Well, you go in and you ask for some toothpaste—the small size—and the man brings you the large size. You tell him you wanted the small size but he says the large size is the small size. I always thought the large size was the largest size, but he says that the family size, the economy size and the giant size are all larger than the large size—that the large size is the smallest size there is.

Charade (1963)

Chapter 5

Arrays and large objects

Need topic ¶

Name it.

~~The software transaction system implemented in the previous chapter~~ clones objects on transactional writes, so that the previous state of the object can be restored if the transaction aborts. Figure 5.1 shows the object size distribution of transactional writes for SPECjvm98, and indicates that over 10% of writes may be to large objects. As we've seen in Section 4.5, the copying cost can become excessive.

The solution I ~~will~~ propose ~~will~~ represents objects as ***functional arrays***. O'Neill and Burton [74] give a fairly inclusive overview of such algorithms; I've chosen Tyng-Ruey Chuang's version [21] of *shallow binding*, which uses randomized cuts to the version tree to limit the cost of a read to $O(n)$ in the worst case. Single-threaded accesses to the array ~~are~~ run in $O(1)$ time. Our use of functional arrays is single-threaded in the common case, when transactions do not abort. Chuang's scheme is attractive, because it limits the worst-case cost of an abort, with ~~very~~ little added complexity.

Name it.

In this chapter I will recast the transaction system design of Chapter 3 as a "small object protocol," and then show how to extend it to a "large object protocol," in the process addressing the large object performance problems. The large object protocol ~~will~~ uses a lock-free variant of Chuang's algorithm, which I ~~will~~ present in Section 5.4.

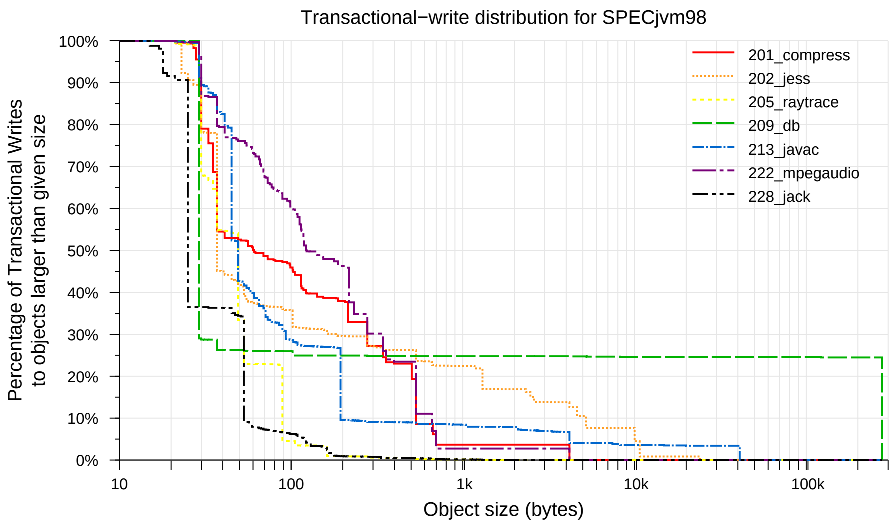


Figure 5.1: Proportion of transactional writes to objects equal to or smaller than a given size.

5.1 Basic operations on functional arrays

Let us begin by reviewing the basic operations on functional arrays. Functional arrays are *persistent*; that is, after an element is updated, both the new and the old contents of the array are available for use. Since arrays are simply maps from integers (indexes) to values, any functional~~-~~map datatype (for example, a functional balanced tree) can be used to implement functional arrays.

~~However,~~ ~~t~~The distinguishing characteristic of an imperative array is its time complexity: $O(1)$ time to access or update ~~of~~ any element. Implementing functional arrays with a functional balanced tree yields $O(\lg n)$ worst-case access or update.[1]

Define

For concreteness, a functional array~~s~~ defines ~~have~~ the following three operations: ~~defined.~~

[1] I ~~will~~ return to a discussion of operational complexity in Section 5.4.

- FA-CREATE(n): Return an array A of size n. The contents of the array are initialized to ~~zero~~. 0.

- FA-UPDATE(A, i, v): Return an array A' that is functionally identical to array A except that FA-READ(A', i) $= v$. Array A is not destroyed and can be accessed further.

- FA-READ(A, i): Return $A(i)$ (that is, the value of the ith element of array A).

We allow any of these operations to *fail*. Failed operations can be safely retried, as all operations are idempotent by definition.

For the moment, consider the following naïve implementation:

- FA-CREATE(n): Return an ordinary imperative array of size n.

- FA-UPDATE(A, i, v): Create a new imperative array A' and copy the contents of A to A'. Set $A'[i] = v$. Return A'.

- FA-READ(A, i): Return $A[i]$.

Since This implementation ~~has~~ costs $O(1)$ to read and $O(n)$ to update, ~~so~~ it matches the performance of imperative arrays only when $n = O(1)$. I ~~will~~ therefore call these *small-object functional arrays*. Operations in this implementation never fail. Every operation is non-blocking and no synchronization is necessary, since the imperative arrays are never mutated after they are created. ~~In~~ Section 5.4 ~~we will~~ review better implementations of functional arrays, and present ~~our own~~ a new lock-free variant.

No, Only when #updates = O(n · #reads)

5.2 A single-object protocol

Given a non-blocking implementation of functional arrays, we can construct a transaction implementation for single objects. In this implementation, fields of at most one object may be referenced during the execution of the transaction.

~~I will~~ Consider the following two operations on objects:

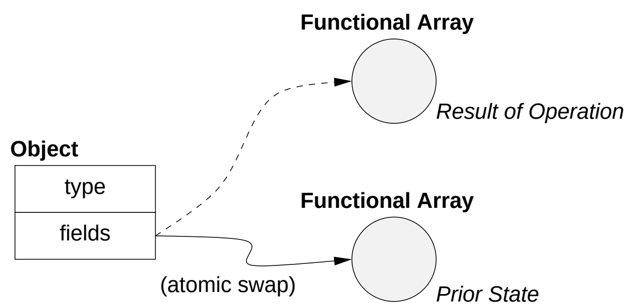


Figure 5.2: Implementing non-blocking single-object concurrent operations with functional arrays.

- READ(o, f): Read field f of o. ~~We will~~ A assume that there is a constant mapping function, which ~~that~~ given a field name, returns an integer index f.index. ~~We will write the result of mapping f as f.index.~~ For simplicity, and without loss of generality, ~~we will~~ assume that all fields sizes are ~~of~~ equal ~~size~~.

- WRITE(o, f, v): Write value v to field f of o.

All other operations on Java objects, such as method dispatch and type interrogation, can be performed using the immutable type field in the object. Because the type field ~~is~~ never changes after object creation, implementing non-blocking ~~implementations of~~ operations on the type field ~~are trivial~~ is straightforward.

As Figure 5.2 shows, our single-object transaction implementation represents objects as a pair, combining type and a reference to a functional array. When not inside a transaction, object reads and writes are implemented using the corresponding functional array operation, with the array reference in the object being updated appropriately:

- READ(o, f): Return FA-READ(o.fields, f.index).

- WRITE(o, f, v): Replace o.fields with the result of FA-UPDATE(o.fields, f.index, v).

The interesting cases are reads and writes inside a transaction. At entry to a ~~our~~ transaction that will access (only) object o, ~~we~~ store o.fields in a

System name

local variable u. We create another local variable u' ~~which we~~ initialize d to u. Then, the ~~our~~ read and write operations are implemented as follows:

- ReadT(o, f): Return FA-Read$(u', f.\texttt{index})$.

- WriteT(o, f, v): Update variable u' to the result of FA-Update$(u', f.\texttt{index}, v)$.

At the end of the transaction, we use Compare-And-Swap to atomically set o.fields to u' iff it contained u. If the CAS fails, ~~we~~ the transaction is aborted (we simply discard u') and we retry.

With our naïve "small object" functional arrays, this implementation is exactly the "small object protocol" of Herlihy [48]. Herlihy's protocol is rightly criticized for an excessive amount of copying. I ~~will~~ address this criticism with a better implementation of functional arrays in Section 5.4. ~~However, the~~ First, however, I remove the restriction that only one object may be referenced within a transaction, ~~is~~ overly limiting. ~~I will~~ first fix this problem.

5.3 Extension to multiple objects

I extend the implementation to allow the fields of any number of objects to be accessed during the transaction. Figure 5.3 shows our new object representation. Compare this to Figure 3.6; we've successfully recast our earlier transaction system design now in terms of operations on an array datatype. name Objects consist of two slots, and the first represents the immutable type, as before. The second field, versions, points to a linked list of Version structures. The Version structures contain a pointer fields to a functional array, and a pointer owner to an *transaction identifier*. The transaction identifier contains a single field, status, which can be set to one of three values: *COMMITTED*, *IN-PROGRESS*, or *ABORTED*. When the transaction identifier is created, the status field is initialized to *IN-PROGRESS*, and it will be updated exactly once thereafter, to either *COMMITTED* or

! A cap letter followed by . is treated as an abbrev in latex. Must use \@.

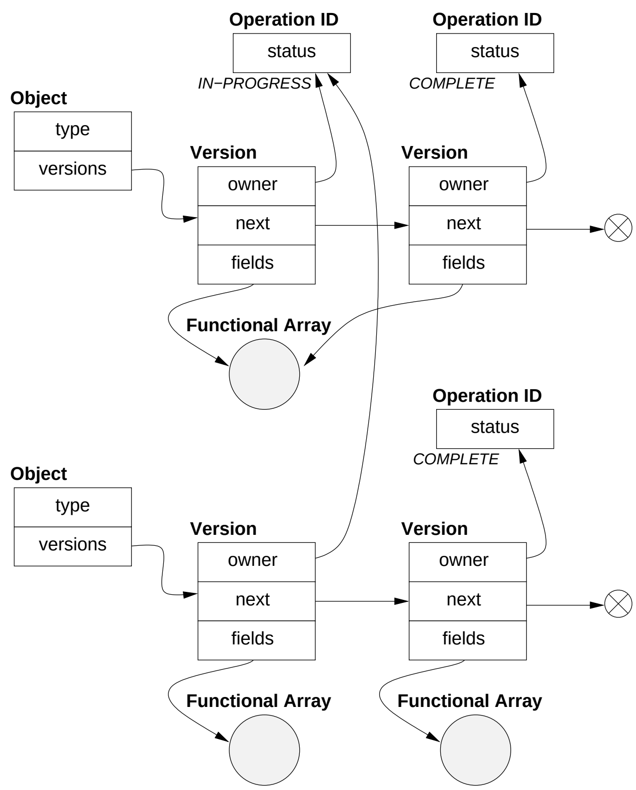


Figure 5.3: Data structures to support non-blocking multi-object concurrent operations. Objects point to a linked list of versions, which reference transaction identifiers. Versions created within the same execution of a transaction share the same transaction identifier. Version structure also contain pointers to functional arrays, which record the values for the fields of the object. If no modifications have been made to the object, multiple versions in the list may share the same functional array. (Compare this model of a transaction system to our concrete design in Figure 3.6.)

```
READ(o, f):
begin
retry:
  u ← o.versions
  u′ ← u.next
  s ← u.owner.status
  if (s = DISCARDED)                        [Delete DISCARDED?]
    CAS(u, u′, &(o.versions))
    goto retry
  else if (s = COMPLETE)
    a ← u.fields                            [u is COMPLETE]
    u.next ← null                           [Trim version list]
  else
    a ← u′.fields                           [u′ is COMPLETE]
  return FA-READ(a, f.index)                [Do the read]
end

READT(o, f):
begin
  u ← o.versions
  if (oid = u.owner)                        [My OID should be first]
    return FA-READ(u.fields, f.index)       [Do the read]
  else                                      [Make me first!]
    u′ ← u.next
    s ← u.owner.status
    if (s = DISCARDED)                      [Delete DISCARDED?]
      CAS(u, u′, &(o.versions))
    else if (oid.status = DISCARDED)        [Am I alive?]
      fail
    else if (s = IN-PROGRESS)               [Abort IN-PROGRESS?]
      CAS(s, DISCARDED, &(u.owner.status))
    else                                    [Link new version in:]
      u.next ← null                         [Trim version list]
      u′ ← new Version(oid, u, null)        [Create new version]
      if (CAS(u, u′, &(o.versions)) ≠ FAIL)
        u′.fields ← u.fields                [Copy old fields]
    goto retry
end
```

Figure 5.4: READ and READT implementations for the multi-object protocol.

```
WRITE(o, f, v):
begin
retry:
  u ← o.versions
  u′ ← u.next
  s ← u.owner.status
  if (s = DISCARDED)                      [Delete DISCARDED?]
    CAS(u, u′, &(o.versions))
  else if (s = IN-PROGRESS)               [Abort IN-PROGRESS?]
    CAS(s, DISCARDED, &(u.owner.status))
  else                                    [u is COMPLETE]
    u.next ← null                         [Trim version list]
    a ← u.fields
    a′ ← FA-UPDATE(a, f.index, v)
    if (CAS(a, a′, &(u.fields)) ≠ FAIL)   [Do the write]
      return                              [Success!]
  goto retry
end
```

```
WRITET(o, f, v):
begin
  u ← o.versions
  if (oid = u.owner)                      [My OID should be first]
    u.fields ← FA-UPDATE(u.fields, f.index, v)[Do write]
  else                                    [Make me first!]
    u′ ← u.next
    s ← u.owner.status
    if (s = DISCARDED)                    [Delete DISCARDED?]
      CAS(u, u′, &(o.versions))
    else if (oid.status = DISCARDED)      [Am I alive?]
      fail
    else if (s = IN-PROGRESS)             [Abort IN-PROGRESS?]
      CAS(s, DISCARDED, &(u.owner.status))
    else                                  [Link new version in:]
      u.next ← null                       [Trim version list]
      u′ ← new Version(oid, u, null)      [Create new version]
      if (CAS(u, u′, &(o.versions)) ≠ FAIL)
        u′.fields ← u.fields              [Copy old fields]
    goto retry
end
```

Figure 5.5: WRITE and WRITET implementations for the multi-object protocol.

\@ ,

ABORTED. A *COMMITTED* transaction identifier never later becomes *IN-PROGRESS* or *ABORTED*, and a *ABORTED* transaction identifier never becomes *COMMITTED* or *IN-PROGRESS*. \@ ,

We create an transaction identifier when we begin or restart a transaction and place it in a local variable *tid*. At the end of the transaction, we use CAS to set *tid*.status to *COMMITTED* iff it was *IN-PROGRESS*. If the CAS is successful, the transaction has also executed successfully; otherwise *tid*.status = *ABORTED* (which indicates that our transaction has been aborted), and we must back off and retry. All Version structures created while in the transaction ~~will~~ reference *tid* in their owner field.

Semantically, the current field values for the object ~~will be~~ are given by the first version in the versions list whose transaction identifier is *COMMITTED*. \@ , This allows us to link *IN-PROGRESS* versions in at the head of multiple objects' versions lists and atomically change the values of all these objects by setting the one common transaction identifier to *COMMITTED*. \@ , We only allow one *IN-PROGRESS* version on the versions list, and it must be at the head. ~~so~~ Thus, before we can link a new version at the head, we must ensure that every other version on the list is *ABORTED* or *COMMITTED*. \@ , etc

Since we ~~will~~ never look past the first *COMMITTED* version in the versions list, we can free all versions past that point. In our presentation of the algorithm, we do this by explicitly setting the next field of every *COMMITTED* version we see to **null**; this allows the versions past that point to be garbage collected. An optimization ~~would be~~ is ~~to have~~ for the garbage collector to do the list trimming for us when it does a collection.

Because We don't want to inadvertently chase the null next pointer of a *COMMITTED* version, ~~so~~ we always load the next field of a version *before* we load owner.status. Since the writes occur in the reverse order (*COMMITTED* to owner.status, then **null** to next), we have ensured that our next pointer is valid whenever the status is not *COMMITTED*.

We begin an atomic method with TRANSSTART and attempt to complete an atomic method with TRANSEND. They are defined as follows:

- TransStart: create a new transaction identifier, with its status initialized to *IN-PROGRESS*. Assign it to the thread-local variable *tid*.
- TransEnd: If

  $$\mathrm{CAS}(\mathit{IN\text{-}PROGRESS}, \mathit{COMMITTED}, \&(\mathit{tid}.\mathtt{status}))$$

  is successful, the transaction as a whole has completed successfully, and can be linearized at the location of the CAS. Otherwise, the transaction has been aborted. Back off and retry from TransStart.

Pseudo-code describing Read, Write, ReadT, and WriteT is presented in Figures 5.4 and 5.5. In the absence of contention, all operations take constant time plus an invocation of FA-Read or FA-Update.

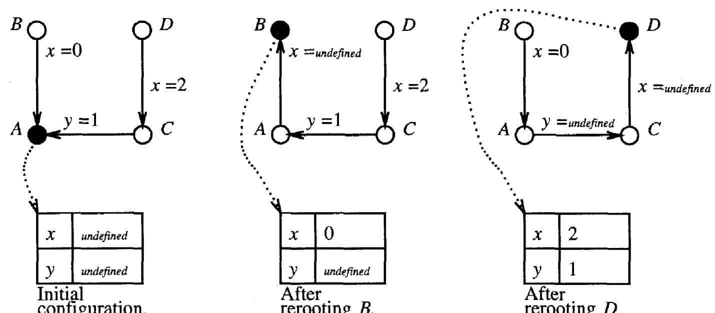
5.4 Lock-free functional arrays

~~In~~ this section ~~I will~~ present a lock-free implementation of functional arrays with $O(1)$ performance in the absence of contention. The crucial operation is a rotation of a *difference node* with the main body of the array. Using this implementation of functional arrays in the multi-object transaction protocol of the previous chapter ~~will~~ complete our re-implementation of non-blocking transactions, solving the large-object problem. name

Let's begin by reviewing the well-known functional array implementations. As mentioned previously, O'Neill and Burton [74] give an inclusive overview. Functional array implementations fall generally into one of three categories: *tree-based*, *fat-elements*, or *shallow-binding*.

Tree-based implementations typically have a logarithmic term in their complexity. The simplest is the persistent binary tree with $O(\ln n)$ look-up time; Chris Okasaki [73] has implemented a purely-functional random-access list with $O(\ln i)$ expected lookup time, where i is the index of the desired element.

Fat-elements implementations have per-element data structures indexed by a master array. Cohen [23] hangs a list of versions from each element in



NOTE. The array is of size 2 and is indexed by x and y. The initial array A is undefined, and B is defined as an update to A at index x by value 0. Similarly for C and D. The dark node is the root node which has the cache. White nodes are differential nodes which must first be rerooted before be read. Note that only the root node has the cache.

Too small

Figure 5.6: Shallow binding scheme for functional arrays, from [21, Figure 1].

the master array. O'Neill and Burton [74], in a more sophisticated technique, hang a splay tree off each element and achieve $O(1)$ operations for single-threaded use, $O(1)$ amortized cost when accesses to the array are "uniform", and $O(\ln n)$ amortized worst case time.

Shallow binding was introduced by Baker [12] as a method to achieve fast variable lookup in Lisp environments. Baker clarified the relationship to functional arrays in [11]. Shallow binding is also called *version tree arrays*, *trailer arrays*, or *reversible differential lists*. A typical drawback of shallow binding is that reads may take $O(u)$ worst-case time, where u is the number of updates made to the array. Tyng-Ruey Chuang [21] uses randomized cuts to the version tree to limit the cost of a read to $O(n)$ in the worst case. Single-threaded accesses are $O(1)$.

Our use of functional arrays is single-threaded in the common case, when transactions do not abort. Chuang's scheme is attractive because it limits the worst-case cost of an abort, with very little added complexity. In this section I will present a lock-free version of Chuang's randomized algorithm.

In shallow binding, only one version of the functional array (the *root*) keeps its contents in an imperative array (the *cache*). Each of the other versions is represented as a path of *differential nodes*, where each node describes the differences between the current array and the previous array. The difference is represented as a pair $\langle index, value \rangle$, representing the new value to be stored at the specified index. All paths lead to the root. An update to the functional array is simply implemented by adding a differential node pointing to the array it is updating.

The key to constant-time access for single-threaded use is provided by the read operation. A read to the root simply reads the appropriate value from the cache. However, a read to a differential node triggers a series of rotations that swap the direction of differential nodes and result in the current array acquiring the cache and becoming the new root. This sequence of rotations is called *re-rooting*, and is illustrated in Figure 5.6. Each rotation exchanges the root nodes for a differential node pointing to it, after which the differential node becomes the new root and the root becomes a differential node pointing to the new root. The cost of a read is proportional to its re-rooting length, but after the first read accesses to the same version are $O(1)$ until the array is re-rooted again.

Shallow binding performs badly if read operations ping-pong between two widely separated versions of the array, as we will continually re-root the array from one version to the other. Chuang's contribution is to provide for *cuts* to the chain of differential nodes: once in a while we clone the cache and create a new root instead of performing a rotation. Since This operation takes $O(n)$ time, ~~so~~ we amortize it over n operations by randomly choosing to perform a cut with probability $1/n$.

Figure 5.7 shows the data structures used for the functional array implementation, as well as ~~and~~ the series of atomic steps used to implement a rotation. The Array class, which represents a functional array; ~~it~~ consists of a size for the array and a pointer to a Node. There are two types of nodes: a CacheNode stores a value for every index in the array, whereas ~~and~~ a DiffNode stores a single change to an array. Array objects that point to CacheNodes are roots.

In step 1 of the figure, we have a root array A and an array B whose differential node d_B points to A. The functional arrays A and B differ in one element: element x of A is z, while element x of B is y. We are about to rotate B to give it the cache, while linking a differential node to A.

Step 2 shows our first atomic action. We have created a new DiffNode d_A and a new Array C and linked them between A and its cache. The DiffNode d_A contains the value for element x contained in the cache, z, so there is no change in the value of A.

We continue swinging pointers until step 5, when we can finally set the element x in the cache to y. We perform this operation with a DCAS operation that checks that C.node is still pointing to the cache as we expect. Note that a concurrent rotation would swing C.node in its step 1. In general, therefore, the location pointing to the cache serves as a reservation on the cache.

Too small

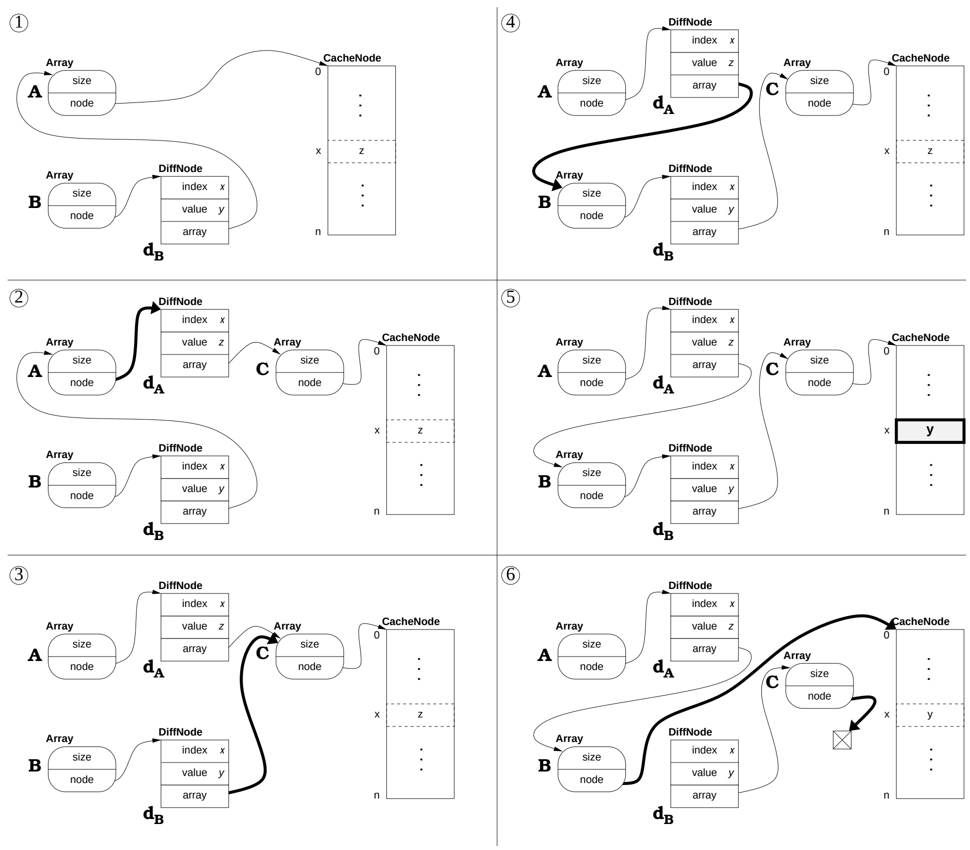


Figure 5.7: Atomic steps in FA-ROTATE(B). Time proceeds top-to-bottom on the left hand side, and then top-to-bottom on the right. Array A is a root node, and FA-READ$(A, x) = z$. Array B has the almost the same contents as A, but FA-READ$(B, x) = y$.

```
FA-UPDATE(A, i, v):
begin
  d ← new DiffNode(i, v, A)
  A′ ← new Array(A.size, d)
  return A′
end

FA-READ(A, i):
begin
retry:
  d_C ← A.node
  if d_C is a cache, then
    v ← A.node[i]
    if (A.node ≠ d_C)[consistency check]
      goto retry
    return v
  else
    FA-ROTATE(A)
    goto retry
end
```

Figure 5.8: Implementation of lock-free functional array using shallow binding and randomized cuts (part 1).

```
FA-Rotate(B):
begin
retry:
  d_B ← B.node      [step (1): assign names as per Figure 5.7.]
  A ← d_B.array
  x ← d_B.index
  y ← d_B.value
  z ← FA-Read(A, x)                        [rotates A as side effect]

  d_C ← A.node
  if d_C is not a cache, then
    goto retry

  if (0 = (random mod A.size))                         [random cut]
    d'_C ← copy of d_C
    d'_C[x] ← y
    s ← DCAS(d_C, d_C, &(A.node), d_B, d'_C, &(B.node))
    if (s ≠ SUCCESS) goto retry
    else return

  C ← new Array(A.size, d_C)
  d_A ← new DiffNode(x, z, C)

  s ← CAS(d_C, d_A, &(A.node))                          [step (2)]
  if (s ≠ SUCCESS) goto retry

  s ← CAS(A, C, &(d_B.array))                           [step (3)]
  if (s ≠ SUCCESS) goto retry

  s ← CAS(C, B, &(d_A.array))                           [step (4)]
  if (s ≠ SUCCESS) goto retry

  s ← DCAS(z, y, &(d_C[x]), d_C, d_C, &(C.node))        [step (5)]
  if (s ≠ SUCCESS) goto retry

  s ← DCAS(d_B, d_C, &(B.node), d_C, nil, &(C.node))[step (6)]
  if (s ≠ SUCCESS) goto retry
end
```

Figure 5.9: Implementation of lock-free functional array using shallow binding and randomized cuts (part 2).

Thus in step 6 we need to again use DCAS to simultaneously swing C.node away from the cache as we swing B.node to point to the cache.

Figures 5.8 and 5.9 present pseudocode for FA-ROTATE, FA-READ, and FA-UPDATE. ~~Note that~~ FA-READ also uses the cache pointer as a reserva- Like FA-Rotate
tion, double-checking the cache pointer after it finishes its read to ensure that the cache hasn't been stolen from it.

Let us now consider cuts, where FA-READ clones the cache instead of performing a rotation. Cuts also check the cache pointer to protect against concurrent rotations. But what if the cut occurs while a rotation is mutating the cache in step 5? In this case because the only array adjacent to the root is B, ~~so~~ the cut must be occurring during an invocation of FA-ROTATE(B)~~. But~~ in which case ~~then~~ the differential node d_B will be applied after the cache is copied, ~~which~~ thereby ~~will~~ safely overwrite ing the mutation we were concerned about.

~~Note that~~ W with hardware support for small transactions [49] we could cheaply perform the entire rotation atomically, instead of using this six-step approach.