because there exists an equivalent, in this case avl tree, which is close to a prefect binary search tree. This process cost is $O(n)$ and can be performed each time the density reach a fixed threshold. The best value for this threshold is still unknown.

4 Parallelization opportunities

The implementation of the low-level operations can be optimized by applying standard loop transformations. The implementation of the *shift* operation is a parallel for, which means that it can be distributed over many cores easily. *Pulls* operations can also be optimized through the use of pipelining and tiling. More details about the internals of the optimization can be found in our research report [10].

5 Conclusion

We proposed a new memory layout for rotation-based balanced binary search trees and we have showed that despite the change in complexity of the underlying mechanisms, the

naive implementation of both strategies have similar running times and there are strategies to parallelize and further optimize the array-based implementation.

References

[1] G. M. Adel'son-Vel'skii and E. M. Landis. 1962. An algorithm for organization of information. *Dokladi Akademia Nauk SSSR* 146, 2 (April 1962), 263–266.

[2] Guy E. Blelloch, Daniel Ferizovic, and Yihan Sun. 2016. Just Join for Parallel Ordered Sets. In *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures* (Pacific Grove, California, USA) *(SPAA '16)*. Association for Computing Machinery, New York, NY, USA, 253–264. https://doi.org/10.1145/2935764.2935768

[3] Albert Cohen. 1999. Analyse de flot de données pour programmes récursifs à l'aide de langages algébriques. *Technique et Science Informatiques* (1999).

[4] Albert Cohen. 1999. *Program Analysis and Transformation: From the Polytope Model to Formal Languages*. Theses. Université de Versailles-Saint Quentin en Yvelines. https://tel.archives-ouvertes.fr/tel-00550829

[5] Paul Feautrier. 1992. Some Efficient Solutions to the Affine Scheduling Problem, I, One-dimensional Time. *International Journal of Parallel Programming* 21, 5 (October 1992), 313–348.

[6] Paul Feautrier. 1992. Some Efficient Solutions to the Affine Scheduling Problem, II, Multi-dimensional Time. *International Journal of Parallel Programming* 21, 6 (December 1992), 389–420.

[7] Paul Feautrier. 1998. A parallelization framework for recursive tree programs. In *Euro-Par'98 Parallel Processing*, David Pritchard and Jeff Reeve (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 470–479.

[8] Paul Feautrier. 2011. *Encyclopedia of Parallel Computing*. Springer, Chapter Polyhedron Model, 1581–1592.

[9] Gaston H. Gonnet and Ricardo Baeza-Yates. 1991. *Handbook of Algorithms and Data Structures in Pascal and C* (2nd ed.). Addison-Wesley Pub (Sd).

[10] Paul Iannetta, Laure Gonnord, and Lionel Morel. 2020. On optimizing scalar self-rebalancing trees.

[11] Tobin J. Lehman and Michael J. Carey. 1986. 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)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 294–303.

[12] Yihan Sun and Guy Blelloch. 2019. Implementing Parallel and Concurrent Tree Structures. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming* (Washington, District of Columbia) *(PPoPP '19)*. Association for Computing Machinery, New York, NY, USA, 447–450. https://doi.org/10.1145/3293883.3302576

[13] Yihan Sun, Daniel Ferizovic, and Guy E. Belloch. 2018. PAM: Parallel Augmented Maps. *SIGPLAN Not.* 53, 1 (February 2018), 290–304. https://doi.org/10.1145/3200691.3178509