Fully Persistent Arrays for Efficient Incremental Updates and Voluminous Reads

Tyng-Ruey Chuang
Department of Computer Science
Courant Institute of Mathematical Sciences
New York University*

Abstract

The array update problem in a purely functional language is the following: once an array is updated, both the original array and the newly updated one must be preserved to maintain referential transparency. We devise a very simple, fully persistent data structure to tackle this problem such that

- each incremental update costs $O(1)$ worst-case time,
- a *voluminous* sequence of r reads cost in total $O(r)$ amortized time, and
- the data structure use $O(n + u)$ space,

where n is the size of the array and u is the total number of updates. A sequence of r reads is voluminous if r is $\Omega(n)$ and the sequence of arrays being read forms a path of length $O(r)$ in the version tree. A voluminous sequence of reads may be mixed with updates without affecting either the performance of reads or updates.

An immediate consequence of the above result is that if a functional program is single-threaded, then the data structure provides a simple and efficient implementation of functional arrays. This result is not new. What is new is that many multi-threaded functional array applications also exhibit the incremental updates/voluminous reads execution pattern. Those applications can also be efficiently implemented by the proposed data structure.

A comparison of our method to previous approaches to the array update problem is briefly discussed. Empirical results have been collected to measure the effectiveness of the proposed data structure.

1 Survey and Motivation

The array update problem in the implementation of a purely functional programming language is the following: once an array is updated, both the original array and the newly updated one must be preserved, preferably at a small cost, to maintain the referential transparency of functional programs. Copying the whole array is usually regarded as being too inefficient. Depending on different perspectives of this problem, there have

*Author's address: 251 Mercer Street, New York, NY 10012, U.S.A. E-mail: chuang@cs.nyu.edu.
This research has been supported, in part, by the National Science Foundation (#CCR-8909634) and DARPA (DARPA/ONR #N00014-91-J1472).

been various approaches to solve it. We classify three popular approaches to the array update problem. They are the compile–time analysis approach, the language restriction approach, and the run–time data structure approach. We briefly state the advantage and disadvantage of the three approaches, as well as our motivation to design yet another run–time structure to tackle this problem.

1.1 Compile–Time Analysis

Let us use the term *access* to include both read and update operations to a data structure. Schmidt [21] defines a program to be *single–threaded* if all accesses to the variables in the program only refer to the most recent versions. A program is called *multi–threaded* if it is not single–threaded. A non–standard semantics can be devised to detect, at compile time, single–threaded accesses to arrays in a functional program and, in those cases, to generate code to update arrays destructively. This approach is taken, for example, in the works of Schwartz [22,23], Hudak & Bloss [16], Bloss [9,10], and Odersky [19].

The advantage of this approach is that it is possible to generate very efficient code because no overhead is spent in maintaining multiple versions of arrays. The disadvantage is that the analyses usually assume some particular evaluation order of a functional program, which is unspecified in the standard semantics. Also, they are incomplete, as are all interesting semantics analyses, in the sense that there remains programs which are single–threaded but not detected by the analyses. In addition, such analyses are expensive (they may have exponential time complexity with respect to program size) and not universal (they currently apply to first–order languages only).

1.2 Language Restriction

Instead of using compile–time analysis to detect single–threaded programs, a functional programming language can be restricted so that only single–threaded programs can be expressed. These restrictions are usually expressed in terms of type rules. Therefore, for well–typed programs, no compile–time analysis for detecting destructible updates is ever needed and all array updates can be done destructively and efficiently. This approach has been explored by Guzmán & Hudak [14] and Wadler [25,26]. The remaining task is to check, at compile time, whether or not a program is well–typed with respect to the single–threading type scheme. The type checking algorithms are usually complicated.

A common drawback of the compile–time analysis approach and the language restriction approach is that they cannot deal with programs which mostly use arrays in a multi–threaded manner. In those programs, old versions of an array must be kept around for possible future accesses, and few destructive updates may be performed.

1.3 Run–Time Data Structure

There also has been much effort to make various data structures persistent such that, during a series of modifications, the old versions, as well as the newest version, of a data structure can still be accessed. Following the terminology of Driscoll, Sarnak, Sleator & Tarjan [13], a data structure is *partially persistent* if all versions can be read but only the newest version can be updated. A data structure is *fully persistent* if every version can be both read and updated.

An array can be easily made fully persistent if it is represented as a balanced search tree. But then we lose much of an array's constant–time accessibility because a read or update operation will take $O(\log n)$ time for an array of size n. Dietz [12] proposes a sophisticated method to achieve $O(\log\log n)$ amortized access time when the arrays are large (say, $n = 2^{10}$) and the total number of operations to be performed is n.

Some techniques, which are called *reversible difference list* or *trailer*, have been used to implement fully persistent arrays. They can be found, for examples, in the works of Holmstrom [15], Hughes [18], Aasa, Holmstrom & Nilsson [6] and Bloss [9,10]. These techniques seem to be variations of the shallow binding scheme devised by Baker [7,8]. Under these techniques, an access to the newest version will take $O(1)$ time; but an access to an old version will take time linear to the number of differences between the old version and the current version. These techniques are good for single–threaded programs because those programs always access the newest version of an array. However, they are bad for multi–threaded programs because accesses to old versions are costly.

A common drawback of the run–time data structure approach is the overhead for maintaining multiple versions of a data structure, even when only the newest version is needed. The storage occupied by inaccessible versions of a data structure may have to be reclaimed.

1.4 Motivation and Outline

Previous attempts to make arrays fully persistent do not work well in multi–threaded cases because they use too little random access memory (RAM) to represent multiple versions of an arrays. Instead, they depend mostly on indirect reference by pointers. In fact, previous data structures use only one set of RAM for multiple versions of an array. An array's random accessibility is then lost once there have been many updates being performed to the array.

We propose here a variation of Baker's shallow binding scheme which holds multiple sets of RAM. We call this method the *fragmented shallow binding scheme*. This scheme employs multiple sets of RAM, which we will later call *caches*, to improve the efficiency of those programs which access arrays in a multi–threaded way. Although there is still a restriction — reads have to be voluminous — it seems that a major portion of multi–threaded applications do fit the incremental updates/voluminous reads restriction.

The outline of this paper is the following. After a short introduction to the shallow binding scheme in section 2, we describe in section 3 how to implement fully persistent arrays by the fragmented shallow binding scheme. Section 4 contains detailed complexity analysis of the proposed scheme. Empirical results are presented in section 5. Section 6 is a short discussion of related issues.

2 Deep Binding and Shallow Binding

In a higher–order functional language, where functions can be passed as arguments and returned as results, there may be more than one accessible environment during a program's execution. In an implementation of a higher–order functional language, it is essentially important to design an efficient data structure to support the following two operations: variable lookup in the current environment and context switch between environments.

In the *deep binding* scheme, the environments are represented as a *context tree* where each node in the tree introduces a new binding of a value to a variable. An environment is represented as a path from its most recent binding to the initial binding in the context tree. The lookup for a variable v in an environment E is performed by searching the path of E, looking for the most recent value bound to v. Under the deep binding scheme, a context switch between environments costs constant time but a variable lookup in an environment may cost time linear to the length of its binding path.

In order to improve the performance of variable lookup, Baker [7] develops the *shallow binding* scheme where a *cache* is introduced to record the bindings of variables in the *current environment.* Variable lookup in the current environment is performed by a constant–time reference to the cache. The current environment is also made the root of the context tree. However, only one environment, the current environment, possesses the cache. Other non–current environments are still represented as binding paths leading to the current environment.

A context switch between environments in the shallow binding scheme will involve a sequence of *rotations* in the context tree. The entire process for a sequence of rotations is called *rerooting.* A rotation is performed between the current environment (*i.e.*, the root node in the context tree) and one of its children. Suppose that the current environment E binds variable v to value a in its cache, and its child node n binds v to b. Then a rotation between node E and node n will make n the current environment and E n's child. Node n will inherit node E's cache and will change variable v's value to b. Node E will only record the binding of variable v to value a; it will not possess the cache anymore. A rerooting is a sequence of rotations to make the desired environment current.

Figure 1 pictures how rotation and rerooting work under the shallow binding scheme. We will show how to use, and to modify, the shallow binding scheme to implement functional arrays in the next section.

3 Functional Arrays and Their Implementations

A data structure for functional arrays must support the following operations.

- *Create* n: Return an array of size n. Each entry of the array is not initialized.
- *Update* A_j i v: Return an array $A_{j'}$ which is functionally identical to array A_j except $A_{j'}(i) = v$. Array A_j is not destroyed and can be accessed further.
- *Read* A_j i: Return $A_j(i)$.

We can use the shallow binding scheme to implement *Create, Update*, and *Read* as the following.

- *Create* n:
  Allocate a cache of size n. Allocate a node of one field and have this field point to the cache. This node is the *root node.* Return the address of the root node.
- *Update* A_j i v:
  Allocate a node of three fields and have it store i, v, and A_j (note that A_j is an address). Return the address of this newly allocated node.

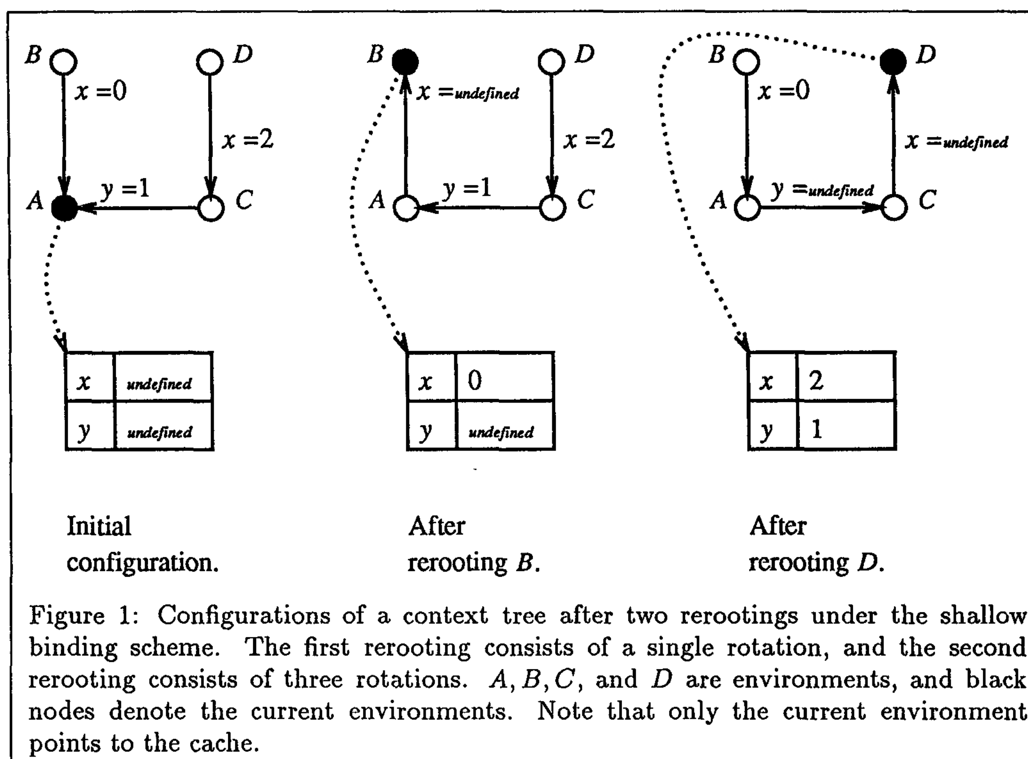

Figure 1: Configurations of a context tree after two rerootings under the shallow binding scheme. The first rerooting consists of a single rotation, and the second rerooting consists of three rotations. A, B, C, and D are environments, and black nodes denote the current environments. Note that only the current environment points to the cache.

- *Read* A_j i:
  Do a rerooting starting from the node pointed to by A_j. After the rerooting, A_j points to the root node which again points to a cache. Return the ith entry in the cache.

For example, the three configurations in figure 1 can be thought as being resulted from the following three successive sequences of operations,

- $A =$ *Create* 2; $B =$ *Update* A i_x 0; $C =$ *Update* A i_y 1; $D =$ *Update* C i_x 2;
- *Read* B i; and
- *Read* D j.

We now use the term *version tree* to refer to the context tree in the shallow binding implementation of functional arrays. It is easy to verify that a *Create* n operation takes $O(n)$ time and uses $O(n)$ space. Each *Update* takes $O(1)$ time and additional $O(1)$ space. A *Read* uses no space but takes time proportional to the distance between the new root node and the old root node, which can be as large as the total number of updates being performed. However, after a *Read* A_j i operation, each additional read to either array A_j, or one of A_j's children in the version tree, will cost only additional $O(1)$ time. This is because, after the *Read* A_j i operation, A_j points to the root node and, in case the read is directed to one of A_j's children, the rotation between A_j and one of its children only costs $O(1)$ time.

It is not difficult to see why we insist on voluminous reads now. Because, after the initial read of a voluminous sequence, each read of the sequence will cost only additional $O(1)$ time. However, there is still one problem. The distance between the new root node and the old root node may be far greater than the volume of the reads, which makes rerooting expensive. The trick is to allocate caches during a long rerooting and, at the same time, keep the total numbers of entries in all caches proportional to the total numbers of updates being performed. By performing this trick, we make sure that the large cost of a long rerooting can be paid off by the small cost of future shorter rerooting. By keeping the total number of entries in caches proportional to the total number of updates, we make sure that only linear space (with respect to the total number of updates) is used.

Before describing the details of the new implementation scheme for efficient increment updates and voluminous reads, let us first define *VRead*, the voluminous read operation.

- *VRead* $[A_{j_0}, A_{j_1}, \ldots, A_{j_{l-1}}]$ $[i_0, i_1, \ldots, i_{l-1}]$: Return a list of l elements, where the list's kth element has value $A_{j_k}(i_k)$. It is required that l be $\Omega(n)$ and the path consisting of $A_{j_0}, A_{j_1}, \ldots, A_{j_{l-1}}$ in the version tree be of length $O(l)$. n is the array size. [1]

Notice that *Read* will also be supported by the new data structure. However, as we will see, an individual *Read* will cost $O(n)$ amortized time in the worst cases.

We now describe the *fragmented shallow binding scheme*, which supports *Create, Update, Read,* and *VRead*. The implementation of *Create* and *Update* are the same as before. *Read* and *VRead* are implemented in the following way.

- *Read* A_j i:
  Let d be the distance between the node p pointed to by A_j and the root node of the tree in which p resides.
  - If $d \leq 2n$, do a rerooting starting from node p and then return the ith entry in the cache pointed to by p. Note that node p is the root node after the rerooting.
  - If $d > 2n$, then cut the link between node p and the root node equally into $k = \lceil \frac{d}{n} \rceil$ segments such that each segment has at least $\lfloor \frac{d}{k} \rfloor$ nodes. Call these segments $s_0, s_1, \ldots, s_{k-1}$. Let H_i and T_i respectively be the first node and last node of segment $s_i, 0 \leq i \leq k-1$. Note that H_0 is the root node and T_{k-1} is node p.
    Do a rerooting starting from node T_0, duplicate the cache pointed to by T_0, allocate a new root node and have it point to the new cache. Make H_1 point to this newly allocated root node. Repeat the above procedure for T_1 and H_2, T_2 and $H_3, \ldots, T_{k-2}$ and H_{k-1}, and T_{k-1}. When all rerooting is done, we have cut the tree in which node p originally resides into k disjoint trees such that each tree has its own root node (which points to its own cache).
    After the cutting and rerooting, A_j points to root node T_{k-1} which again points to a cache. Return the ith entry in the cache.

[1] $O(f(x))$ is defined as the set of all functions $g(x)$ such that there exist positive constants C and x_0 with $|g(x)| \leq C\ f(x)$ for all $x \geq x_0$. $\Omega(f(x))$ is the set of all functions $g(x)$ such that there exist positive constants C and x_0 with $g(x) \geq C\ f(x)$ for all $x \geq x_0$ (see [20], for example). In the definition of *VRead*, it simply says that the volume of the read sequence is at least the magnitude of the size of the array, and the length of the path is at most the magnitude of the volume of the read sequence.

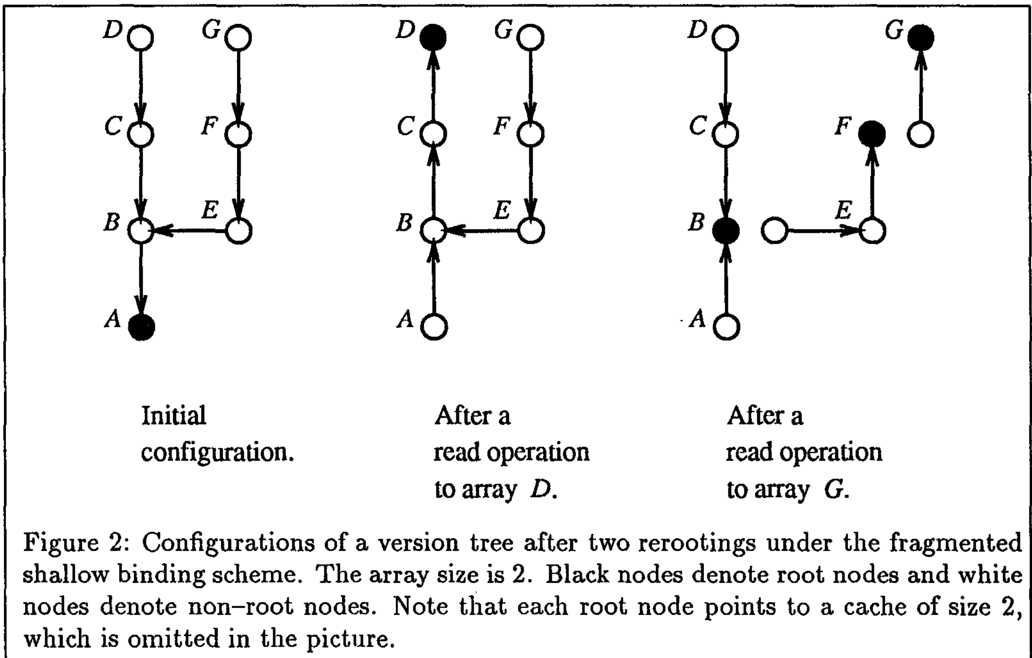

Figure 2: Configurations of a version tree after two rerootings under the fragmented shallow binding scheme. The array size is 2. Black nodes denote root nodes and white nodes denote non–root nodes. Note that each root node points to a cache of size 2, which is omitted in the picture.

- *VRead* $[A_{j_0}, A_{j_1}, \ldots, A_{j_{l-1}}]$ $[i_0, i_1, \ldots, i_{l-1}]$:
  Perform (in their specified ordering) the following operations: $v_0 = \mathit{Read}\ A_{j_0}\ i_0$; $v_1 = \mathit{Read}\ A_{j_1}\ i_1$; ...; and $v_{l-1} = \mathit{Read}\ A_{j_{l-1}}\ i_{l-1}$. Return $[v_0, v_1, \ldots, v_{l-1}]$.

Figure 2 illustrates what a version tree of an array of size 2 will look like after two rerootings under the fragmented shallow binding scheme. The three configurations in figure 2 can be thought as resulting from the following three successive sequences of operations,

- $A = \mathit{Create}\ 2$; $B = \mathit{Update}\ A\ i_b\ b$; $C = \mathit{Update}\ B\ i_c\ c$; $D = \mathit{Update}\ C\ i_d\ d$; $E = \mathit{Update}\ B\ i_e\ e$; $F = \mathit{Update}\ E\ i_f\ f$; $G = \mathit{Update}\ F\ i_g\ g$;
- *Read* $D\ i$; and
- *Read* $G\ j$.

In order to analyze the time and space complexity of the above implementation, let us first introduce some terminology. Since a rerooting may cut a tree into a set of disjoint trees, the resulting data structure is more likely to be a forest rather than a tree. Among the many nodes in a particular tree of the forest, we will use the term *backward gate* (*b–gate* for short) for the node whose link between itself and its parent has been cut. Similarly, a *forward gate* (*f–gates* for short) is a node whose link to one of its children has been cut.

Suppose that each tree in the forest is viewed as a *region* and an imaginary bridge is used to connect those regions that were connected previously. Then it is not difficult to see that the resulting region graph is also a tree. Let us call it a *region tree.* Note that

the root region in the region tree has no b–gate and the leaf regions have no f–gates. To simplify the analysis, let us assign the b–gate of the root region to the root node of the version tree (as if no rerooting had ever occurred). Also note that the locations of b–gates and f–gates in a region will not change once they are created, regardless of further operations being performed on the data structure. For example, in the final configuration in figure 2, the region including nodes A, B, C, and D has A as b–gate and B as f–gate. The region including nodes E and F has E as b–gate and F as f–gate. The region including node G has G as b–gate but has no f–gate.

It can be easily shown that once cutting has occurred in the data structure, then each region contains at least $\lfloor \frac{2n+1}{3} \rfloor$ non–root nodes, where n is the array size.

4 Time and Space Complexity of the Fragmented Shallow Binding Scheme

We will use the potential method described by Tarjan [24] to analyze the amortized time complexity of a sequence of *Update* and *VRead* operations, starting from a single *Create* operation. A potential function Φ maps any configuration D of a data structure into its potential $\Phi(D)$, which can be viewed as the amount of energy stored in the configuration. The *amortized time* of an operation is defined to be $t+\Phi(D')-\Phi(D)$, where t is the actual time needed by the operation, and D and D' are the configurations of the data structure before and after the operation, respectively. We can regard amortized time as a fixed amount of cost. If this fixed amount is larger than the actual need, then the remaining amount is stored in the data structure for future use. If the fixed amount is less than the actual need, then the energy released from the data structure will compensate the difference.

With this definition, the total actual time for a sequence of m operations can be written as

$$\sum_{i=1}^{m} t_i = \sum_{i=1}^{m} (a_i - (\Phi_i - \Phi_{i-1})) = \sum_{i=1}^{m} a_i - (\Phi_m - \Phi_0),$$

where t_i and a_i are the actual time and amortized time of the ith operation, respectively. Φ_i is a shorthand for $\Phi(D_i)$. Φ_0 is the initial configuration of the data structure and Φ_m is the final one. If the difference between Φ_m and Φ_0 remains positive, then the total amortized time is an upper bound of the total actual time.

We define the potential of a configuration D in the fragmented shallow binding scheme as

$$\Phi(D) = 3(NR(D) - n \cdot R(D)) + \sum_{S \in D} W(S)$$

where $NR(D)$ is the total number of non–root nodes in D, $R(D)$ is the total number of root nodes in D, n is the size of array, and $W(S)$ is the weight of a region S. The weight of a region is defined as the distance between the root node and the b–gate in the region. It is clear that the minimal weight of a region is 0 and the maximal weight can be as large as the total number of non–root nodes in the region. Also note that a region's weight stores the exact amount of energy to move the root node of the region to the region's b–gate.

The intuition behind the definition of the potential function is that the more root nodes (*i.e.*, the more regions, hence, the more caches) a configuration has, the less potential it keeps; and, for each region, the more distant its cache is from its b–gate, the more potential it has. The constant 3 in the potential function Φ may vary if we change the way cutting is performed during a long rerooting. The constant 3 is chosen here because a link is cut only when its length is roughly 3 times larger than the size of the array. It also makes the following lemma true.

Lemma 4.1 If a reroot operation involves cutting the data structure, then the amortized time of the rerooting is less than or equal to 0. □

PROOF. If cutting occurs during a rerooting, then the distance d between the old root node and new root node in the original region must be greater then $2n$, where n is the array size. Let $d = (k+3)n - r$, where $0 \leq k$ and $0 \leq r \leq n-1$. After the rerooting, the original region is cut into $k+3$ regions, and the difference between the total weight of the $k+3$ regions and the weight of the original region is less than or equal to d. That is, $\sum_{S \in D'} W(S) - \sum_{S \in D} W(S) \leq d$, where D and D' are, respectively, the configurations of the data structure before and after the rerooting.

Let a be the amortized time of the rerooting and t be the actual time. We have $\Phi(D') - \Phi(D) \leq -3(k+2)n + d$. It is clear that d units of actual time suffices to perform the rerooting. We then have the amortized time

$$a = t + (\Phi(D') - \Phi(D)) \leq d + (-3(k+2)n + d) = -kn - 2r \leq 0.$$

◇

Before establishing the amortized time complexity for other operations, we state without proof two simple observations. First, we define a *walk* in a region tree as a sequence of either rotation between adjacent nodes in a region or, if the walk crosses several regions, the crossing of bridges between the regions (where each bridge consists of a b–gate/f–gate pair of nodes). We also define the *depth* of a region in the region tree as the number of bridges between itself and the root region. For example, in the final configuration in figure 2, the region including A, B, C, and D has depth 0, while the region including E and F has depth 1 and the region including G has depth 2. We then have the following two observations.

Proposition 4.2

1. If a walk leaves a region s via a b–gate/f–gate g, then the same walk will enter region s only via gate g during its first return to s (if the walk does return to s).
2. During a walk, if the current position is in a region of depth k, then the walk cannot lead to another different region also of depth k without first going to a region of depth $k-1$.

□

We can draw the course of a walk on a plane with the time step as the X–axis and the depth of the current visited region as the Y–axis, as in figure 3. According to the above proposition, then, in the specific walk illustrated in figure 3, regions S, T, and W are the same region, but they may differ from region U. Furthermore, the f–gates s and

Figure 3: A walk in a region tree.

t are the same node because region V has only one b–gate, v, and v is bridged to both s and t.

Let us give an example. Suppose that the following *VRead* operation,

$$VRead\ [G, E, G, F, G, F, F, A, B, C, D]\ [i_0, i_1, i_2, i_3, i_4, i_5, i_6, i_7, i_8, i_9, i_{10}],$$

where $0 \leq i_k \leq 1$ for each index i_k, is issued to the final configuration in figure 2. The length of path formed by the above *VRead* is 13. (2 for G to E, 2 for E to G, 1 for G to F, 1 for F to G, 1 for G to F, 0 for F to F, 3 for F to A, 1 for A to B, 1 for B to C, and 1 for C to D.) Therefore, the *VRead* can be accomplished as a walk of 13 steps in the region tree. Each step of the walk first makes the current visited node the root node, then performs a *Read* operation if required. For example, the following 13–step walk accomplishes the above *VRead* operation.

> G (0 rotation, 1 read), F (0 rotation, 0 read), E (1 rotation, 1 read), F (1 rotation, 0 read), G (0 rotation, 1 read), F (0 rotation, 1 read), G (0 rotation, 1 read), F (0 rotation, 2 read), E (1 rotation, 0 read), B (0 rotation, 0 read), A (1 rotation, 1 read), B (1 rotation, 1 read), C (1 rotation, 1 read), D (1 rotation, 1 read).

Each of the above steps performs the operations in the parenthesis on the currently visited node. The rotations are used to make the current node the root node, and the reads fetch the required data. The course of the above walk is roughly pictured by figure 3.

However, the implementation of *VRead* as described in section 3 does not use a walk to accomplish a *VRead* operation. It does not move the root node of a region to the entrance gate whenever the next read operation enters the region from another region. Neither does it move the root node of a region to the exit gate whenever the next read operation leaves the region to another region. As a result, it will use less rotations than the walk sequence described above does. This is because the distance between two nodes A and B in a region is always less than the sum of the distance from A to an f–gate/b–gate g and the distance from g to B. This matters when the read operations to A and B are

interrupted by a read operation to a node outside the current region. For example, the implementation described in section 3 only uses the following operations to accomplish the same *VRead* operation.

> G (0 rotation, 1 read), E (1 rotation, 1 read), G (0 rotation, 1 read), F (1 rotation, 1 read), G (0 rotation, 1 read), F (0 rotation, 1 read), F (0 rotation, 1 read), A (1 rotation, 1 read), B (1 rotation, 1 read), C (1 rotation, 1 read), D (1 rotation, 1 read).

The concept of a walk is introduced to prove the following crucial lemma.

Lemma 4.3 A *VRead* operation of volume l costs $O(l)$ amortized time. □

PROOF. By the definition of *VRead*, the sequence of arrays being read forms a path of length $d = O(l)$ in the region tree. In the worst cases, the *VRead* operation must be accomplished by a walk of d steps in the region tree. For each step during the walk, two tasks are performed: a rotation between the current root node and the next node in the path (plus bridge–crossing if the two nodes reside in different regions), and possibly several reads in the new root node. In the following, we will count the read cost and the rotation cost in a *VRead* operation separately.

However, there is one complication: at the beginning of a *VRead*, a rerooting may be needed to make the first node in the walk the root node, and, if the walk crosses several regions, a rerooting may be needed every time the walk crosses a bridge between two regions (which is to make the walk's first node in the destined region the root node).

It is clear that a read to a root node costs one unit of time and the operation changes no potential of the data structure. Therefore, without counting the rotation cost, the total read cost for a *VRead* of volume l is l. It remains to count the cost for rotations.

Assume that the walk goes through, in order, regions $s_0, s_1, \ldots, s_{k-1}$. Also assume that d_i rotations are needed by the walk to cross region $s_j, 0 \leq j \leq k-1$. We have $\sum_{j=0}^{k-1} d_i = d$. For region $s_j, 1 \leq j \leq k-2$, there are four possible ways for a walk may cross the region. They are shown in figure 4. The amortized cost for each case will be analyzed. Let us assume that a region s_j has weight $W(s_j)$ and potential Φ_j before the crossing, and weight $W(s'_j)$ and potential Φ'_j afterward.

The actual cost for crossing a region s_j includes two parts: the cost for rerooting, which makes the entrance gate of the region the root node, and the cost for d_j rotations. Recall that we have shown in lemma 4.1 that if a rerooting involves cutting then its amortized cost is less than or equal to 0. Therefore, it suffices to analyze the situation where the rerooting does not involve any cutting.

The amortized costs for crossing each of the four cases in figure 4 are the following.

- Case 1. The actual cost t_j equals $W(s_j) + d_j$ because the crossing needs $W(s_j)$ time, exactly the weight of region s_j, to move the root node to the b–gate, and needs additional d_j time to rotate the root from the b–gate to the f–gate. We then have amortized cost

$$a_j = t_j + (\Phi'_j - \Phi_j) = (W(s_j) + d_j) + (W(s'_j) - W(s_j)) = d_j + W(s'_j) \leq 2d_j.$$

  Note that $W(s'_j) \leq d_j$ because the weight of region s_j is 0 when the crossing starts at its b–gate and each rotation only increases the weight of s_j at most by one.

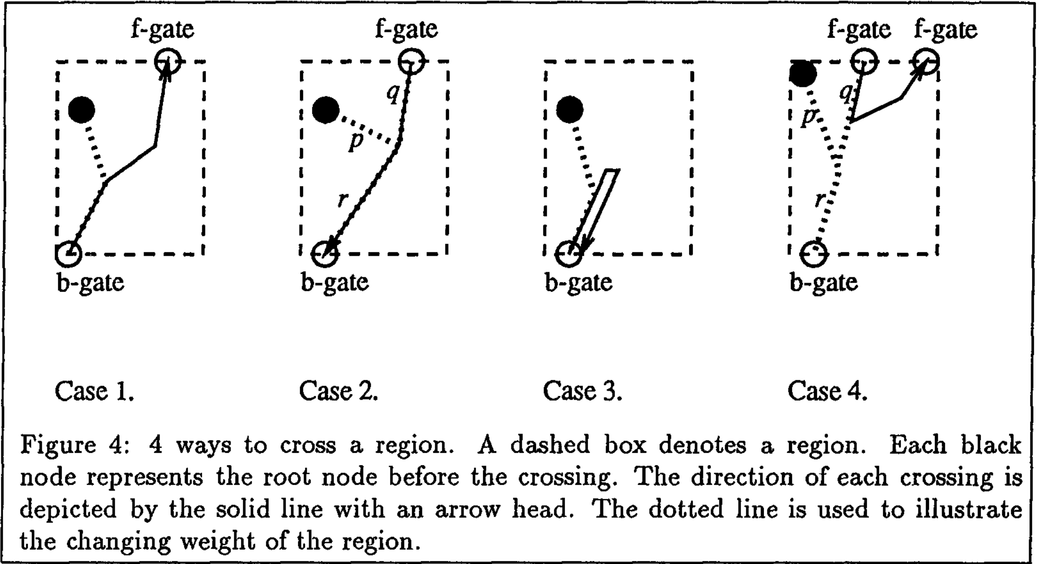

Figure 4: 4 ways to cross a region. A dashed box denotes a region. Each black node represents the root node before the crossing. The direction of each crossing is depicted by the solid line with an arrow head. The dotted line is used to illustrate the changing weight of the region.

- Case 2. Assume that, as pictured, the rerooting first reduces the weight of s_j from $p+r$ to r, then increases it to $q+r$. The actual cost for the rerooting is then $p+q$. For the rotations, the actual cost is d_j. It can be shown that $0 \leq p, 0 \leq q, 0 \leq r$, and $r+q \leq d_j$. Then, we have amortized cost

$$a_j = t_j + (\Phi'_j - \Phi_j) = (p+q+d_j) + (0-(p+r)) = d_j + (q-r) \leq 2d_j.$$

- Case 3. Similar to case 1, except that the weight of s_j after the crossing is 0. We then have

$$a_j = t_j + (\Phi'_j - \Phi_j) = (W(s_j) + d_j) + (0 - W(s_j)) = d_j.$$

- Case 4. Similar to case 2, except that the weight of s_j after the crossing may be as large as $q+r+d_j$. We then have

$$a_j = t_j + (\Phi'_j - \Phi_j) \leq (p+q+d_j) + ((q+r+d_j) - (p+r)) = 2d_j + 2q.$$

Except for case 4, the amortized cost for crossing region s_j is bounded by $2d_j$.

Let w denote the region which has the least depth among the crossed regions. By proposition 4.2, such a region is unique. Suppose that a region s of depth greater than w is crossed as in case 4. Let s be s_j for some $1 \leq j \leq k-2$. Then either region s was crossed as $s_{j'}, j' < j$, by case 1 in order to exit from region w, or s will be crossed as $s_{j'}, j' > j$, by case 2 in order to enter region w. Furthermore, by proposition 4.2, the two crossings s_j and $s_{j'}$ will use a same f–gate as the exit/entrance gate. Therefore, no rerooting is necessary between the transition from $s_{j'}$ to s_j (or from s_j to $s_{j'}$).

We can then combine s_j and $s_{j'}$ as they are a single crossing in region s. The combined crossing will be of case 1 or 2 depending of what case $s_{j'}$ is. The combined amortized cost is then less than $2(d_j + d_{j'})$. It is possible that between s_j and $s_{j'}$, there are other

case 4 crossings at the same region. If so, we can combine those case 4 crossings and make them as they are a single case 4 crossing. For example, in figure 3, the crossings at regions S, T, and W can be combined as a single case 2 crossing. (Note that regions S, T, and W are the same region.)

In the above, we assume that region s is of depth greater than w, the deepest region ever crossed. Suppose that $s = w$ is crossed as in case 4. Then the above analysis still holds except that all the crossing to w may be of case 4. In such a case, we can combine them into a single case 4 crossing. The combined amortized cost is $2\sum_{j\in J} d_j + 2q$, for some subset $J \subseteq \{1..(k-2)\}$. Furthermore, since the rerooting does not involve any cutting, we have $q \leq 2n$

Summarizing the total amortized cost for $s_j, 1 \leq j \leq k-2$, we have

$$\sum_{j=1}^{k-2} a_j = 2q + 2\sum_{j=1}^{k-2} d_j \leq 4n + 2\sum_{j=1}^{k-2} d_j.$$

It is not difficult to see that s_0 and s_{k-1} each costs at most $4n + 2d_0$ and $4n + 2d_{k-1}$ amortized time, respectively. Therefore, we have the total amortized time

$$\sum_{j=0}^{k-1} a_j = (a_0 + a_{k-1}) + \sum_{j=1}^{k-2} a_j \leq 12n + 2d.$$

Adding both the cost for read (which is l) and rotation (which is at most $12n + 2d$), the total amortized time for a *VRead* of volume l is at most $l + 12n + 2d$, which is $O(l)$ because l is $\Omega(n)$ and d is $O(l)$. ◇

We then can show the following main result.

Theorem 4.4 Let a *VRead* operation of volume l be counted as l individual operations, a *Create* n operation be counted as n individual operations, and an *Update* operation be counted as 1 individual operation. Then a sequence of *Create*, *Update*, and *VRead* operations can be implemented in $O(m)$ time and $O(n+u)$ space by the fragmented shallow binding scheme if the sequence has m individual operations, starts from a single *Create* n operation, and has u *Update* operations. □

PROOF. Let a_i and t_i denote respectively the amortized and actual time for the ith *individual* operation in the sequence, and Φ_{i-1} and Φ_i respectively be the potential of the data structure before and after the ith individual operation. Also recall that the potential of a configuration D of the data structure is defined as

$$\Phi(D) = 3(NR(D) - n \cdot R(D)) + \sum_{S\in D} W(S).$$

The first operation in the sequence is a *Create* n operation. It needs $n+1$ units of actual time, and space, to allocate and initialize both a cache of size n and a root node. Therefore, the total amortized time for the first n individual operations is described by

$$\sum_{i=1}^{n} a_i = \sum_{i=1}^{n} t_i + (\Phi_n - \Phi_0) = (n+1) + 3(-n) = -2n + 1.$$

Suppose that the ith individual operation, $n < i \leq m$, is an *Update* operation. It will need 1 unit of actual time and 1 unit of space to complete the operation. The amortized time a_i is then

$$a_i = t_i + (\Phi_i - \Phi_{i-1}) = 1 + 3 = 4.$$

Suppose that the ith to $(i+l-1)$th individual operations constitute a *VRead* operation of volume l. By lemma 4.3, the total amortized time for the *VRead* is $O(l)$. That is, it is less than $C_i \cdot l$ for some constant $C_i \geq 0$.

The total amortized cost for the m individual operations is then

$$\sum_{i=1}^{m} a_i = \sum_{i=1}^{n} a_i + \sum_{i=n+1}^{m} a_i \leq (-2n+1) + C(m-n),$$

for some constant $C = max\{4, C_{i_1}, C_{i_2}, \ldots, C_{i_j}\}$. Constants $C_{i_1}, C_{i_2}, \ldots, C_{i_j}$ are derived from the j *VRead* operations in the whole sequence.

If the final configuration of the data structure has $k > 1$ regions, then it can be shown that each region has at least $\lfloor \frac{2n+1}{3} \rfloor$ nodes. Therefore, we have its potential as

$$\Phi_m \geq 3(k \left\lfloor \frac{2n+1}{3} \right\rfloor - n \cdot k) + 0 \geq 0,$$

when the array size $n \geq 1$. If the final configuration has only one region, then its potential is

$$\Phi_m \geq 3(0 - n \cdot 1) + 0 = -3n.$$

Therefore, the total actual time for the entire m individual operations is

$$\sum_{i=1}^{m} t_i = \sum_{i=1}^{m} a_i - (\Phi_m - \Phi_0) \leq (-2n + 1 + C(m-n)) + 3n.$$

The total actual time is $O(m)$ because $m \geq n$. It is clear that the data structure use only $O(n+u)$ space, where u is the total number of *Update* operations in the sequence. ◇

Corollary 4.5 A sequence of m *Read* and *Update* operations, which starts from a single *Create* n operation and includes u *Update* operations, can be implemented in $O(n+m)$ time and $O(n+u)$ space by the fragmented shallow binding scheme if all the *Read* operations in the sequence can be partitioned into disjoint voluminous subsequences. It does not matter if *Update* operations are mixed in the voluminous sequences of *Read*. □

PROOF. Since an *Update* operation does not change the weight of the region to which it is applied, *Update* will not change the amortized cost analysis for *VRead* in the proof of Lemma 4.3. Except that the amortized cost of the *Update* operation is now incorporated in the summarization of the amortized cost for the voluminous *Read* operations. But this does not change the total amortized cost of the entire m operations either. The proof then follows immediately from theorem 4.4. ◇

Note that disjoint voluminous sequences of reads arise naturally in many functional array applications due to the dense locality of their accesses to arrays. Compile–time partition of *Read* operations into voluminous sequences may not be necessary in most

cases to achieve good running time. Also the fragmented data structure resulted from the fragmented shallow binding scheme is likely to make voluminous sequences of reads disjoint. The empirical results in section 5 will show that the above observations are valid.

5 Examples and Empirical Results

Is it practical to use the fragmented shallow binding scheme to implement functional arrays? Before answering this question, we have to argue first that functional arrays are often used in an incremental updates/voluminous reads style. In single–threaded accesses to functional arrays, the entire read sequence is voluminous because each read is applied to the newly updated array and the total number of rotations needed in all the read operations is bounded by the the total number of updates being performed. For multi–threaded applications of functional arrays, we observe that many of them will read every entry of an array if there is ever a need to read the array. If a long sequence of reads is not directed to a single array, it is usually directed to a sequence of closely related arrays. Both of these multi–threaded read patterns are voluminous.

We can take the full histogram problem as an example in the multi–threaded case. The full histogram problem is to classify a sequence of incoming events into a fixed set of categories, and to query either the distribution of events among the categories in a certain past time step, or to query the evolution of the events of a particular category in a certain sequence of past time steps. The queries can occur while events are still coming in. It is common to represent the fixed set of categories as an array and have each entry of the array store the number of events which have happened so far in the given category. To answer the query of distribution of events, all entries of the array have to be read. Also, the evolution of events of a category can be accessed by straight–line sequence of reads to the arrays in the version tree. Both the distribution query and the evolution query are implemented by voluminous reads.

We give empirical results collected from the execution of four sample programs to support our argument. The programs are for the multiplication of two matrices (each of size 100×100), all the safe positions for 8 queens, the transitive closures of a graph of 100 nodes, and the simulation of a histogram involving 10,000 events in 100 categories. The Warshall [27] algorithm is used in the transitive closures program, and all intermediate closures (which are those considering only paths through node 0 to node $i, 0 \leq i \leq 99$) are preserved.

Except for the matrix multiplication program, all programs are multi–threaded. After the results are computed, each program also reads every entry of the resulting arrays. We do this because otherwise the fragmented shallow binding scheme will have a clear advantage over other implementation schemes. This is because much of the work in the fragmented shallow binding scheme only occurs when reads are performed to the resulted arrays. Also, no effort is spent in partitioning the *Read* operations into voluminous sequences in the fragmented shallow binding implementation; the ordering of *Read* operations is left as it is (specified by the program).

Table 1 shows the total number of *Create*, *Update*, and *Read* operations being performed; the total number of root nodes and non–root nodes in the resulted data structure;

	create	update	read	root node (cache size)	non–root node	rerooting	rotation
matrix	202	40200	2070102	202 (100)	40200	20202	40200
queens	1	2056	41018	1 (8)	2056	2056	5020
closures	1	9803	5328354	49 (100)	9803	14851	27415
histogram	2	20101	37625	60 (100)	20001	25070	37713

Table 1: Operation counts of the four programs.

	destructive	fragmented shallow	shallow	copying
matrix	43.08 (1.47)	50.38 (5.91)	50.32 (5.09)	84.73 (10.34)
queens	not applicable	1.12 (0.07)	1.10 (0.05)	1.10 (0.05)
closures	not applicable	58.99 (2.19)	59.07 (2.22)	66.00 (5.54)
histogram	not applicable	4.23 (1.95)	19.26 (12.06)	1455.45 (603.03)

Table 2: Execution times of the four programs.

and the total number of rerootings and rotations being performed. We can see that

$$\text{(the number of non–root nodes)} + \text{(the cache size)} \cdot \text{(the number of root nodes)},$$

which is the total space used, is about the magnitude of the number of updates. The number of rotations, which is the total time spent in rerooting nodes, is about the magnitude of the number of read operations too. Also note that the total number of reads being performed is far greater than the total number of updates being performed in each of the four programs. These statistics are very likely the result of incremental updates/voluminous reads execution patterns, which make the fragmented shallow binding scheme very much applicable. Figure 5 and figure 6 show the traces of the four programs' read sequences. The traces show that most of the entire read sequence in each of the four programs is voluminous.

Table 2 shows the execution times of the four programs. The programs are implemented in Standard ML of New Jersey, version 0.66, and run on a Sparc workstation (Sun 4/290, with 32 MB of physical memory) Each entry in the table is of the format total time (garbage collection time), as provided by the `System.Timer` module of SML of New Jersey. Each datum in the table is the average of the data from three runs.

Four sets of timing are shown. In the cases where array updating is done destructively all the time, where the fragmented shallow binding scheme is used, where the shallow binding scheme is used, and where the whole array is copied when being updated. The same program is used in all four sets of experiments; it just calls different implementation of the array module in each case. Also note that the destructive updating implementation of functional arrays is not applicable to the 8–queens, transitive closures, and histogram programs because those programs are multi–threaded.

According to the results in table 2, the fragmented shallow binding scheme performs well. It is as fast as the shallow binding scheme when arrays are mostly used in single–threaded ways; and is far better in case when a program uses arrays mostly in a multi–threaded way, as demonstrated in the histogram example. But it is clear from the 8–queens example that for the fragmented shallow binding scheme to be effective, the array size had better be large.

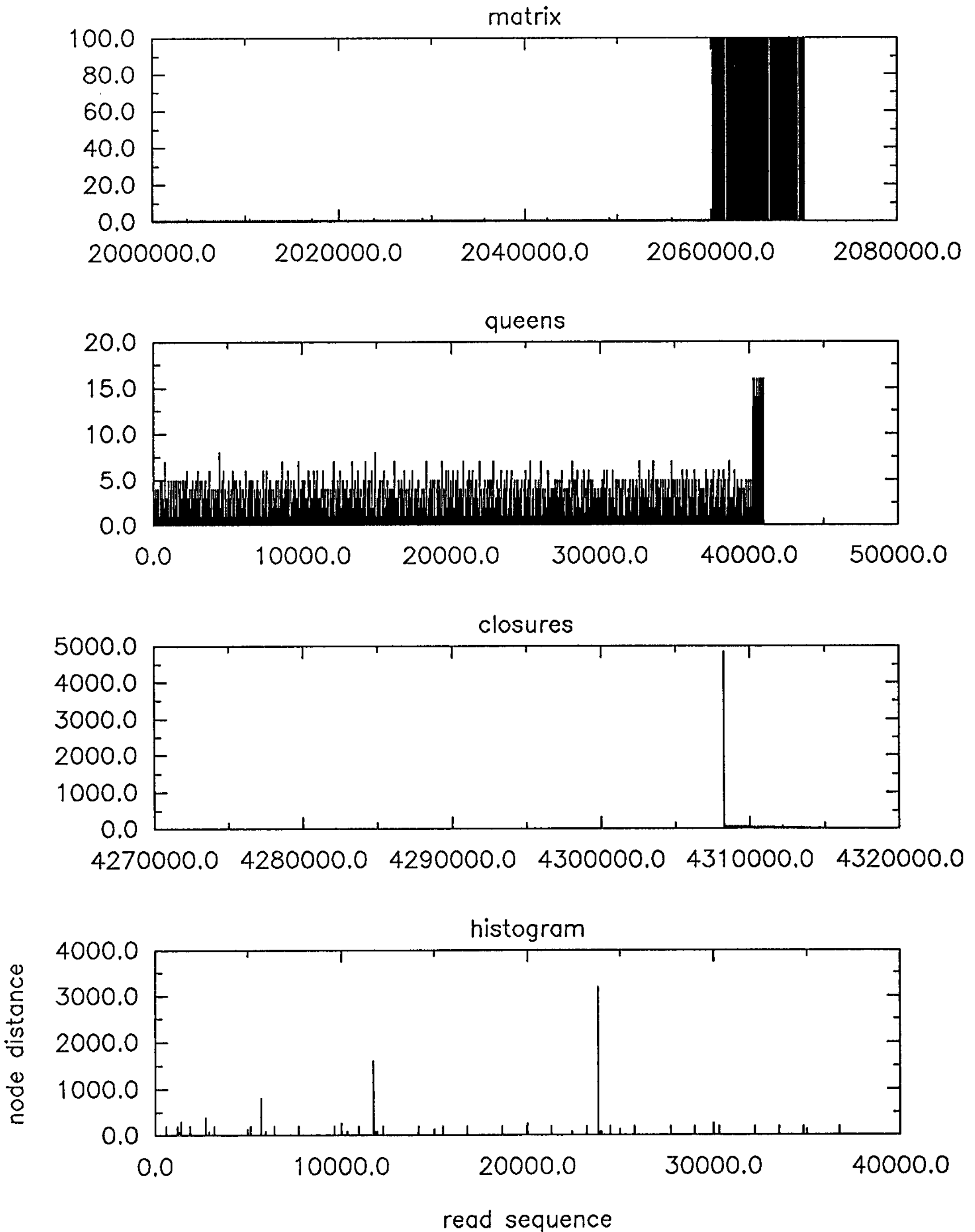

Figure 5: Trace graphs for the read sequences of the four programs. The X–axis is for the read sequence, and the Y–axis is for the distance between the current root node and the node to be read. Due to the large number of reads in the matrix and the histogram programs, only parts of their traces are shown. Note that each graph has its own scale.

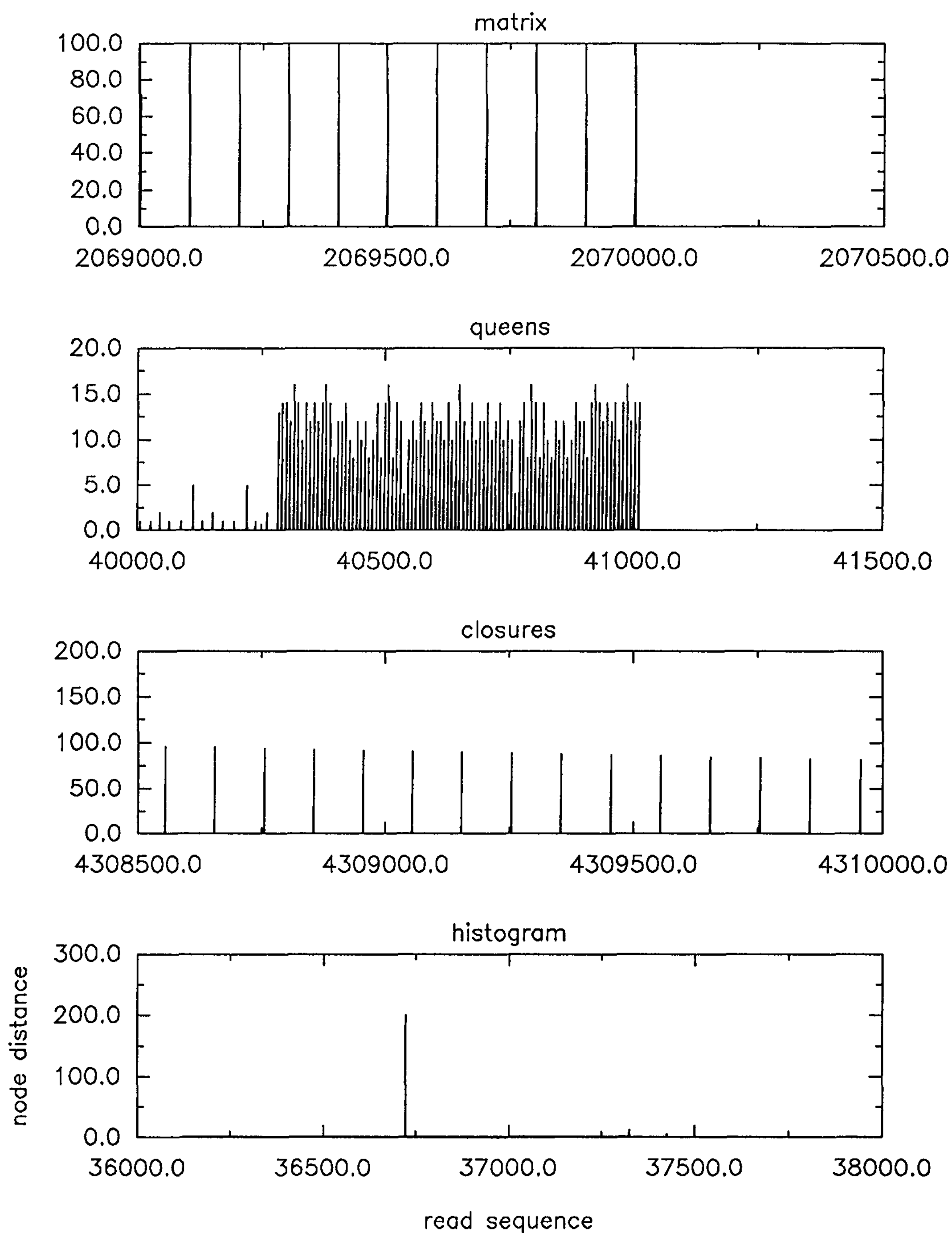

Figure 6: Magnified versions of the trace graphs. Only some interesting portions are shown.

6 Future Work

There remains several performance issues to be addressed. For example, does the fragmented shallow binding scheme perform well in a lazy functional language? How about in a parallel system where the voluminous read sequences are not coordinated and may intervene with one another?

We believe the fragmented shallow binding scheme will perform well in a lazy evaluation setting because the an update operation is in fact carried out lazily. In our scheme, the result of an *Update* operation is computed only if there is a need to read the resulting array afterward. That is, the rerooting operation, which makes the desired version of an array current, is performed (if necessary) only during a *Read* operation, not during an *Update* operation. On the other hand, however, it remains to be seen if read operations will typically lead to voluminous sequences in a lazy evaluation setting.

Regarding uncoordinated voluminous *Read* sequences in a parallel system, it has been observed that performance will suffer if the sequences intervene with one another. This is because the cost spent in rerooting may not be paid off by voluminous reads. In the worst case, each read in a voluminous sequence may have to do a long rerooting. But we view this as a common problem faced by sharing aggregate data structures in a parallel system, not a problem caused by the fragmented shallow binding scheme *per se.* On the other hand, the fragmented shallow binding scheme also has its advantage because its data structure is more likely to be fragmented and to be distributed well.

7 Acknowledgment

I am very grateful to Malcolm Harrison for introducing the idea of shallow binding to me, to Atul Sibal for suggesting that having multiple caches is better than a single caches, and to Chia-Hsiang Chang for helping me with the proof of lemma 4.3. I am also very grateful to Benjamin Goldberg, my advisor, for his encourage and support, and for the time he spent with me to improve the presentation of this paper, to Henry G. Baker for making several detailed comments on a draft of this paper and sending me related literature, to Annika Aasa for providing literature, and to the referees for their helpful comments.

References

[1] *12th Annual ACM Symposium on Principles of Programming Languages.* A.C.M., January 1985. New Orleans, Louisiana, U.S.A.

[2] *Functional Programming Languages and Computer Architecture.* A.C.M./Addison-Wesley, September 1989. Imperial College, London, U.K.

[3] *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming.* A.C.M., June 1990. Nice, France.

[4] *18th Annual ACM Symposium on Principles of Programming Languages.* A.C.M., January 1991. Orlando, Florida, U.S.A.

[5] *Proceedings of the Symposium on Partial Evaluation and and Semantics-Based Program Manipulation.* A.C.M., June 1991. New Haven, Connecticut, U.S.A. Also appears as *SIGPLAN Notices*, 26(9), September 1991.

[6] Annika Aasa, Sören Holmström, and Christina Nilsson. An efficiency comparison of some representations of purely functional arrays. *BIT*, 28(3):490–503, 1988.

[7] Henry G. Baker, Jr. Shallow binding in Lisp 1.5. *Communications of the ACM*, 21(7):565–569, July 1978.

[8] Henry G. Baker. Shallow binding makes functional arrays fast. *SIGPLAN Notices*, 26(8):145–147, August 1991.

[9] Adrienne Gael Bloss. *Path Analysis and the Optimization of Non-Strict Functional Languages*. PhD thesis, Department of Computer Science, Yale University, May 1989. Also appears as report YALEU/DCS/RR–704.

[10] Adrienne Bloss. Update analysis and the efficient implementation of functional aggregates. pages 26–38. In [2].

[11] Frank Dehne, Jörg-Rüdiger Sack, and Nicola Santoro, editors. *Algorithms and Data Structures*. Ottawa, Canada, August 1989. Lecture Notes in Computer Science, Volume 382, Springer–Verlag.

[12] Paul F. Dietz. Fully persistent arrays. pages 67–74. In [11].

[13] James R. Driscoll, Neil Sarnak, Daniel D. Sleator, and Robert E. Tarjan. Making data structures persistent. *Journal of Computer and System Sciences*, 38(1):86–124, February 1989.

[14] Juan C. Guzmán and Paul Hudak. Single-threaded polymorphic lambda calculus. In *Proceedings of 5th Annual IEEE Symposium on Logic in Computer Science*, pages 333–343. I.E.E.E., June 1990.

[15] Sören Holmström. How to handle large data structures in functional languages. In *Proceedings of the SERC Chalmers Workshop on Declarative Programming Languages*. University College London, 1983.

[16] Paul Hudak and Adrienne Bloss. The aggregate update problem in functional programming systems. pages 300–314. In [1].

[17] Paul Hudak, Simon Peyton Jones, and Philip Wadler, editors. *Report on the Programming Language Haskell — A Non-Strict, Purely Functional Language, Version 1.1*. August 1991. Available from Yale University and University of Glasgow.

[18] John Hughes. An efficient implementation of purely functional arrays. Technical report, Department of Computer Sciences, Chalmers University of Technology, 1985.

[19] Martin Odersky. How to make destructive updates less destructive. pages 25–36. In [4].

[20] Paul Walton Purdom, Jr. and Cynthia A. Brown. *The Analysis of Algorithms*. Holt, Rinehart and Winston, 1985.

[21] David A. Schmidt. Detecting global variables in denotational specifications. *ACM Transactions on Programming Languages and Systems*, 7(2):299–310, April 1985.

[22] J. T. Schwartz. Optimization of very high level languages — i. value transmission and its corollaries. *Computer Languages*, 1(2):161–194, June 1975.

[23] J. T. Schwartz. Optimization of very high level languages — ii. deducing relationships of inclusion and membership. *Computer Languages*, 1(3):197–218, September 1975.

[24] Robert Endre Tarjan. Amortized computational complexity. *SIAM Journal on Algebraic and Discrete Methods*, 6(2):306–318, April 1985.

[25] Philip Wadler. Comprehending monads. pages 61–78. In [3].

[26] Philip Wadler. Is there a use for linear logic? pages 255–273. In [5].

[27] Stephen Warshall. A theorem on boolean matrices. *Journal of the Association for Computing Machinery*, 9(1):11–12, January 1962.