

# A Randomized Implementation of Multiple Functional Arrays

Tyng-Ruey Chuang

Department of Computing Science, Chalmers University of Technology, S-412 96 Göteborg, Sweden;  
Institute of Information Science, Academia Sinica, Nankang, Taipei 11529, Taiwan\*

## Abstract

The array update problem in the implementation of 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. Previous approaches have mainly based on the detection or enforcement of single-threaded accesses to an aggregate, by means of compiler-time analyses or language restrictions. These approaches cannot deal with aggregates which are updated in a multi-threaded manner.

Baker's shallow binding scheme can be used to implement multi-threaded functional arrays. His scheme, however, can be very expensive if there are repeated alternations between long binding paths. We design a scheme that fragments binding paths randomly. The randomization scheme is on-line, simple to implement, and its expected performance comparable to that of the optimal off-line solutions. All this is achieved without using compiler-time analyses, and without restricting the languages.

The expected performance of the randomization scheme is analyzed in details, and some preliminary results are shown. Applications of the randomization technique to other functional aggregates, as well as its implications, are briefly outlined.

## 1 A Brief Survey of the Aggregate Update Problem

In a functional program, if an aggregate data structure is updated then both the original version and the updated version of the aggregate must be preserved, preferably at a small cost, to maintain referential transparency. It is generally regarded as too expensive to make a complete copy of the aggregate that differs from the original only in the updated position. There have been various approaches to solve the aggregate update problem.

If a compile-time analysis or a run-time test can determine that the original version of an aggregate will not be referenced following an update, then the update can be performed in place [7, 12, 16, 20]. However, such an opti-

mization analysis or run-time testing can be quite expensive to implement.

Language primitives, in the forms of type disciplines or built-in functions, can also be provided to write functional programs such that single-threadedness can easily be recognized and implemented by a compiler. A sequence of read/update accesses to an aggregate is called single-threaded if each operation only refers to the newest version of the aggregate [21]. Thus, all update operations in a single-threaded sequence can be performed in place because previous versions of the aggregate will never be needed and can be safely overwritten [2, 17, 23]. However, such language constraints can be too restrictive in that the resulting functional programs may be too imperative-like and impose unnecessary evaluation order.

A common drawback of the above two approaches — that single-threadedness is detected by compiler-time analyses or is enforced by language constraints — is that they cannot deal with aggregates which are updated in a multi-threaded manner. This is because the two approaches aim to keep only one copy of each aggregate. As an example, the above two approaches cannot properly handle the following expression:

$$(\text{Read}(\text{Update } A \ i \ u) \ j) + (\text{Read}(\text{Update } A \ m \ v) \ n)$$

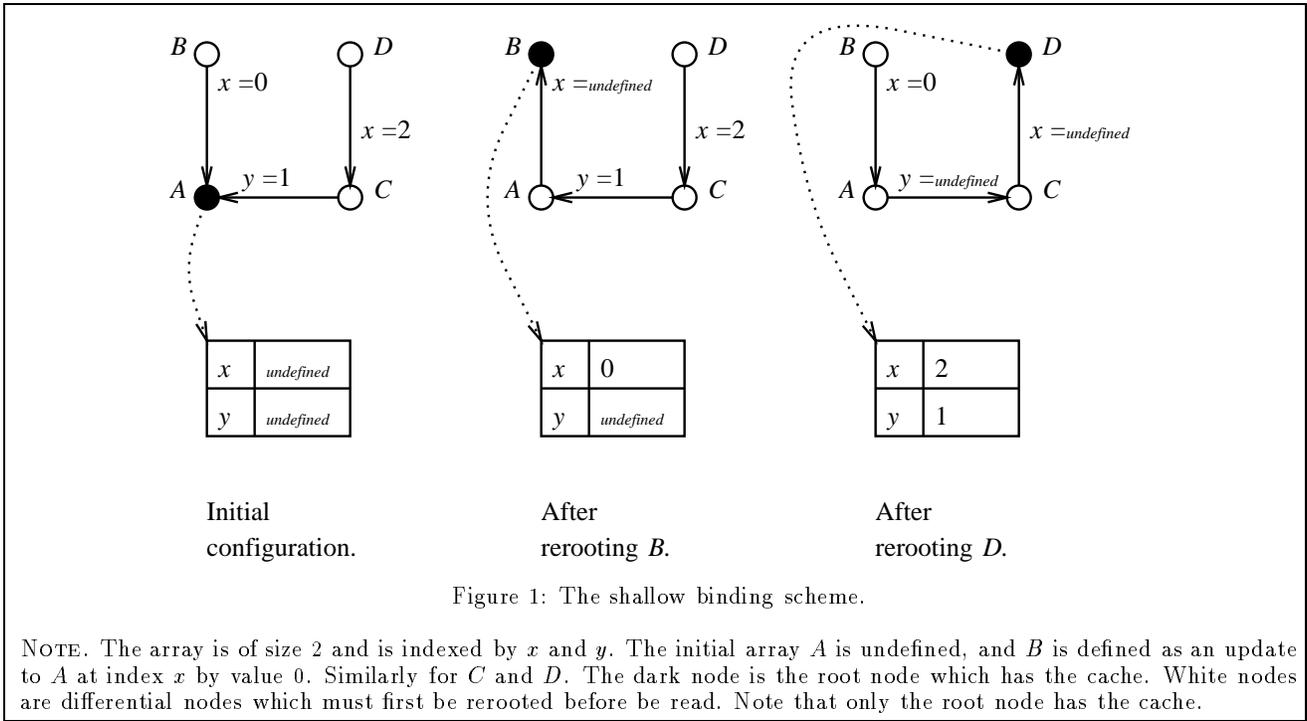
where  $A$  is an array,  $i, j, m, n$  indexes, and  $u, v$  values to be stored in the array. In image processing, the above multi-threaded update sequences can occur in the following way: various transformations are applied to an original image, then the transformed images are compared to one another for their effects. We can construct a monolithic array to represent the image after each transformation, but this performs badly if each transformation only make few changes to the original image.

The above drawback leads one to another approach, which is to design efficient algorithms to make aggregates fully persistent (*i.e.*, purely functional) such that, after a sequence of updates, the newest version as well as all previous versions of the aggregate are still accessible [1, 5, 6, 8, 9, 10, 11, 15, 19, 22]. The challenge is to reduce as much as possible the associated overhead when maintaining multiple versions of an aggregate.

In this paper, we develop a randomization technique for the efficient implementation of multiple functional arrays. The expected performance of the randomized implementation will be analyzed in details. We will also briefly describe its applications to other functional aggregates, and its implications to functional programming in general. The paper is

\*The author's affiliation since January 1, 1994. E-mail: trc@iis.sinica.edu.tw.

Appeared in *Proceedings of the 1994 ACM Conference on Lisp and Functional Programming*, pages 173–184. Orlando, Florida, USA, June 1994. ACM Press. The proceedings also appears as *Lisp Pointers*, Volume VII, Number 3, July–September 1994.



organized as the following: Section 2 describes an implementation of functional arrays based on the fragmented shallow binding scheme, and its problems. A randomization scheme is proposed in Section 3, and its expected performance analyzed in Section 4. Section 5 describes a preliminary implementation based on the Chalmers Lazy ML compiler and explains the results. Section 6 states its beneficial effect on garbage collectors, and its applications to other functional aggregates. Section 7 concludes this paper.

## 2 Functional Arrays by Fragmented Shallow Binding

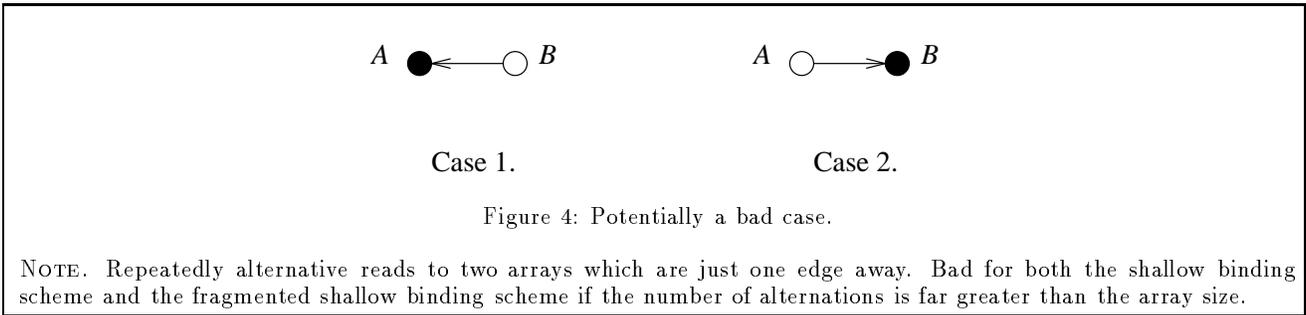
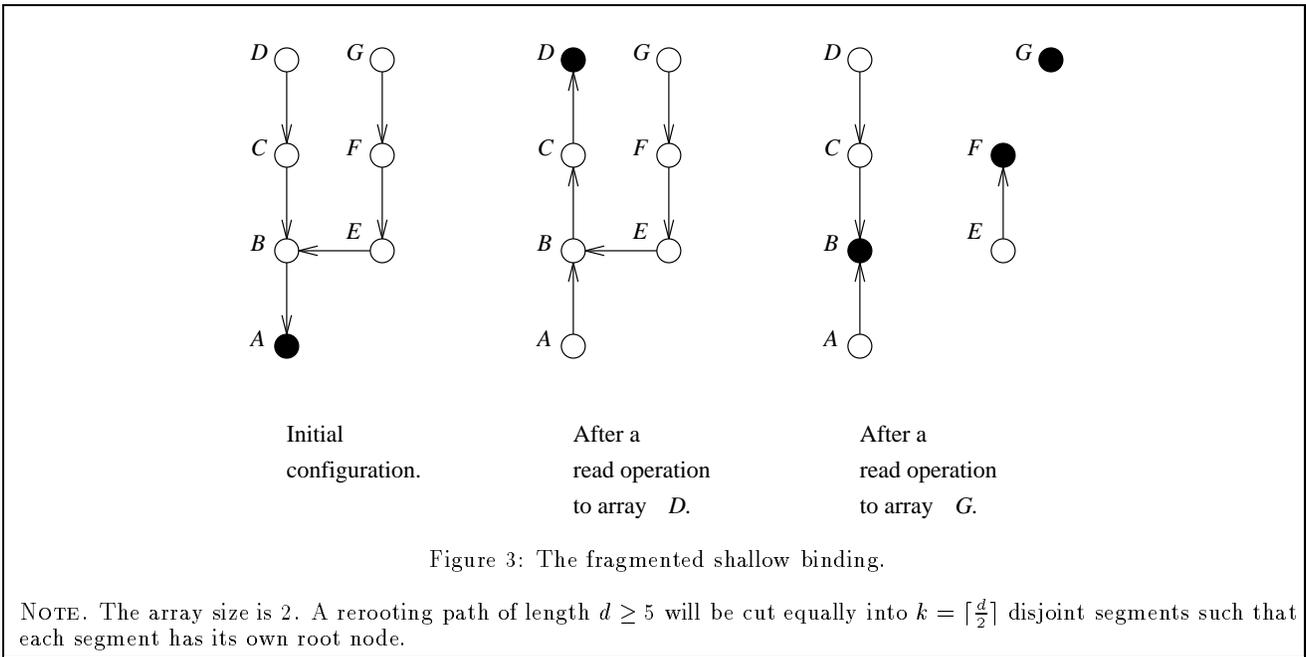
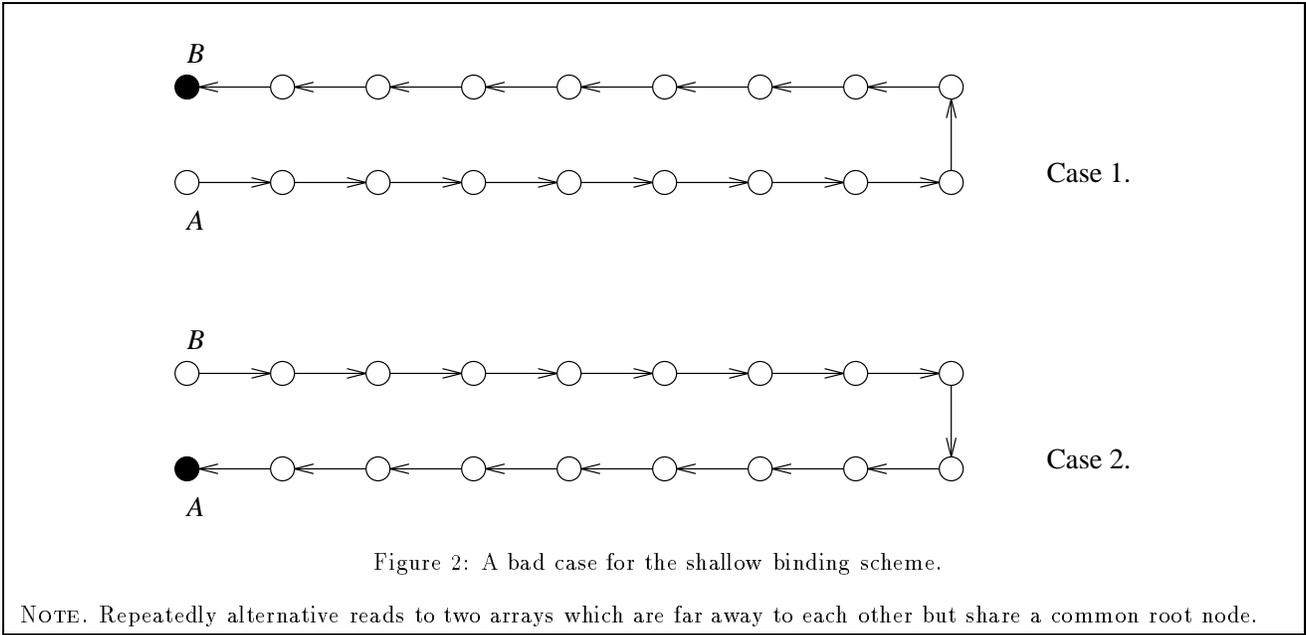
General techniques for making aggregate data structures fully persistent are described by Driscoll, Sarnak, Sleator, and Tarjan [10]. However, these techniques only apply to linked data structures. They cannot be applied to the usual representation of an array, which is not implemented as a linked data structure but as a successive block of locations in the random access memory (RAM). In this paper, we will use the term *cache* to call this block of RAM. This allows us to distinguish this ephemeral implementation of an array from the fully persistent implementations of an array.

Note that an array can be implemented as a balanced search tree such that the fully persistent techniques of Driscoll, Sarnak, Sleator, and Tarjan apply. But this is unattractive because we lose constant time accessibility to an array even if the array is used only in a single-threaded matter. We aim to retain the constant time accessibility of an array as often as possible.

Baker's *shallow binding scheme*, which was originally designed to support multiple environments for higher-order functional languages [5], can be used to implement functional arrays [6]. In this scheme, also known as the *reversible differential list* method in the folklore, only one

version of the arrays keep the physical copy of the cache. This array is the *root* in the version tree. Each of other versions is represented as a path of *differential nodes*, where each node describes the difference between the current array and the previous array. The difference is represented as an  $(\text{index}, \text{value})$  pair, which describes the new value to be stored at the specified index. All paths lead to the root. An update to an array is simply implemented by attaching a differential node to the node it is updating. A read to the root node just fetches the corresponding entry in the cache. A read to a differential node is accomplished by a sequence of *rotations* which exchange the  $(\text{index}, \text{value})$  information along the path leading to the root. Such a sequence of rotations is called *rerooting*. Notice that 1) a rotation can only occur between a root node and a differential node pointing to it, 2) the differential node becomes the new root after the rotation, and 3) the original root node becomes a differential node pointing to the new root. In this scheme, each update costs constant time while the cost for a read is first proportional to its rerooting length. However, after the first read, each (immediately) following read to the same array costs only constant time. Figure 1 illustrates the shallow binding scheme.

Shallow binding scheme performs badly if read operations are issued alternatively to arrays which are far away to one another in the version tree. Then the rerooting cost, which can be linear to the total number of updates so far, dominates each read. See Figure 2 for such a case. A *fragmented shallow binding scheme* has been proposed to solve this problem [8]. In this new scheme, whenever a read operation discovers it is rerooting a path of length  $d \geq 2n + 1$ , where  $n$  is the array size, the read cuts the path between the new root and old root evenly into  $k = \lceil \frac{d}{n} \rceil$  disjoint segments such that each segment has at least  $\lfloor \frac{d}{k} \rfloor \approx n$  nodes



(including a root node with a cache of size  $n$ ). See Figure 3 for illustration. The cost of cutting long rerooting paths will be balanced by that of shorter rerooting in the future. It is shown that, at the cost of doubling the space usage, this scheme performs well under some mild locality assumption [8].

However, cutting long paths when rerooting does not quite solve the whole problem. There are short paths which are equally annoying. Notice that there is a threshold  $d$  in the fragmented shallow binding scheme such that rerooting paths of length less than  $d$  are not cut. Also notice that we cannot arbitrarily reduce this threshold. This can increase space usage — because each cut will demand a new cache of size  $n$  — without any efficiency gain. Figure 4 illustrates two arrays  $A$  and  $B$  which are different to each other in one index, but may cause problem in the long run. Suppose  $A$  and  $B$  are read alternatively for  $m$  times. Further, suppose that it costs one unit of time for each rotation, and  $n$  units of time to copy the cache. Then the fragmented shallow binding, as well as the shallow binding scheme, will require  $m$  units of overhead for the rotations. This becomes un economical if  $m > n + 1$ . In this case, the optimal solution will opt to copy the cache (and make the necessary modification required by the differential node) at the first very read that requires a rotation. This makes both  $A$  and  $B$  possess their own cache. There is no additional overhead for the remaining reads. This only incurs  $n + 1$  overhead, which can be arbitrarily smaller than  $m$ . On the other hand, if  $m \leq n + 1$  then the optimal solution will not duplicate the cache. It just uses rotations all the way.

The above optimal solution requires us to foretell the future. (Will the number of alternations in the entire read sequence larger than the array size?) It requires an off-line algorithm, where the entire access sequence is given in advance in order to derive the optimal solution. We will show in the next section a randomized version of the fragmented shallow binding scheme. It is an on-line scheme, simple to implement, and its expected performance comparable to that of the optimal off-line solution.

### 3 A Randomization Scheme

The idea is very simple. Whenever there is a rotation between a root node and a differential node, we copy the cache owned by the root node with probability  $\frac{1}{n}$ , and give it to the differential node. To be precise, the following three array 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  $A_j(i)$ .

are implemented in the randomization scheme by the following:

- *Create  $n$* :  
Allocate a cache of size  $n$ . Allocate a root node of one field and have this field point to the cache. Return the address of the root node.
- *Update  $A_j$   $i$   $v$* :  
Allocate a differential node of three fields and have

it store  $A_j$ ,  $i$ , and  $v$ . (Notice that  $A_j$  is an address.) Return the address of this newly allocated node.

- *Read  $A_j$   $i$* :  
Do a rerooting starting from the node pointed to by  $A_j$ . However, for each rotation between a root node  $u$  and a differential node  $v$  during the rerooting, the cache owned by  $u$  is copied with probability  $p = \frac{1}{n}$ . The duplicated cache is then modified by the (index, value) pair dictated by  $v$ . Make  $v$  a root node which own this modified cache. (Notice that the original cache is still solely owned by  $u$ .)

For the remaining probability  $q = 1 - p$ , the rotation between  $u$  and  $v$  is carried out by just a plain rotation, which makes  $v$  a root node and  $u$  a differential node pointing to  $v$ .

After the rerooting,  $A_j$  points to a root node which owns a cache. Return the  $i^{\text{th}}$  entry in the cache.

Note that each array creation costs  $O(n)$  time, and each update operation costs constant time. The expected cost of a read operation is more difficult to see, and the main purpose of the next section is to analyze it. Note also that we have two kinds of rotation now: one that cuts an edge and the other that does not. The term rotation will still be used to address both kinds of rotations. If necessary, the term *plain rotation* is used to distinguish the rotations with no cutting from the rotations that cut.

Let us illustrate the effect of the above randomization scheme by Figure 3. Recall that the array size is 2. We cut the edge during each rotation if the coin tossing results in head. The edge is retained if the coin comes out in tail. The chance of a head is  $\frac{1}{2}$ , so is a tail. The last configuration (“after a read ...  $G$ ”) can be viewed as resulting from the previous one (“after a read ...  $D$ ”) by the following coin tossing: tail, tail, head, tail, head.

What are the justifications behind such a randomization scheme? Recall that the motivation beyond the fragmented shallow binding scheme is to cut a long rerooting path into segments of length about  $n$ . This reduces the cost of subsequent reads which may otherwise require the same long rerooting. In the randomization scheme, we can expect a long rerooting path to be cut once in every  $\frac{1}{p} = n$  rotations. This reduces a long rerooting path into shorter ones. Furthermore, for a short rerooting path which is being repeatedly rerooted, the randomization scheme is also expected to cut the path for every  $n$  rotations. This solves the problem described by Figure 4. On the other hand, the additional overhead for each rotation (which is incurred by the randomization scheme to copy a cache of size  $n$ ) is expected to cost only  $p \cdot n = 1$  unit of time and space.

Detailed analyses on the expected performance of the randomization scheme will be described in the next section. But let us make some remarks before leaving this section. Note that the randomized method does not make any probabilistic assumption on the distribution of the read/update access sequences. Rather, the method relies on a private source of random bits to guide its actions — to copy or not to copy. The expected performance of the randomization scheme, therefore, shall be interpreted in the following way: a given access sequence  $s$  is expected to cost  $O(f(s))$  overhead if, when running the randomization scheme over the entire sequence repeatedly for sufficiently many times, the overhead averaged over all runs will be  $O(f(s))$ . It is

possible the randomization scheme can perform badly by duplicating too many, or too few, caches. But almost surely such situation does not occur every time in the long run.

#### 4 Expected Performance of the Randomization Scheme

We start by defining some symbols which will be used in the performance analysis. We will use  $n$  to denote the array size, which is also the cache size pointed to by a root node. It is always assumed that  $n > 0$ . The real number  $p = \frac{1}{n}$  is the probability that a rotation between a root node and a differential node will result in cutting the edge between them and duplicating the cache; while  $q = 1 - p$  is the probability that the edge is retained and the rotation is just a plain rotation. We use  $l$  to denote the total number of edges in the configuration; the configuration initially is a tree but later may become a forest due to cutting by rotations. Notice that the total number of edges equal the total number of differential nodes in a configuration. For each rotation that cuts the edge between two nodes, the cut reduces the number of differential nodes by one and increases the number of root nodes by one. We use  $m$  to denote the total number of read operations in a given access sequence.

In the analysis, we will only measure the *additional overhead* incurred by the randomization scheme in the process of rerooting. We count 1 unit of time for a plain rotation and  $n + 1$  units of time of a rotation that duplicates the cache and cuts the edge. We do not count the cost of creating the initial array, the cost of appending the differential nodes due to update operations, and the cost of accessing the caches for read operations. Measuring only the overhead is more useful if we want to compare the overhead incurred by the randomized method with that of the optimal off-line solution.

The function  $R(l, m)$  denotes the expected overhead of the randomized method on a configuration of  $l$  edges for a sequence containing  $m$  reads. How these edges connect to one another, however, is not specified, but will be put into details if necessary. So, for a sequence of  $1 + l + m$  array operations which starts with a single array creation and includes  $l$  updates and  $m$  reads, the expected *total cost* of the randomization scheme will be  $n + l + m + R(l, m)$ . Similarly, the function  $Opt(l, m)$  denotes the overhead of the optimal off-line solution. For the same sequence of  $1 + l + m$  array operations, then the total cost of the optimal solution will be  $n + l + m + Opt(l, m)$ . We aim to measure the relationship between  $R(l, m)$  and  $Opt(l, m)$  for the same access sequence.

The optimal algorithm is given the access sequence in its entirety when deriving its solution; while the randomization scheme must serve the requests on-line. The optimal algorithm try to cut the rerooting path, and duplicate the caches, in a way that it reduces the overhead as much as possible. However, it obeys the same restrictions as the randomization scheme in that 1) a read to a differential node must first make it a root node, and 2) if a rotation duplicates a cache from the root node and gives it to the differential node, the rotation must also cut the edge between the two nodes. 1) means that we do not allow the optimal algorithm to search a path to satisfy a read operation, while without moving the cache to the node to be read. 2) means that we do not allow a differential node to point to more than one node at the same time; otherwise we may have unbound number of out-going pointers from a differential node.

We assume that all update operations precede all read operations. We will later show how to relax this restriction without affecting the analysis. Now, instead of asking how good the randomization scheme can be, our first question is: how bad we can expect from it in the worst cases?

**Proposition 4.1**  $R(0, m) = 0$ , and

$$R(l, m) \leq \sum_{i=0}^l \binom{l}{i} p^i q^{l-i} \left( l + \frac{i}{p} + R(l-i, m-1) \right)$$

for all  $l > 0$ .  $\square$

**PROOF.** We give an upper bound of  $R(l, m)$  by employing an adaptive on-line adversary. It is obvious that  $R(0, m) = 0$  because a configuration with no edge contains only root nodes, and reads to a root node incur no overhead. If  $l > 0$ , then a read to a differential node may cut, randomly, along its rerooting path. After each read operation, the adversary looks at the resulting configuration. It then picks a node in the forest which has the longest distance to its corresponding root node. It demands this array to be read next.

The initial configuration is a tree of  $l$  edges. Successive read operations may cut the tree into a forest, and each cut reduces the number of edges by one. For a configuration of  $l$  edges, the adversary, at most, can pick a node with a rerooting path of length  $l$ .

After such a node is picked by the adversary, the randomization scheme with probability  $\binom{l}{i} p^i q^{l-i}$  cuts the rerooting path of  $l$  edges into  $i$  disjoint segments. It also reduces the total number of edges by  $i$ . Such a rerooting costs  $l + \frac{i}{p}$  time, where  $l$  is for the rotations and  $\frac{i}{p}$  is for the duplications of  $i$  caches which each costs  $\frac{1}{p} = n$  units of time. At the next round, the adversary, at most, can only pick a node with a rerooting path of length  $l - i$ .

Summing up the costs for all  $0 \leq i \leq l$ , each with probability  $\binom{l}{i} p^i q^{l-i}$ , we then bound  $R(l, m)$  by

$$R(l, m) \leq \sum_{i=0}^l \binom{l}{i} p^i q^{l-i} \left( l + \frac{i}{p} + R(l-i, m-1) \right)$$

for all  $l > 0$ .  $\diamond$

But, how large is  $R(l, m)$  actually? The recurrence relation is solved in the following lemma. A detailed proof is in Appendix A.

**Lemma 4.2**  $R(l, m) \leq 2l \cdot \frac{1-q^m}{1-q}$ .  $\square$

Since  $1 - q = p = \frac{1}{n}$ , we then have  $R(l, m) \leq 2nl(1 - q^m) \leq 2nl$ . That is, in the worst cases, the expected overhead of the randomization scheme is bounded only by  $l$  (the total number of updates) and  $n$  (the array size), but not by  $m$  (the total number of reads). This is quite different from the fragmented shallow binding scheme and the shallow binding scheme in that their overheads are also bounded by  $m$ . In those two schemes, there can be some reads whose associated overhead (which can be linear to  $l$  or  $n$ ) remains the same during the entire sequence. In the randomization scheme, however, we can expect the overhead associated with a read to the same array diminishes exponentially for

each successive read. Also note that we bound  $R(l, m)$  by a very powerful opponent, an on-line adversary that peeks into the resulting configuration in order to give us the most headache. But in fact the resulting configuration is hidden from the users, and they cannot construct those mischievous sequences. So we can expect a much less upper bound.

Function  $R(l, m)$  describes the overall computational behavior of the randomization scheme, but does not give any information on the effect of the randomization on an existing configuration. We now set out to show the effect of random cutting on some configurations.

Let  $v_0, v_1, \dots, v_i, \dots, v_l$  be a sequence of nodes in the configuration such that  $v_i$  points to  $v_{i+1}$  for each  $0 \leq i \leq l-1$ . Furthermore, let  $v_l$  be a root node. A read operation directed to node  $v_0$  will cause a rerooting from node  $v_l$  to node  $v_0$  and cut the path between them randomly. Let  $L(i)$  be the expected distance between node  $v_i$  and its root (which is not necessarily  $v_0$ ) just after the rerooting. We then can show the following.

**Lemma 4.3**  $L(i) = (n-1)(1-q^i)$ , for all  $0 \leq i \leq l$ .  $\square$

PROOF. Let us look at the edge connecting nodes  $v_i$  and  $v_{i-1}$ . This edge can be cut, with probability  $p$ , during the rerooting. In this case, the distance between  $v_i$  and its root is 0 because  $v_i$  is already a root. If, with probability  $q$ , the edge is not cut, we then look at the edges between node  $v_{i-1}$  and  $v_{i-2}$  too see if it is cut or not. If it is, then  $L(i) = 1$ , with probability  $pq$ . If it is not cut, we then look at the edges between node  $v_{i-2}$  and  $v_{i-3}$ , and so on. Until we reach node  $v_0$ , which must be a root node. If we reach  $v_0$  without cutting any edges in between, we then conclude that, with probability  $q^i$ ,  $L(i) = i$ .

Summing up, we get

$$L(i) = \sum_{k=0}^{i-1} kpq^k + iq^i.$$

Let  $S = \sum_{k=0}^{i-1} kpq^k$ . Then,

$$\begin{aligned} (1-q)S &= \sum_{k=0}^{i-1} kpq^k - \sum_{k=1}^i (k-1)pq^k \\ &= \sum_{k=1}^{i-1} pq^k - (i-1)pq^i = (q - q^i) - (i-1)pq^i. \end{aligned}$$

Since  $1 - q = p = \frac{1}{n}$ , it follows that

$$L(i) = S + iq^i = nq - nq^i + q^i = (n-1)(1-q^i).$$

$\diamond$

In order to get an idea of how large, or small,  $L(i)$  is, we need to show the following two results. The proofs are not difficult and are omitted to save space.

**Lemma 4.4** If  $0 \leq i \leq n$ , then  $1 - \frac{i}{n} \leq q^i \leq 1 - \frac{i}{2n}$ .  $\square$

**Corollary 4.5** If  $i > n > 0$ , then  $0 \leq q^i < \frac{1}{2}$ .  $\square$

By Lemma 4.3, 4.4, and Corollary 4.5, we now bound  $L(i)$  by the following.

**Corollary 4.6**

$$\begin{aligned} \frac{1}{2} \cdot qi &\leq L(i) \leq qi, & \text{if } 0 \leq i \leq n; \\ \frac{1}{2} \cdot (n-1) &< L(i) \leq (n-1), & \text{if } i > n. \end{aligned}$$

$\square$

What does this mean? It means that, after a rerooting sequence passing node  $u$  and reaching at node  $v$ , the root for node  $u$  can be found in distance related to the distance between  $u$  and  $v$ . More precisely, if the distance between  $u$  and  $v$  is larger than  $n$ , then the root for  $u$  can be expected to be  $n-1$  edges away from it at the most. If the distance is smaller than  $n$ , then distance between  $u$  and its root diminishes at the rate of  $q = 1 - \frac{1}{n}$  for each successive rerooting. This implies that the random cutting can be expected to work well for both long and short rerooting paths.

Now, let us try an example to see how well the randomization performs. In particular, we want to compare it to the optimal algorithm. This example is a generalized case described by Figure 4. That is, we want to consider the case of issuing  $m$  alternative reads to two versions of an array which are  $l$  edges away from each other. Formally, let us assume that there are  $l+1$  different versions of array:  $v_0, v_1, \dots, v_i, \dots, v_l$ . They form a single line in the version tree, with  $v_0$  at one end and  $v_l$  at the other. Initially  $v_0$  is the root, with  $v_1$  points to  $v_0, \dots$ , and  $v_l$  points to  $v_{l-1}$ . A sequence of  $m$  read operations is then directed to node  $v_0$  and  $v_l$  alternatively, with the first read to  $v_l$ , the second read to  $v_0$ , the third read to  $v_l$ , the fourth read to  $v_0$ , and so on.

How well will the optimal off-line method perform? If  $ml \leq l+n$ , then it does not pay to duplicate any cache. Therefore the optimal algorithm will need  $ml$  additional overhead to reroot the cache between node  $v_0$  and  $v_l$  repeatedly for  $m$  times. If  $ml > l+n$ , then the optimal algorithm, at the very first read, will duplicate the cache associated with node  $v_0$  and reroot this cache to node  $v_l$ , leaving both  $v_0$  and  $v_l$  each with its own cache. Successive reads then fetch data from the caches at node  $v_0$  and  $v_l$  with no additional overhead. The total overhead is then  $l+n$ , where  $n$  is for duplicating the cache and  $l$  for the first, and the only, rerooting between  $v_0$  and  $v_l$ . In short, we have

$$\begin{aligned} Opt(l, m) &= ml, & \text{if } 0 \leq ml \leq l+n; \\ Opt(l, m) &= l+n, & \text{if } ml > l+n. \end{aligned}$$

We now relate the expected performance of the randomization scheme to that of the optimal solution.

**Theorem 4.7** If  $m$  reads are issued alternatively to two arrays which are initially  $l$  edges apart as described above, then

$$R(l, m) \leq 2 \cdot Opt(l, m)$$

for all  $l$  and  $m$ .  $\square$

PROOF. For the randomized method, it is expected to incur  $2l$  overhead for the first read. This is because each of the  $l$  rotations between  $v_0$  and  $v_l$  needs 1 unit of time for the rotation and is expected to need another  $pn = 1$  unit of time to duplicate the cache. After the first read, the second read is directed to node  $v_0$ , which, by Lemma 4.3, will expect  $2(n-1)(1-q^l)$  unit of time. (Again, two units of time for each of the expected edges between  $v_0$  and its root.) But what about the reads afterward?

If the third read finds out that  $v_l$  has the cache, then it costs no additional overhead for the read, so is all the following reads. This is because both  $v_0$  and  $v_l$  are root nodes now. If  $v_l$  is not a root node, then the nearest root can only be as far as at  $v_0$ . It costs at most  $2(n-1)(1-q^l)$  for rerooting. But what is the probability that  $v_l$  is not the root node in the first place? That  $v_l$  is not a root node at the beginning of the third read means that the root had been rerooted to  $v_0$  for the purpose of the second read. But this is possible only if the first read does not cut any edges between node  $v_0$  and  $v_l$ . This occurs with probability  $q^l$ . Therefore, the expected cost of the third read is  $2(n-1)(1-q^l)q^l$ .

By the same reasoning, the expected cost for the  $(i+2)^{th}$  read is  $2(n-1)(1-q^l)(q^l)^i$ . Summing up, the total expected overhead of the  $m$  reads is

$$\begin{aligned} R(l, m) &= 2l + \sum_{i=0}^{m-2} 2(n-1)(1-q^l)(q^l)^i \\ &= 2l + 2(n-1)(1-q^{(m-1)l}) \end{aligned}$$

for all  $m \geq 1$ . We then conclude

$$R(l, m) < 2l + 2(n-1) < 2(l+n) = 2 \cdot Opt(l, m)$$

if  $ml > l+n$ . If  $ml \leq l+n$ , then either  $0 \leq (m-1)l \leq n$  or  $m=0$ . If  $0 \leq (m-1)l \leq n$ , by Lemma 4.4,

$$R(l, m) \leq 2l + 2q(m-1)l = 2(p+qm)l \leq 2ml = 2 \cdot Opt(l, m).$$

For the remaining case of  $m=0$ , it is obviously that

$$R(l, 0) = 0 = 2 \cdot Opt(l, 0).$$

Summarized, we get  $R(l, m) \leq 2 \cdot Opt(l, m)$  for all  $l$  and  $m$ .  $\diamond$

Note that the above Theorem is true no matter how large, or small,  $l$  is. The randomization scheme solves both the problems described by Figure 2 and 4! The additional overhead incurred by the randomization scheme is expected to be less than twice of that of the optimal solution.

However, in the above case, only two nodes,  $v_0$  and  $v_l$ , are read in the entire read sequence. This is too restrictive. We now generalize the case to the following: the  $(2i)^{th}$  read is directed to node  $v_0$ , but the  $(2i+1)^{th}$  read is directed to node  $v_{f(i)}$ , where  $f$  is a monotonic function. That is, while the even-numbered read operation is always directed to the original node  $v_0$ , we allow the odd-numbered read operation to be further away from the original node for each successive alternation. This is a useful generalization in the following sense: the odd-numbered reads can be thought of mixing with incoming updates. We can think of the gap between the  $(2i+1)^{th}$  read and the  $(2i+3)^{th}$  read being created by  $f(i+1) - f(i)$  additional updates directed to node  $v_{f(i)}$ , one by one. After the updates, we read the resulting array. We will character a read/update sequence with this kind of reference behavior as a *thread*. Right now we have only one thread (which can look back to its origin periodically). We will later show how to generalize it to the multi-threaded cases.

To analyze the performance of a single thread, we still assume that, initially, nodes  $v_0, v_1, \dots, v_i, \dots, v_l$ , form a single line in the version tree, with  $v_0$  — the root node — at one end and  $v_l$  at the other. The last read, the  $m^{th}$ , is assumed to be directed to node  $v_l$  if  $m = 2k + 1$  is odd and

$l = f(k)$ . If  $m = 2k + 2$  is even, the last read is directed to node  $v_0$ . By an argument similar to Theorem 4.7, we can show the following.

**Corollary 4.8** For the above case of  $m$  alternations, in which the even-numbered read operations move successive further away from the origin and finally reach node  $v_l$ , we have

$$R(l, m) \leq 2l + 2(n-1)(1-q^{(m-1)l})$$

and

$$R(l, m) \leq 2 \cdot Opt(l, m)$$

for all  $l$  and  $m$ .  $\square$

So, the expected performance of a single thread (with periodical looking-back) under the randomization scheme is still comparable to that of the optimal performance. Now, we consider the cases of  $w$  independent threads sharing a single origin, resulting a “spokey” configuration as illustrated in Figure 5. This is exactly the image processing situation we discuss earlier in Section 1. The execution of a thread can be interrupted by another thread if the reroot operation occurring in the other thread happens to request the cache under the possession of the current thread. Notice that the execution of a thread can be switched to other threads without interruption if other threads have their own caches.

**Theorem 4.9** Suppose that, in the above described spokey  $w$ -threads case, thread  $s_i, 1 \leq i \leq w$ , is switched  $m_i$  times in order to execute other threads. Further suppose that thread  $s_i$  performs  $l_i$  update operations. Then the total overhead incurred by the randomization scheme is bounded by

$$\sum_{i=1}^w (2l_i + 2(n-1)(1-q^{2m_i l_i}))$$

$\square$

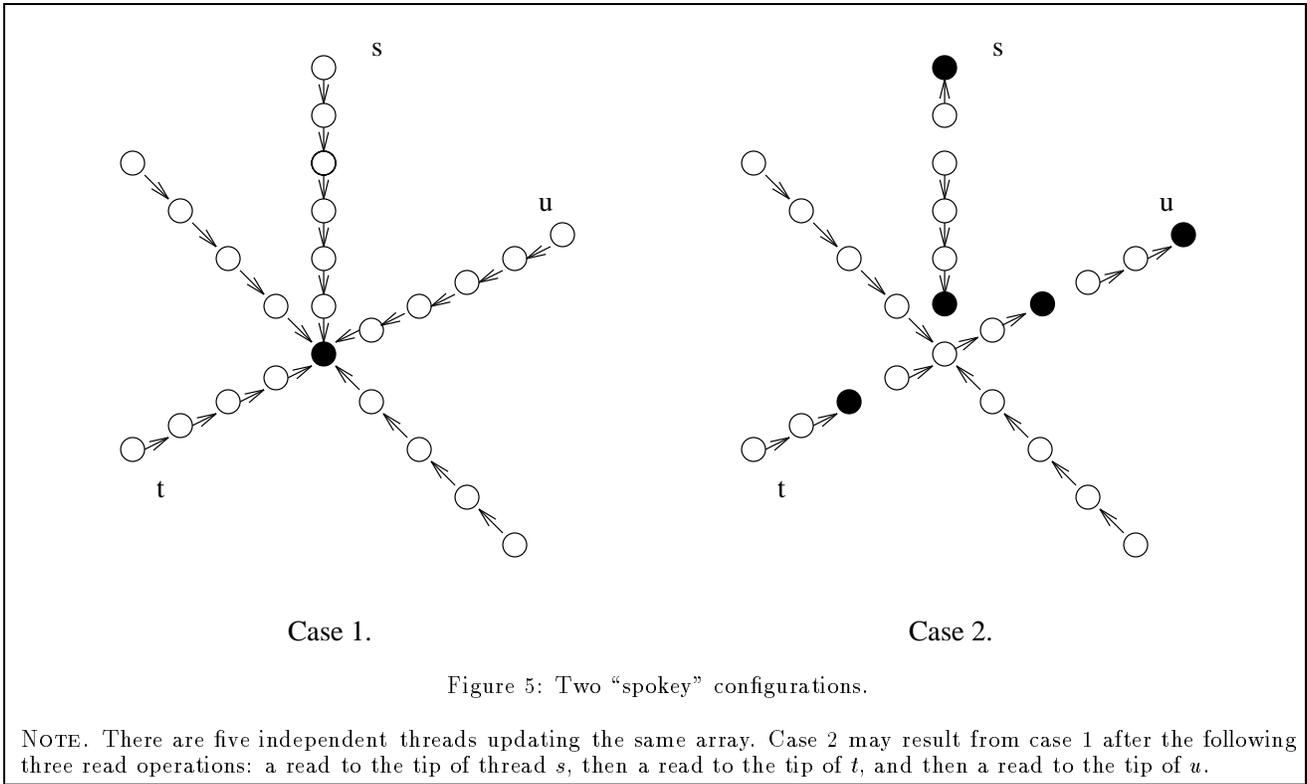
**PROOF.** If a thread  $s_i$  is switched  $m_i$  times, then it requires at most  $2m_i + 1$  alternations between the tip of the thread and the origin. When a thread is switched, the root at its tip is rerooted to the origin, then the root is rerooted to the tip of requesting thread to satisfy the request. Notice that not each switch would need a rerooting en route to the origin in order to satisfy the requesting thread. But the overhead incurred by such a route is an upper bound of the actual overhead.

By Corollary 4.8, the  $m_i$  interrupts to thread  $s_i$  causes at most  $2l_i + 2(n-1)(1-q^{2m_i l_i})$  overhead. Summing up the overheads for all threads, the result follows.  $\diamond$

Notice that we are not able to bound the above overhead with respect to that of the optimal solution. This is because the composition of local optimal solutions not necessarily gives a global optimal solution. That is, although we have bound like

$$R(l_i, 2m_i + 1) \leq 2 \cdot Opt(l_i, 2m_i + 1)$$

for each thread  $s_i$ , we do not know the relation between  $\sum_{i=1}^w Opt(l_i, 2m_i + 1)$  and  $Opt(\bigsqcup_{i=1}^w l_i, \bigsqcup_{i=1}^w m_i)$ , where  $\bigsqcup$  is just an ad-hoc notation to describe the composition of the whole problem from the isolated local problems. Also, using  $\sum_{i=1}^w R(l_i, 2m_i + 1)$  to bound  $R(\bigsqcup_{i=1}^w l_i, \bigsqcup_{i=1}^w m_i)$  gives a



overly pessimistic estimation of the actual overhead. Still, by Theorem 4.9, we now have

$$R\left(\bigoplus_{i=1}^w l_i, \bigoplus_{i=1}^w m_i\right) < 2wn + 2 \sum_{i=1}^w l_i$$

which states that the total overhead of the randomization scheme is less than twice the total cost of updates and caches (with one cache for each thread).

The spokey configuration is a special case of trees. In a tree, there may be updating threads that shared a common update sequence to the origin node. We can count  $l_i$ , the number of updates in that thread, by the number of edges from its tip to the origin. Such counting will overcount because shared paths are counted more than once. Therefore,  $\sum_{i=1}^w l_i$  can be larger than the total number of updates. Nevertheless, Theorem 4.9 gives an upper bound of the total overhead.

## 5 A Preliminary Implementation

We have implemented the randomization scheme on top of the Chalmers Lazy ML (LML) compiler [3, 4]. LML is strongly typed, lazy, and purely functional. It provides primitives to construct monolithic arrays but admits no update to these arrays. We add a function, `write`, which performs destructive modification to an array. We also write a small library for references à la Standard ML, to provide a general mechanism for referencing, dereferencing, and assignment at the programmer level. These added-on routines are written in a low level machine language, the M-code,

and are properly decorated to give out useful heap profiling information when executed. The heap profiling tool is designed by Runciman and Wakeling [18].

Adding `write` and references to LML amounts to adding side-effecting features to the language. Together with these low-level routines and the LML primitive `seq`, which is of type  $*a \rightarrow *b \rightarrow *b$  and returns the second argument after evaluating the first argument, we then write — all in LML — a module for purely functional arrays. Parts of the code are included in Appendix B. Notice that, in a lazy language like LML, `seq` is very useful for synchronizing side-effects. For example, `seq (reroot array) (read array index)` dictates that a reroot operation has to be performed first when reading an array not yet represented by a root node.

There is one problem we have not addressed: how to generate fast (and good) random bits. We are not too concerned about this problem because random bits are required only when performing rerooting. If read operations are mostly directed to root nodes, random bits are not needed at all. Update operations do not need random bits either. Therefore, we probably will not need too many random bits once the resulting version forest has evolved to fit the read pattern of a given program. Another way of saying this is that, for programs with good reference locality, the overhead incurred by rerooting (and that of generating random bits) may not be so significant when compared to the overall cost of array operations. We use the built-in random number generator of LML.

Instead of running a number of benchmark to see how well the randomization scheme performs, we choose to test the randomization scheme, along with other schemes for

purely functional arrays, on only one problem: the histogram problem. The histogram problem is to classify a sequence of incoming events into a fixed set of categories, and to query the distribution of events among the categories. 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. The histogram problem is interesting in itself in several ways. For example, the problem can be expressed single-threadedly if we know in advance that only the final distribution will ever be needed. In a strict language like Standard ML, it is easy to express this single-threadedness because programmers have firm control over the evaluation order of function applications. In a lazy language like LML, however, the evaluation order is not so clear to the programmers. It has been shown that, in a lazy setting, a straightforward implementation of a histogram by the shallow binding scheme results in bizarre zip-zap multi-threaded accesses to the associated cache, even only the final distribution is ever demanded [13]. It will be interesting to see how well the randomization scheme performs in this multi-threaded setting, and, under a lazy setting, how to conduct a single-threaded execution of the histogram.

We show the heap profiles resulting from using different implementation schemes of functional arrays when evaluating the final distribution of a 64-entry histogram after a sequence of  $64 \times 64$  random events. The results, along with some explanations, are shown in Figure 6. The histogram program is shown in Appendix B. All runs use the same histogram program, but utilize different implementations of functional arrays.

## 6 Implications

The concept of normal forms, and its unique existence for a terminating expression, occupies an important place in the foundations of functional programming. Evaluating an expression corresponds to its reduction to the normal form, and, once evaluated, an expression remains in its normal form. Suppose we take the above principle literally, and use the cache representation as the normal form of an array expression. Then, after evaluating an array update expression, we should get a cache normal form for the newly updated array, *without* disturbing the cache normal form of the old version. This is too costly because we end up with two caches. One interesting aspect of Baker’s shallow binding scheme is that, even after it is evaluated, an array expression may not always retain its normal form — the cache. An array in its normal form may get un-evaluated if evaluations of other array expressions demand the cache currently under its possession. That is the reroot operation. The randomization scheme takes this further. An array in its normal form will remain so only with probabilistic assurance if its cache is demanded by other arrays. Even though an evaluated expression may not always in its normal form representation, however, it still provides the same functionality. This separation of representation and functionality of an evaluated expression is a key design in the shallow binding scheme, the randomization scheme, and several other efficient implementations of functional aggregates [9, 11, 22].

It has been suggested that cuts to rerooting paths have some beneficial effects to garbage collection [14]. For example, in the final configuration of Figure 3, the segment with arrays  $E$  and  $F$  can be garbage collected if the only

remaining references are to arrays  $D$  and  $G$ . However, we cannot collect  $E$  or  $F$  in the previous two configurations (of the same figure) because  $D$  and  $G$  are mutually defined by these intermediate nodes. In short, cutting rerooting paths also cuts out some artificial dependency, and this is good for efficient usage of heap space. We can observe this phenomenon in the heap profile of the randomization scheme as shown in Figure 6, where the heap space usage diminishes quickly after sufficiently many links are cut (and many copies of the cache are made).

The randomization scheme can be applied to implement other functional aggregates as well. Suppose that a functional aggregate is implemented by the method of reversible differential list. That is, there is a unique ephemeral representation of the aggregate and other versions are represented by a modification sequence to this ephemeral copy. Modifications can be performed, and undone, by means of rotations. (In the array case, the modification and the undo operations happen to be the same: a destructive update to the cache.) If the current root node has an ephemeral aggregate  $E$  of size  $|E|$ , then a rotation, originally of cost  $t$ , between this node and a differential node pointing to it will cut the edge between them, and duplicate  $E$ , with probability  $\frac{t}{|E|}$ . Such a rotation-with-random-copying is then expected to cost  $2t$  time, twice the original rotation cost, but at the benefit of cutting dependency in the long run. The ephemeral size,  $|E|$ , and the rotation cost,  $t$ , are fixed in the array case, but may not so in other settings.  $E$  may grow or shrink after each rotation, and its size may not be easily calculated, especially when some parts of  $E$  are shared. Still, such a randomization scheme may be worthy of further investigation.

## 7 Conclusion

To copy or not to copy? Let’s toss a coin and see!

## 8 Acknowledgment

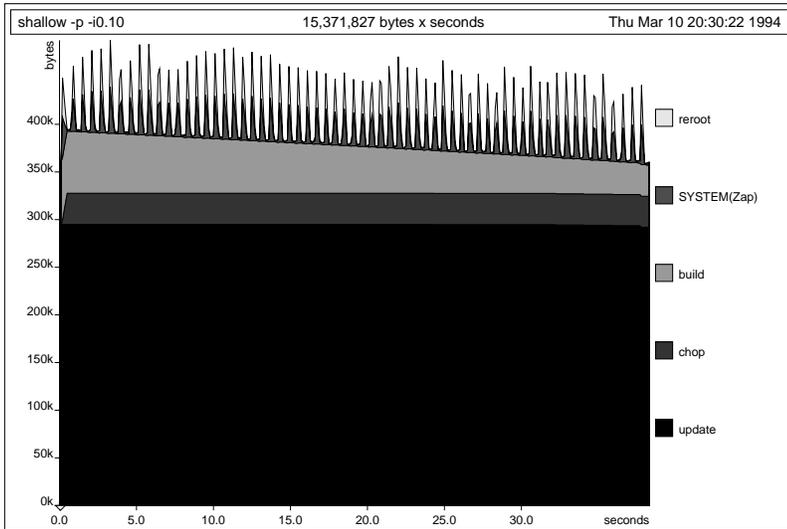
This work was initiated when I was visiting the Department of Computing Science, Chalmers University of Technology, from September 1993 to December 1993. I would like to thank the department, and especially the “multi” group, for their hospitality. I am very grateful to Thomas Johnsson for discussions, for showing me the bizarre behavior of the histogram program under the shallow binding, and for explaining to me the working of the LML compiler. I thank Colin Runciman for their wonderful heap profiling tool, and Herbert Kuchen for helpful suggestions. Finally, I would like to thank Henry G. Baker for his comments on a draft of [8], which keep me thinking about this problem.

## A Proof

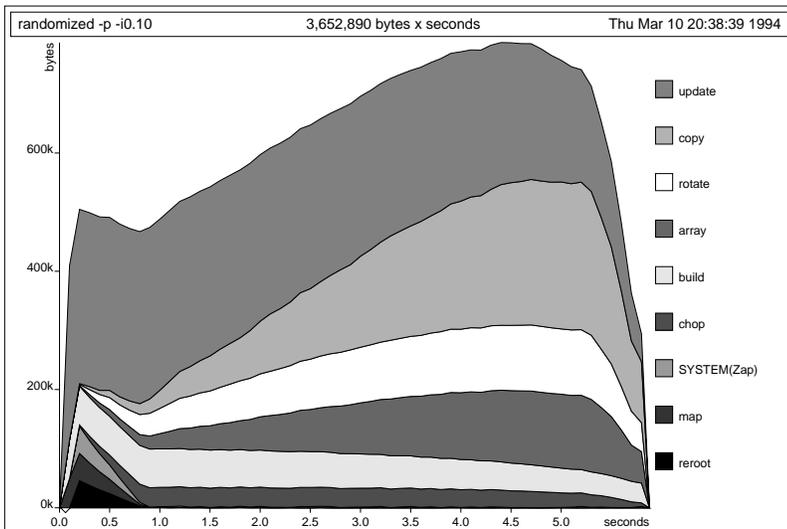
**Lemma A.10**  $R(l, m) \leq 2l \cdot \frac{1-q^m}{1-q}$ . □

PROOF. From Proposition 4.1, we can derive, for  $l > 0$ ,

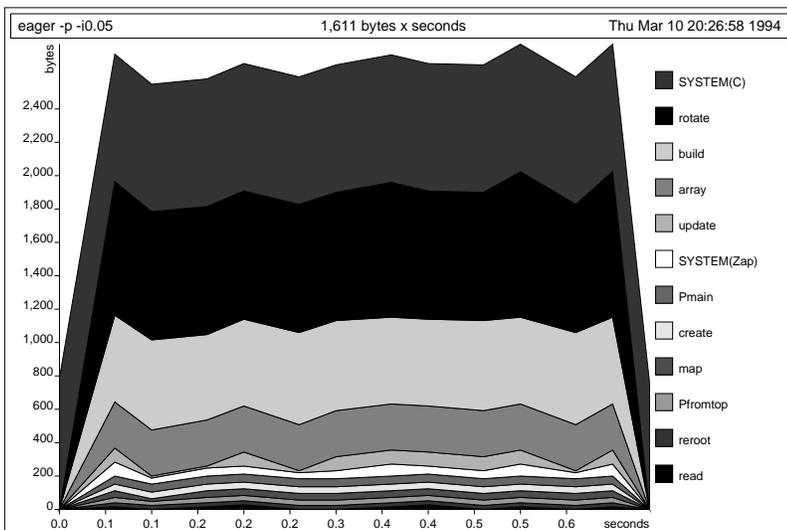
$$R(l, m) \leq l \sum_{i=0}^l \binom{l}{i} p^i q^{l-i} + \sum_{i=0}^l \binom{l}{i} p^i q^{l-i} \frac{i}{p} + \sum_{i=0}^l \binom{l}{i} p^i q^{l-i} R(l-i, m-1)$$



The heap profile of the histogram program, when the arrays are implemented by the shallow binding scheme. First we notice that there are many spines. If we count very carefully, we can see there are 64 of them. Each spine represents the usage of transient space when rerooting the path between the final array and the initial array. That is because, when reading the final distribution at the end of events, each entry in the final array is represented as an expression of a sequence of increments over the corresponding entry of the initial array. Evaluating this expression will demand the initial array, hence, the rerooting. The 4096-edge-long path between the final array and the initial array never get broken, and occupies a big chunk of the heap (*i.e.*, the space in “build,” “chop,” and “update”) to the end. This run demands 15.37 mbs (million byte second), and takes 38 seconds.



The heap profile of the randomization scheme, running the same histogram program. The first thing to notice is that the profiles are not of the same scale. This run demands 3.65 mbs and takes 6 seconds. Similarly to the shallow binding scheme, evaluating the final array will demand the initial array. However, when rerooting the cache from the initial array to the final array, the path between them are randomly cut. We can expect that, after a few rerooting, both the initial array and the final array have their own caches. No rerooting is needed afterward. The cuts, however, need space for copies of caches. That is the space occupied by “copy,” “rotate,” and “array.” This additional space takes up roughly 400 kb at the peak, almost the same amount used by the original 4096-edge-long path.



The heap profile of a single-threaded execution of the same histogram program. It demands only 1.6 kbs and takes under 0.7 seconds. This is done with the EAGER flag turned on (in the code of update, see Appendix B). This causes the update operation to evaluate the value to be assigned to the array when updating. (Previously an update is non-strict in all of its 3 arguments.) Though unsound, in this case we get rid of the dependency to the old histogram as soon as possible. The old histogram can be discarded accordingly as well. The cache is overwritten in a single-threaded matter and is passed from the initial array to the final array. Only one array is kept and the execution takes up at most 2.7 kb heap space at all time. This is what a clever compiler would have done.

Figure 6: The heap profiles of the histogram program, under three implementation schemes of functional arrays.

$$= 2l + \sum_{i=0}^l \binom{l}{i} p^i q^{l-i} R(l-i, m-1)$$

because

$$\begin{aligned} & l \sum_{i=0}^l \binom{l}{i} p^i q^{l-i} + \sum_{i=0}^l \binom{l}{i} p^i q^{l-i} \frac{i}{p} \\ &= (p+q)^l + l \sum_{i=1}^l \binom{l-1}{i-1} p^{i-1} q^{l-i} \\ &= l + l \sum_{i=0}^{l-1} \binom{l-1}{i} p^i q^{(l-1)-i} \\ &= 2l \end{aligned}$$

We now prove the lemma by an induction on  $l$ . The induction base, for  $R(0, m)$ , is true because  $R(0, m) = 0 \leq 2 \cdot 0 \cdot \frac{1-q^m}{1-q}$ .

From the induction hypotheses, that  $R(k, m) \leq 2k \cdot \frac{1-q^m}{1-q}$  for all  $0 \leq k \leq l$ , we derive the inductive step by the following. We first observe that

$$\begin{aligned} & R(l+1, m) \\ &\leq 2(l+1) + \sum_{i=0}^{l+1} \binom{l+1}{i} p^i q^{l+1-i} R(l+1-i, m-1) \\ &\leq 2(l+1) + q^{l+1} R(l+1, m-1) + \\ &\quad \sum_{i=1}^{l+1} \binom{l+1}{i} p^i q^{l+1-i} (2(l+1-i) \frac{1-q^{m-1}}{1-q}). \end{aligned}$$

But

$$\begin{aligned} & \sum_{i=1}^{l+1} \binom{l+1}{i} p^i q^{l+1-i} (2(l+1-i) \frac{1-q^{m-1}}{1-q}) \\ &= 2 \frac{1-q^{m-1}}{1-q} \sum_{i=1}^{l+1} \binom{l+1}{i} (l+1-i) p^i q^{l+1-i} \\ &= 2 \frac{1-q^{m-1}}{1-q} (l+1) q \sum_{i=1}^l \binom{l}{i} p^i q^{l-i} \\ &= 2 \frac{1-q^{m-1}}{1-q} (l+1) q ((p+q)^l - q^l) \\ &= 2 \frac{1-q^{m-1}}{1-q} (l+1) q (1-q^l). \end{aligned}$$

We now have

$$\begin{aligned} & R(l+1, m) \\ &\leq 2(l+1)(1+q(1-q^l) \frac{1-q^{m-1}}{1-q}) + q^{l+1} R(l+1, m-1). \end{aligned}$$

Solving the above recurrence relation yields

$$\begin{aligned} & R(l+1, m) \\ &\leq \sum_{i=0}^{m-1} (q^{l+1})^i (2(l+1)(1+q(1-q^l) \frac{1-q^{m-1-i}}{1-q})) \end{aligned}$$

$$\begin{aligned} &= 2(l+1) \sum_{i=0}^{m-1} (\frac{1-q^{l+1}}{1-q} (q^{l+1})^i - q^m \frac{1-q^l}{1-q} (q^l)^i) \\ &= 2(l+1) (\frac{1-q^{l+1}}{1-q} \cdot \frac{1-q^{(l+1)m}}{1-q^{l+1}} - q^m \frac{1-q^l}{1-q} \cdot \frac{1-q^{lm}}{1-q^l}) \\ &= 2(l+1) \frac{1-q^m}{1-q} \end{aligned}$$

This completes the proof.  $\diamond$

## B Code

The main program for the histogram problem is written as the following. The characters following “--” are comments.

```
#include "array.t"
let
rec build []          array = array
  || build (head.tail) array =
    build tail (update array head (read array head + 1))
and sum array bound index acc & (index > bound) = acc
  || sum array bound index acc =
    sum array bound (index + 1) (read array index + acc)
and (low, high, mod) = (1, 64, high - low + 1)
and zeros          = create low high 0
and chop num       = random num % mod + low
and events         = map chop [1 .. mod * mod]
and histogram      = build events zeros
and summation      = sum histogram high low 0
in summation -- will demand all entries of the histogram
-- shall be evaluated to 4096
```

The file `array.t` contains the interface to an implementation of functional arrays. The data type of arrays are defined by the following:

```
rec type Info      == Int # Int # Int
and type Node     *a = NEW *a
                    + CACHE (LArray (Ref *a))
                    + DIFF (Ref (Array *a)) Int (Ref *a)
and type Array *a = ARRAY (Ref (Info # (Node *a))) !
```

where “#” is for tupling, Ref for references, and LArray the built-in LML monolithic arrays. Info specifies the lower-bound, the upperbound, and the size of the array. A Node either denotes a new array with all entries to be initialized to an identical value, or a cache, or a differential node. The array operations are implemented as following:

```
rec create low high item = -- lazy creation
  ARRAY (REF ((low, high, high - low + 1), NEW item))
and update (A as ARRAY (REF (info, node))) index value =

#ifdef EAGER
  seq value -- strict in *value*
#endif

  (ARRAY (REF (info, DIFF (REF A) index (REF value))))
and read (ARRAY (REF (_, CACHE cache))) index =
  case cache ? index in REF value : value end
  || read A index = seq (reroot A) (read A index)
and reroot (A as ARRAY (REF (_, DIFF (REF B) _ _))) =
  seq (reroot B) (rotate A)
  || reroot (ARRAY (REF (_, CACHE _))) = UNIT
  || reroot (ARRAY (a as REF ((info as (low, high, _)),
    NEW item))) =
    let cache = sarray low high 0 [(index, REF item) ;;
      index <- [low .. high]]
    in assign a (info, CACHE cache)
```

where `?` and `sarray` are respectively the LML primitives for indexing and constructing a monolithic array. The construction needs the array's lowerbound, upperbound, index offset, and an associate list. The `assign` function overwrites the content of a reference. The real thing, `rotate`, is defined by the following:

```
-- Kids, don't try this at home!
rec swap (a as REF va) (b as REF vb) =
  seq (assign a vb) (assign b va)

#ifdef RANDOMIZE
and seed = REF 19640824 -- Someone's birthday
and duplicate cache low high =
  let rec items = [(index, cache ? index) ;;
    index <- [low .. high]
    and copy (index, REF v) = (index, REF v)
  in sarray low high 0 (map copy items)
#endifif

and rotate (ARRAY (A as REF ((low, high, count),
  DIFF (RB as REF
    (ARRAY (B as REF (info, CACHE cache))))
    index (ra as REF va)))) =

#ifdef RANDOMIZE
  case seed in REF r :
    seq (assign seed (random r))
      (if r % count = 0 -- once in *count* times!
        then case duplicate cache low high in copy :
          seq (write copy index (REF va))
            (assign A (info, CACHE copy)) end
        else
          ) end
#endifif

  case cache ? index in REF vb :
    seq (assign ra vb) (
      seq (write cache index (REF va)) (
        seq (assign RB (ARRAY A))
          (swap A B)))
  end

#ifdef RANDOMIZE
  ) end
#endifif
```

where `write` is the destructive array update operation.

## References

- [1] Annika Aasa, Sören Holmström, and Christina Nilsson. An efficiency comparison of some representations of purely functional arrays. *BIT*, 28(3):490–503, 1988.
- [2] Arvind, Rishiyur S. Nikhil, and Keshav K. Pingali. I-structures: Data structure for parallel computing. *ACM Transactions on Programming Languages and Systems*, 11(4):598–632, October 1989.
- [3] Lennart Augustsson and Thomas Johnsson. The Chalmers Lazy-ML compiler. *Computer Journal*, 32(2):127–141, April 1989.
- [4] Lennart Augustsson and Thomas Johnsson. *Lazy ML user's manual Version 0.999.4*, July 1994. Draft, distributed with the Chalmers Lazy ML compiler.
- [5] Henry G. Baker, Jr. Shallow binding in Lisp 1.5. *Communications of the ACM*, 21(7):565–569, July 1978.
- [6] Henry G. Baker. Shallow binding makes functional arrays fast. *SIGPLAN Notices*, 26(8):145–147, August 1991.
- [7] Adrienne Bloss. Update analysis and the efficient implementation of functional aggregates. In *Functional Programming Languages and Computer Architecture*, pages 26–38. ACM/Addison-Wesley, September 1989. Imperial College, London, UK.
- [8] Tyng-Ruey Chuang. Fully persistent arrays for efficient incremental updates and voluminous reads. In Bernd Krieg-Brückner, editor, *4th European Symposium on Programming*, pages 110–129. Rennes, France, February 1992. Lecture Notes in Computer Science, Volume 582, Springer-Verlag.
- [9] Tyng-Ruey Chuang and Benjamin Goldberg. Real-time dequeues, multihead Turing machines, and purely functional programming. In *Conference on Functional Programming Languages and Computer Architecture*, pages 289–298. University of Copenhagen, Denmark, June 1993. ACM Press.
- [10] 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.
- [11] Robert Hood and Robert Melville. Real-time queue operations in pure Lisp. *Information Processing Letters*, 13(2):50–53, 1981.
- [12] Paul Hudak and Adrienne Bloss. The aggregate update problem in functional programming systems. In *12th Annual ACM Symposium on Principles of Programming Languages*, pages 300–314. ACM, January 1985. New Orleans, Louisiana, USA.
- [13] Thomas Johnsson. Personal communication, October 1993.
- [14] Herbert Kuchen. Personal communication, September 1993.
- [15] Eugene W. Myers. Efficient applicative data types. In *11th Annual ACM Symposium on Principles of Programming Languages*, pages 66–75. ACM, January 1984. Salt Lake City, Utah, USA.
- [16] Martin Odersky. How to make destructive updates less destructive. In *18th Annual ACM Symposium on Principles of Programming Languages*, pages 25–36. Orlando, Florida, USA, ACM, January 1991.
- [17] Simon L Peyton Jones and Philip Wadler. Imperative functional programming. In *20th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 71–84. Charleston, South Carolina, USA, ACM, January 1993.
- [18] Colin Runciman and David Wakeling. Heap profiling of lazy functional programs. *Journal of Functional Programming*, 3(2):217–245, April 1993.
- [19] Neil Sarnak. *Persistent Data Structures*. PhD thesis, Department of Computer Science, New York University, 1986.
- [20] A. V. S. Sastry, William Clinger, and Zena Ariola. Order-of-evaluation analysis for destructive updates in strict functional languages with flat aggregates. In *Conference on Functional Programming Languages and Computer Architecture*, pages 266–275. University of Copenhagen, Denmark, June 1993. ACM Press.
- [21] David A. Schmidt. Detecting global variables in denotational specifications. *ACM Transactions on Programming Languages and Systems*, 7(2):299–310, April 1985.
- [22] Berry Schoenmakers. *Data Structures and Amortized Complexity in a Functional Setting*. PhD thesis, Department of Mathematics and Computing Science, Eindhoven University of Technology, September 1992.
- [23] Philip Wadler. Comprehending monads. In *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, pages 61–78. Nice, France, ACM, June 1990.