Fully Persistent Arrays

(Extended Abstract)

Paul F. Dietz
Department of Computer Science
University of Rochester
Rochester, NY, USA 14627

dietz@cs.rochester.edu

1 Introduction

A data structure is *partially persistent* if previous versions remain available for queries but not for updates. A data structure is *fully persistent* if past versions remain available both for queries and for updates in a "branching" model of time. The tree representing the branching temporal evolution of the data structure is called the *version tree* [4]. Each node in the version tree represents the result of one update operation on a version of the data structure.

An array is a data structure A on which two operations are defined: $Access(A, i)$, which returns the ith element of A, and $Store(A, i, x)$, which replaces the ith element of A with x. [1] This paper outlines a technique for making arrays fully persistent in $O(\log \log n)$ expected amortized time per operation and $O(n)$ space.[2]

Since an array can model a RAM's memory, any data structure on a RAM in which each operation takes $O(F(n))$ time and performs $O(U(n))$ memory modifications can be made fully persistent using $O(F(n) \log \log n)$ expected amortized time per operation and using $O(nU(n))$ space.

---

[1] Array indices are assumed to be in the range $\{1, \ldots, n\}$, where n is the number of operations to be performed.

[2] All logarithms in this paper are binary.

2 Previous Work

Partially persistent data structures were investigated by Sarnak and Tarjan [7]; fully persistent data structures by Sarnak *et. al.* [4]. They gave a technique for converting a data structure into a (fully) persistent data structure in a constant factor extra time per and space per memory modification. However, their techniques applied only to data structures consisting of a network of records of bounded size and bounded in–degree. In particular, they could not make persistent arrays or data structures containing unbounded fan–in directed graphs.

Fully persistent arrays are related to *context trees*, which are models of inheritance and resemble tree structured association lists [11, 12, 5]. In a context tree, one has two operations: $AddContext(x, name, val)$, which creates a new leaf beneath an existing node x and assigns it the pair $\langle name, val\rangle$, and $LookUp(x, name)$, which finds the deepest ancestor of a node x (including x) which has a pair whose first element is $name$, and returns the associated value (with an error if no such node exists). Fully persistent arrays may be implemented as context trees where one stores (arrayindex,value) pairs at the nodes of the tree.

Dietz [1] showed that context trees may be efficiently implemented using the *order maintenance problem.* This is the problem of maintaining a list of record on which two operations are performed: insert a new record at a specified position in the list, and, given two records in the list, determine which is closer to the beginning of the list. An $O(1)$ amortized time algorithm for OMP was given by Tsakalidis [8] and an $O(1)$ worst case algorithm by Dietz and Sleator [2]. These algorithms permit one to implement fully persistent arrays in $O(1)$ space per update and $O(\log m)$ time per operation, where m is the number of times the array location in question occurs in the version tree. This paper improves this time bound to $O(\min(\log m, \log\log n))$. OMP was used by Sarnak *et. al.* [4] to represent version trees; this representation is described in section 3.1.

Dietzfelbinger *et. al.* [3] showed how to implement partially persistent arrays in $O(\log\log m)$ worst case time for *Access* operations and $O(\log\log m)$ amortized expected time for *Store* operations, where m is again the number of occurences in the version tree (here a tree consisting of a single path) of the array index in question. The result in this paper extends their result to full persistence.

3 An Outline of the Algorithm

3.1 Representing the Version Tree

The version tree is represented by a list $L(A)$ of records, called the *traversal list.* For each vertex v in the version tree the list contains two records, v' and v''. These records represent the first and last times v is visited during a traversal of the tree. When a *Store* is performed on some version v of the array, a new leaf u is added to the tree and is made the first child of v, u' is inserted immediately after v', and u'' after u'.

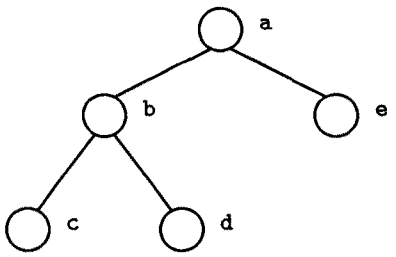

Figure 1: Version Tree

For example, the tree in figure 1 has five nodes a, b, c, d and e and the traversal list

$$L(A) = \langle a', b', c', c'', d', d'', b'', e', e'', a'' \rangle. \tag{1}$$

For each array index i, keep a separate data structure representing those vertices in the version tree which were created by operations that stored into $A[i]$. We say that i *occurs at* these vertices. Call this set of vertices $V(i)$. $V(i)$ partitions the version tree into subtrees, such that a vertex v is in the same subtree as its parent iff v is not the root and $v \notin V(i)$. Let $V(i,1), \ldots, V(i,m_i)$ be the partition induced by $V(i)$. $V(i)$ also partitions $L(A)$ into contiguous sublists in the natural way: two adjacent records are in the same sublist iff they belong to vertices that are in the same set $V(i,j)$.

Lemma 1 *$V(i)$ partitions $L(A)$ into at most $2\|V(i)\| + 1$ sublists.*

Proof: If $\|V(i)\|$ is empty, it partitions $L(A)$ into one sublist, $L(A)$ itself. Now add vertices to $V(i)$ in increasing order of depth. Each additional vertex may split an existing sublist into at most three pieces. The lemma follows by induction. ∎

Lemma 2 *Any record in $L(A)$ is the initial record in sublists induced by at most two $V(i)$'s.*

Proof: Let v be a vertex in the version tree. The record v' can be the initial record in sublists for at most two indices: the index occuring at v, and the index occuring at v's left sibling (if any). The record v'' can be the initial record of a sublist of $L(A)$ only for the index occuring at its rightmost child, if any. ∎

The motivation behind this approach is that we can find the set $V(i,j)$ containing a node x by finding the sublist of $L(A)$ induced by $V(i)$ that contains x', if we label the sublists of $L(A)$ appropriately.

For each array index i, we represent the sublists of $L(A)$ induced by $V(i)$ by storing their initial elements in a balanced tree. Since we can determine the order of list elements in constant time, we can find the sublist containing x' and therefore perform *Access* operations in $O(\log \|V(i)\|)$ time. This is essentially the algorithm of [1] (although the time bound there was not quite as good, because the algorithm presented for OMP was not optimal).

3.2 The Order Maintenance Problem

The idea for speeding up the algorithm is inspired by observing how the order maintenance problem is solved. The list L is broken up into sublists $L_1, \ldots, L_k$, each of length $O(\log \|L\|)$. Inside each sublist, elements are assigned integer labels that increase monotonically along the list. When a new item is inserted into a sublist it is assigned a label equal to the average of the labels of its neighbors, rounded down. When there is no room (two neighbors have consecutive labels) or when the sublist becomes too large it is split into sublists of half size and the elements of each are relabeled evenly, with labels of consecutive elements at least $\|L\|$ apart.

The sublists are placed into another list, where they are also monotonically labeled. There are several algorithms that will monotonically label a list at a cost of $O(\log n)$ amortized relabelings per insertion [1, 8, 2]. Note that in all cases the labels are expressible with $O(\log n)$ bits; i.e., are nonnegative integers bounded by some polynomial in n. We call the problem of maintaining such labels under insertions the *monotonic labeling problem (MLP)*.

Let x and y be elements of L, contained in sublists L_i and L_j. If $L_i \neq L_j$ then the order of x and y can be determined by comparing the labels of L_i and L_j. Otherwise, the order of x and y can be found by comparing their own labels. This takes $O(1)$ time.

By observing that in n insertions the sublists can split at most $O(n/\log n)$ times, we conclude that this algorithm implements OMP in $O(1)$ amortized time per insertion.

3.3 Using $O(\log \log U)$ Priority Queues

Because the labels assigned in the algorithm for OMP are small integers, it is natural to try to store subsets of the list elements using the fast data structure of van Emde Boas, Kaas and Zijlstra [10, 9] (hereafter "VKZ"), in which we can perform insertions, deletions and greatest lower bound queries in $O(\log \log U)$ time, U the bound on the labels. This data structure can be made space efficient using dynamic perfect hashing, at the expense of requiring randomization. Using time stamps instead of list labels, this is the idea behind partially persistent arrays [3].

When implementing fully persistent arrays, however, serious problems arise. The $O(1)$ amortized time algorithms for OMP do not assign labels directly to elements of $L(A)$, but rather to sublists of $L(A)$ of size $O(\log n)$. We might define the label of an element of $L(A)$ to be the label of its sublist in the OMP data structure concatenated with the its label in its sublist. In this case, however, each insertion into $L(A)$ causes $O(\log n)$ (amortized) implicit labels to change. If each of these records is at the beginning of some a sublist for some index i, changing the VKZ data structures may take $O(\log n \log \log n)$ time.

3.4 Bucketing

We overcome these problems with several tricks. In the VKZ data structures, use the "bucketing trick" (see [6], for example). Group together data items into subsets or buckets of size $O(\log^2 n)$. The buckets are contiguous; that is, for two distinct buckets S_1 and S_2 either $x < y$ for all $x \in S_1$ and $y \in S_2$ or all $x > y$ for $x \in S_1$ and $y \in S_2$. Each bucket is represented by a conventional balanced tree of depth $O(\log \log n)$. Store the buckets in a VKZ data structure, indexed by their minimum elements. To insert into the data structure, find the bucket whose minimum element is the greatest minimum element less than the new element, and put the new element into the bucket. When a bucket gets too large split it in half and insert the new fragment into the VKZ data structure.

The bucketing trick means that we only need modify a VKZ data structure when the label of the minimum record of a bucket is changed. Since each record occurs only $O(1)$ times, there are at most $O(n/\log^2 n)$ such bucket headers. Since each relabeling by the OMP algorithm causes $O(\log n)$ implicit labels to change, we might expect an average of only $O(\log^{-1} n)$ bucket headers to be relabelled on any operation.

3.5 Weighted Monotonic Labeling

Unfortunately, the distribution of bucket headers in sublists of $L(A)$ may be nonuniform. To solve this problem, we modify the monotonic labeling algorithm to take into account the cost of relabeling a sublist. Let the function $h(r)$ be 1 if r is a bucket header, 0 otherwise. For each sublist L_i of $L(A)$ produced by the OMP algorithm, define the *cost* of relabeling L_i to be

$$cost(L_i) = 1 + \sum_{r \in L_i} h(r) \log \log n. \tag{2}$$

The cost is proportional to the time we must spend when the sublist is relabeled. If there are k sublists of $L(A)$, the sum of the costs is $(1 + o(1))k$.

The weighted analogue of the problem of maintaining monotonic labels under insertions is the *weighted monotonic labeling problem (WMLP)*. This is the problem of performing the following operations on a list L:

$ChangeCost(x, \Delta)$	Change the cost of list item x from $cost(x)$ to $cost(x)+\Delta$.
$Split(x, c_1)$	Split the list element x into two list elements x_1 and x_2 with costs c_1 and $cost(x) - c_1$.

Assume the costs are positive integers. If the sum of the costs of the elements of L are always $O(\|L\|)$, and we assume it takes time $cost(x)$ to relabel x, we now there is an algorithm for WMLP that takes $O(|\Delta| \log \|L\|)$ amortized time for $ChangeCost$ operations and $O(cost(x) + \log \|L\|)$ amortized time for $Split$ operations.

The idea behind the algorithm is to surround each element x with $cost(x)$ dummy elements on each side, and use the existing algorithms for MLP on this augmented list. The motivation

is that the existing labeling algorithms have the property that they renumber list elements in a contiguous region around the insertion point. Therefore, if some "real" list element x is renumbered, at least $cost(x)$ dummy elements are also renumbered, and the cost of renumbering x will be at most a constant factor larger than the number of relabelings of dummy elements. The upper bound on the MLP algorithm of $O(\log \|L\|)$ amortized relabelings per insertion, together with the linear bound on the total number of dummy elements, yields the desired result.

We now describe in more detail how *Split* and *ChangeCost* operations are performed:

1. When performing $ChangeCost(x, \Delta)$, if $\Delta < 0$ then we delete $2|\Delta|$ elements from the list. This takes time $O(|\Delta| \log \|L\|)$ (actually, it can be done in $O(\Delta)$ time). If $\Delta > 0$, we insert $2\ \Delta$ additional dummy elements around x. We do the insertion as far from x as possible, so that if x is renumbered then so are $cost(x)$ dummy elements. This takes time $\Delta \log \|L\|$.

2. When performing a $Split(x_1, c)$ operation, we must split the existing $2\ cost(x_1)$ dummies into two groups of size $2\ c$ and $2\ (cost(x_1) - c)$, respectively. We delete x_1 and insert it and a new element x_2 at the proper positions in the dummy list. This takes times $O(cost(x) + \log \|L\|)$.

3.6 Using WMLP for Fully Persistent Arrays

We use the algorithm for WMLP to implement OMP. We again break the list $L(A)$ into sublists, only now we limit each sublist to have cost (as defined in equation 2) $O(\log \|L(A)\|)$. When a sublist is split it splits into two sublists of nearly equal cost (the costs can be made equal to within $\Theta(\log \log n)$, and are always $O(\log n)$).

When a bucket splits in the VKZ data structure, a new element of $L(A)$ becomes a bucket header. The cost of the sublist of $L(A)$ containing that element increases by $O(\log \log n)$. Using *ChangeCost*, the cost of the sublist is increased, taking time $O(\log n \log \log n)$. Because this can happen at most $O(n/\log^2 n)$ times in n operations, the amortized cost due to *ChangeCost* operations is $O(\log \log n / \log n)$ per operation, which is not significant.

Because the cost of a sublist is proportional to the actual time needed to fix up the VKZ data structures when that sublist's label is changed, we can conclude:

Theorem 3 *Fully persistent array operations can be performed in* $O(\log \log n)$ *amortized time per operation.*

If dynamic perfect hashing is used to store the VKZ data structures then space is $O(n)$, but the time bound is degraded to expected time by the need to perform randomization.

4 Summary

This paper has outlined an algorithm for fully persistent arrays in which *Access* operations take $O(\log \log n)$ time and *Store* operations take $O(\log \log n)$ amortized expected time. The algorithm uses linear space.

Because any data structure can be implemented with an array, we can use the techniques described in this paper to make any data structure fully persistent at a cost of an extra factor of $\log \log n$ per operation and using space proportional to the number of memory modifications. Of course, a data structure that is efficient only in the amortized sense may perform poorly if made fully persistent, because expensive operations may be replicated on many branches of the version tree.

References

[1] Paul F. Dietz. Maintaining order in a linked list. In *Proc. 14th ACM STOC*, pages 122–127, May 1982.

[2] Paul F. Dietz and Daniel D. Sleator. Two algorithms for maintaining order in a list. In *Proc. 19th ACM STOC*, pages 365–372, May 1987. Submitted to JCSS. A revised version of the paper is available that tightens up the analysis.

[3] M. Dietzfelbinger, A. Karlin, K. Melhorn, F. Meyer auf der Heide, H. Rohnert, and R. E. Tarjan. Dynamic perfect hashing: Upper and lower bounds. In *Proc. 29th FOCS*, pages 524–531, 1988. The application to persistent arrays was presented but does not appear in the proceedings.

[4] James R. Driscoll, Neil Sarnak, Daniel D. Sleator, and Robert E. Tarjan. Making data structures persistent. In *Proc. 18th ACM STOC*, pages 109–121, May 1986.

[5] Carlo Montangero, Giuliano Pacini, Maria Simi, and Franco Turini. Information management in context trees. *Acta. Info.*, 10:85–94, 1978.

[6] Mark Overmars. A $O(1)$ average time update scheme for balanced binary search trees. *Bull. EATCS*, pages 27–29, 1982.

[7] Neil Sarnak and Robert E. Tarjan. Planar point location using persistent search trees. *Communications of the ACM*, 29(7):669–679, July 1986.

[8] A. K. Tsakalidis. Maintaining order in a generalized linked list. *Acta. Info.*, 21(1):101–112, 1984.

[9] P. van Emde Boas. Preserving order in a forest in less than logarithmic time and linear space. *Info. Proc. Lett.*, 6(3):80–82, June 1977.

[10] P. van Emde Boas, R. Kaas, and E. Zijlstra. Design and implementation of an efficient priority queue. *Math. Sys. Theo.*, 10:99–127, 1977.

[11] Ben Wegbreit. Retrieval from context trees. *Info. Proc. Lett.*, 3(4):119–120, March 1975.

[12] Ben Wegbreit. Faster retrieval from context trees. *Communications of the ACM*, 19(9):526–529, September 1976.