A New Approach for Parallel Functional Arrays

Ananya Kumar
Carnegie Mellon University
ananyak@andrew.cmu.edu

Guy Blelloch
Carnegie Mellon University
blelloch@cs.cmu.edu

Abstract

In this paper we introduce a $O(1)$ wait-free, parallel, functional array, that allows $O(1)$ reads and writes to the most recent version of the array. We describe the cost dynamics and sketch out a provable implementation. We show favorable benchmarks comparing our functional arrays with regular arrays in Java.

Categories and Subject Descriptors CR-number [*subcategory*]: third-level

Keywords array, parallel, cost semantics

1. Introduction

Arrays are very important in functional programming languages because they allow work-efficient implementations of algorithms like depth-first search. Accessing old versions of arrays can be useful for efficient checkpointing, logging, and event handling.

1.1 Previous Approaches

There has been a lot of great work on functional arrays, but previous approaches have limitations.

In many functional programming languages, like Haskell, monads (Wadler 1995) can be used to enforce single threaded use of arrays. However, enforcing single threaded use obstructs persistence - it prevents us from accessing or modifying old versions of an array.

Another approach is to use reference counting (Hudak 1986). If the number of references to an array is one, the array can be updated in place, otherwise the array is copied and then updated. This approach makes it difficult for programmers to statically analyze the time complexity of array operations. Furthermore, even if there are multiple reference to an array the array might not need to be copied, so the unnecessary copying could make the implementation inefficient.

A third approach is to use linear types (Wadler 1990) to ensure that programmers can only access and mutate arrays in valid ways. Linear types have not been used practically for a few decades since its inception, and would require a complete re-think of a language's type system to be useful.

O' Neill's (O'Neill 2000) functional arrays guarantee $O(1)$ access and insertions when used like an imperative array. However, the arrays explicitly implement version trees which are inefficient in practice. Additionally, the reads and writes to the most recent version of the array might take $O(\log n)$ work if the array is not used in a single threaded way.

Chuang's approach (Chuang 1992) supports $O(1)$ accesses and insertions to the most recent version of arrays, however accessing old versions of the array takes $O(n)$ work which is often impractical.

None of the existing approaches support concurrent operations on arrays. O' Neill suggests having a lock for each element of the array. However, when many threads contend for the same element, this would serialize accesses and make accesses not $O(1)$. Additionally, per-element locks add significant memory overhead. Separation logic (Reynolds 2002) could be used to parallelize divide and conquer algorithms on arrays however they would severely restrict the way functional arrays can be used.

1.2 Our Approach

Our arrays support 3 functions: tabulate, get, and set. The dynamics are functional, in particular set returns a new array and leaves the original array intact.

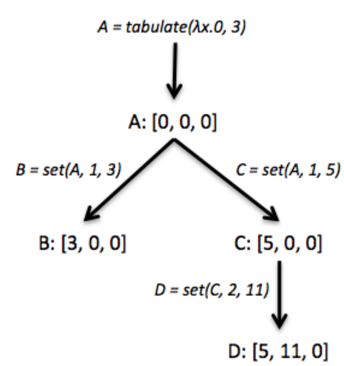

Figure 1. Example usage of functional arrays

After all the operations performed in figure 1, A and C are *interior* arrays and B and D are *leaf* arrays. Calling get and set on leaf arrays costs $O(1)$ but calling get and set on interior arrays is more expensive. In the cost dynamics, we thread a store that keeps track of whether an array is an interior or leaf array.

We also give and prove fork join cost dynamics for our arrays. The dynamics for arrays used concurrently are non trivial because one thread might call get on a leaf array while another thread calls set. To deal with this, we compute the work of each fork separately, and then charge additional work at the join point if an array is used in both forks.

In our implementation of a functional array, we keep an array of values (multiple functional arrays might point to the same array). For each index of the array, we also keep a history of values that were previously at that index. To access an element in an interior array, we need to search through the history of values.

We give a wait-free concurrent implementation that uses careful synchronization. We prove linearizability of the implementation and use that to prove that our arrays work correctly when used concurrently. Finally, we give a provable implementation for our cost dynamics.

2. Dynamics

We use a standard applicative-order language defined as follows:

$$e = x \mid c \mid \lambda x.e \mid e_1 e_2 \mid e_1 || e_2 \mid \text{if } e_1 \text{ then } e_2 \text{ else } e_3$$

L contains the usual arithmetic types, such as the natural numbers, and numerical operations such as sums and products. Our dynamics are given by the judgement $e \Downarrow v$ which means that expression e evaluates to v. Dynamics for function application and fork-join are given as examples.

$$\frac{e_1 \Downarrow v_1 \quad e_2 \Downarrow v_2}{e_1 || e_2 \Downarrow (v_1, v_2)} \text{ (fork-join)}$$

$$\frac{f \Downarrow \lambda x.e \quad y \Downarrow v}{f\ y \Downarrow [v/x]e} \text{ (function-app)}$$

The language also includes 3 functions for working with arrays: tabulate, get, and set. The dynamics for tabulate is given below:

$$\frac{f(1) \Downarrow v_1, ..., f(n) \Downarrow v_n}{\text{tabulate } f\ n \Downarrow [1 \mapsto v_1, ..., n \mapsto v_n]}$$

$get(A, i)$ returns the value at the ith index of A. $set(A, i, v)$ returns a new array where the value at index i is v and the value at all other indices is the same as in A. The evaluation of a simple program in our language is shown below:

$(\lambda A.\ get(A, 2) + get(A, 3))set(tabulate(\lambda i.\ i, 5), 2, 10)$
$(\lambda A.\ get(A, 2) + get(A, 3))set([1, 2, 3, 4, 5], 2, 10)$
$(\lambda A.\ get(A, 2) + get(A, 3))[1, 10, 3, 4, 5]$
$get([1, 10, 3, 4, 5], 2) + get([1, 10, 3, 4, 5], 3)$
$10 + 3$
13

3. Cost Dynamics

The dynamics given in the previous section are functional. However, the costs of get and set depend on whether the array is a leaf array or an internal array. To deal with this, we thread a store through the cost dynamics to keep track of whether each array is a leaf or internal array.

The cost dynamics of fork-join pose an additional challenge. An array can be used concurrently by multiple threads and the costs could depend on the order in which the instructions are interleaved. Figure 2 shows 3 ways a leaf array A might be used. In particular, one side of the fork might call get on A while the other side calls set (see the center panel). Calls to get on the left branch would be $O(1)$ if and only if they execute before the call to set on the right branch.

We cannot make any assumptions about how the instructions in different branches of a fork are interleaved, so we give the worst case time complexity over all possible interleavings. We first compute the work of each fork separately. Then for arrays that are used in both forks, we add additional work at the join point. To compute the additional work at the join point, we keep track of the number of gets that cost $O(1)$ on each side of the fork. If the array was modified on the other side of the fork, then the modification might have happened before the get, so we need to charge additional work for the get.

Our cost dynamics are defined by the following judgement, where δ is the store, e is the expression, δ' is the new store, v is

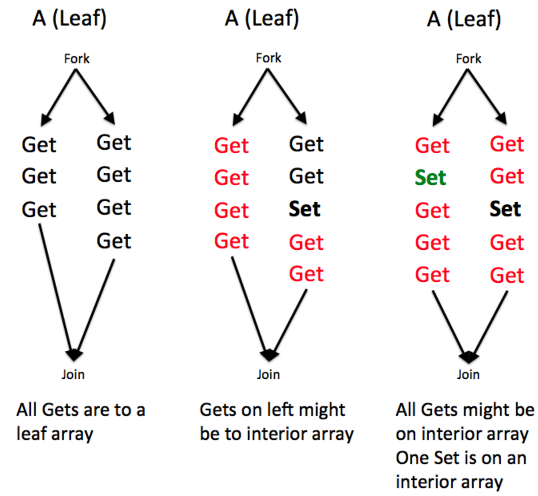

Figure 2. Different ways an array might be used in a fork join

the value returned by evaluating e, w is the work, and s is the span.

$$\delta; e \Downarrow \delta'; v; w; s$$

We give the cost dynamics for expressions in the language: (insert cost dynamics for standard functions later)

Arrays are represented by (l, V) where l is a label that uniquely identifies the array, and V contains the values in the array. $V[i]$ is the i^{th} value in the array. If the set function has been called on an array, the array is an interior array (represented by $-$), otherwise the array is a leaf array (represented by $+$).

Array functions have different costs depending on whether the array is a leaf or interior array. Let $C(l)$ be the number of get operations applied on array $A = (l, V)$ that had $O(1)$ work. We thread a store δ through each operation. δ maps array labels l to $(+/-, c)$, where $c = C(l)$. Let $f(n)$ be the work of calling get on an interior array and $g(n)$ be the work of calling set on an interior array.

The cost dynamics for the functions on a functional array of size n are:

$$\frac{l = \text{new label} \quad f(1) \Downarrow v_1; w_1; s_1; ... f(n) \Downarrow v_n; w_n; s_n}{\delta; \text{tabulate } f\ n \Downarrow \delta[l \mapsto (+, 0)]; (l, \bigcup_{i=1}^{n}[i \mapsto v_i]); 1 + \sum w_i; 1 + \max s_i}$$

$$\frac{A = (l, V) \quad \delta[l \mapsto (+, c)] \quad l' = \text{new label}}{\delta; \text{set } A\ i\ v \Downarrow \delta[l \mapsto (-, c), l' \mapsto (+, 0)]; (l', V[i \mapsto v]); 1; 1} \text{ (set-leaf)}$$

$$\frac{A = (l, V) \quad \delta[l \mapsto (-, c)] \quad l' = \text{new label}}{\delta; \text{set } A\ i\ v \Downarrow \delta[l' \mapsto (+, 0)]; (l', V[i \mapsto v]); g(n); 1} \text{ (set-interior)}$$

$$\frac{A = (l, V) \quad \delta[l \mapsto (+, c)]}{\delta; \text{get } A\ i \Downarrow \delta[l \mapsto (+, c + 1)]; V[i]; 1; 1} \text{ (get-leaf)}$$

$$\frac{A = (l, V) \quad \delta[l \mapsto (-, c)]}{\delta; \text{get } A\ i \Downarrow \delta; V[i]; f(n); 1} \text{ (get-interior)}$$

Set creates a new label and array and modifies the store to indicate that the newly created array is a leaf array, and the array set was called on has become an interior array. The work for set is higher for interior arrays. Get returns the value at the desired index, if the array is a leaf then the operation costs $O(1)$ but we increment the counter of $O(1)$ gets in the store.

The fork-join cost semantics are the most interesting. Suppose we have an expression $e_L || e_R$ and

$$\delta, e_L \Downarrow \delta_L; v_L; w_L, s_L$$

$$\delta, e_R \Downarrow \delta_R; v_R; w_R, s_R$$

We assume that newly created labels in δ_L and δ_R do not conflict. Consider an array $A = (l, V)$ with $\delta[l] = (s, c)$, $\delta_L[l] = (s_L, c + c_L)$, $\delta_R[l] = (s_R, c + c_R)$. When multiplying two signs or multiplying a sign with an integer, consider + to be 1 and - to be 0.

Combine describes how to combine the values in the store on both sides of the fork-join. The array is a leaf array iff it is a leaf array on both sides of the fork-join. $O(1)$ gets on one side of the fork remain $O(1)$ iff there were no calls to set on the other side of the fork.

$$combine((s, c), (s_L, c + c_L), (s_R, c + c_R)) = (s_L s_R, c + s_R c_L + s_L c_R)$$

Extrawork gives the additional cost incurred if an array was modified on either side of a fork-join. If the array was an interior array before the fork-join, then all functions incurred their maximal cost and there is no additional cost. Otherwise, $O(1)$ gets on one side of the fork become $O(f(n))$ iff the other side of the fork called set. If both sides of the fork called set, then one of the sets came first and is $O(1)$, the subsequent set is $O(g(n))$.

$$extrawork((s, c), (s_L, c + c_L), (s_R, c + c_R)) = s \cdot ((\neg s_R) c_L f(n) + (\neg s_L) c_R f(n) + (\neg s_R)(\neg s_L) g(n))$$

$$\delta' = (\delta_L / \delta) \cup (\delta_R / \delta) \cup \bigcup_{l \in L(\delta_L) \cap L(\delta_R)} [l \mapsto combine(\delta[l], \delta_L[l], \delta_R[l])]$$

$$w' = \sum_{l \in L(\delta_1) \cap L(\delta_2)} extrawork(\delta[l], \delta_L[l], \delta_R[l])$$

Then, the fork-join cost semantics are:

$$\delta; e_1 || e_2 \Downarrow \delta'; (v_1, v_2); 1 + w_1 + w_2 + w'; 1 + \max(s_1, s_2)$$

4. Approach

An array A is NEW if the set function has not been called with A as an argument, otherwise A is OLD. While the dynamics of our arrays are functional, the costs are different for OLD and NEW arrays.

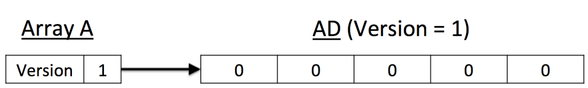

Figure 3. New functional array filled with 0s

Suppose that A is a NEW functional array (see figure 3). A has a version number V, and a pointer to an ArrayData object AD. AD keeps a regular (mutable) array of values, which corresponds to the values in A. AD has a version number, which is the same as A's (V) to indicate that A is NEW. For each element of the array in AD, keep a log of values that used to be at that index. The logs are initially empty.

Get on NEW arrays: to get the i^{th} element in A, simply access array[i] in AD.

Set on NEW arrays: Suppose that $A[i]$ is v_{old} and we want to do $B = set(A, i, v_{new})$ (see figure 4). Add an entry (V, v_{old}) to the log at index i, expressing that $A[i]$ was v_{old} at version V. Increment the version of AD to $V + 1$. Set $array[i]$ in AD to v_{new}. Create a new functional array B, with version $V + 1$, which points to AD. Notice that both A and B point to AD. However, the version of A is V and the version of AD is $V + 1$, which indicates that A is OLD. The version of B is $V + 1$ which indicates that B is NEW.

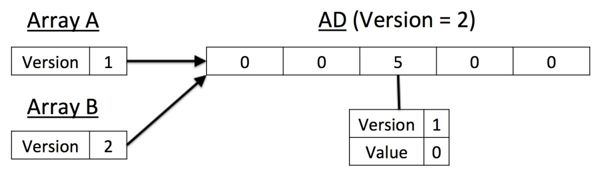

Figure 4. $B = set(A, 3, 5)$ changes the array in AD and adds a log entry.

Get on OLD array: Suppose we do $get(A, i)$ where A is OLD and has version V. We binary search the log at index i to find what value was stored in the array at version V.

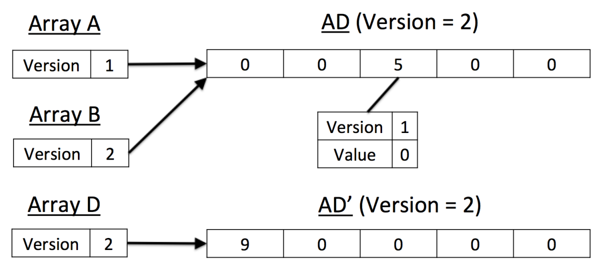

Figure 5. $D = set(A, 1, 9)$ copies AD because A is OLD

Suppose we do $D = set(A, i, v)$ where A is old (see figure 5). Create a new ArrayData object AD' (with a new array). Copy the values from the array in AD to the array in AD', and then set $array[i]$ to v in AD'.

The logs are stored in a doubling array which doubles in size when full. This guarantees amortized $O(1)$ insertion of a log entry, and allows us to binary search with $O(\log m)$ work, where m is the number of log entries.

Suppose that the size of the array is n. Every n times an ArrayData object is modified, we create a new ArrayData object, and copy the values over to the new ArrayData object. This ensures that $m \leq n$ i.e. we don't have too many log entries in an ArrayData object. Copying is amortized $O(1)$.

It follows that in the RAM model the work of set and get is $O(1)$ for a NEW array. Get in an OLD array involves a binary search and has work $O(\log n)$. Set in an OLD array involves copying the array, and has work $O(n)$.

5. Concurrent Implementation

We implement the functional array functions using an imperative target language.

5.1 Concurrency Model

We assume a sequential consistency model of concurrency. That is, when analyzing the effects of a program, we assume that at each time step exactly one pending instruction is executed. Consider the execution of a program P. We label the time steps in the execution from 1 to t.

Definition 5.1. Consider an object A. Suppose that the program invokes functions M_1, M_2, ..., M_n on A. Consider arbitrary i, and suppose that the instructions in M_i were executed at times s_1, s_2, ..., s_m. We say that A is *linearizable* if the effect of the program is as if M_i executed atomically at some time s_j, and this holds for all M_i.

Definition 5.2. Suppose that first instruction in function call f was executed at time s_1 and the last instruction at time e_1. Suppose that first instruction in function call g was executed at time s_2 and the last instruction at time e_2. We say the function calls do not *overlap* if the intervals $[s_1, e_1]$ and $[s_2, e_2]$ do not overlap.

Our programs have access to the link load store conditional (LLSC) function. LLSC takes an (address, old value, new value) tuple as argument. LLSC first checks that the value at address is the same as old value, if it is different then LLSC returns false. Otherwise LLSC atomically stores new value at address and returns true, however if the value at address was modified between the load and store, LLSC returns false.

5.2 Push Arrays

We use an auxiliary data structure, *PushArrays*, to store log entries. PushArrays are initially empty and support 3 functions. Suppose we have a PushArray A. $A.push(e)$ inserts entry e, $A.size()$ returns the number of entries inserted, $A.get(i)$ returns the i^{th} entry inserted. Note that get is only defined between indices 0 and $A.size() - 1$. All operations are amortized $O(1)$.

PushArrays can be used semi-concurrently. At most one thread can execute $push$ at any time, but multiple threads can call $size$ and get. In a PushArray A, $A.size$ stores the number of entries and $A.data$ references an array that stores the actual entries. Pseudocode for $A.push(e)$ is shown below.

```
if (isFull(A.data)) {
  newData = new Array(A.data.capacity * 2);
  copyValues(from = A.data, to = newData);
  A.data = newData;
}
A.data[A.size] = e;
A.size += 1;
```

Theorem 5.1. *Given that different calls to push (on the same PushArray) do not overlap, PushArrays are linearizable.*

Proof. The linearization point of $push$ is $A.size += 1$, and the linearization point of the other functions is their single instruction. In push, expanding the capacity of $A.data$ when it is full does not interfere with accessing $A.data$ because we copy values to the new array before pointing $A.data$ to the new array. The effects of adding e are only observable after we increase $size$, since, by the specification of PushArrays, the result of calling $A.get(i)$ where $i \geq A.size()$ is not defined. □

5.3 Functional Array Implementation

Suppose we have a FunctionalArray A. A points to an ArrayData object which is referenced by $A.data$. The (imperative) array in $A.data$ is referenced by $A.data.values$, and the log entries corresponding to the i^{th} index are stored in a PushArray $A.data.undo\_lists[i]$. We first explain the implementation of $A.set(pos, val)$.

We use a link load store conditional to ensure that only 1 thread can modify an ArrayData object at any time. If m threads try to modify $A.data$, then $m - 1$ threads will fail the LLSC (returning false). Instead of modifying $A.data$, these threads will create a new ArrayData object, and will copy over values from $A.data$, as shown below.

```
FunctionalArray newA;
newA.version = A.version + 1;
if (!LLSC(&A.data.version, A.version,
        newA.version) OR
    A.version % A.size == 0) {
  // Create a new ArrayData object AD
  // Forall i = 1:n, AD.values[i] = get(A, i)
  AD.values[pos] = val;
  newA.data = AD;
  return newA;
}
```

At most one of the threads trying to modify the ArrayData object will get past the LLSC, will insert a log entry and then modify $A.data.values$. The ordering of inserting a log entry and modifying $A.data.values$ matters in the proof of get. The pseudocode for this step is shown below.

```
newA.data = A.data;
old_value = newA.data.values[pos];
undo_list = newA.data.undo_lists[pos];
undo_list.push((A.version, old_value));
newA.data.values[pos] = val;
return newA;
```

Next we explain the implementation of $A.get(pos)$.

```
guess_val = A.data.values[pos];
if (A.version == A.data.version) {
  return guess_val;
}

undo_list = this.data.undo_lists[pos];
if (undo_list.size() == 0) {
  return guess_val;
}

upper = undo_list.size() - 1;
if (undo_list[upper].version < A.version) {
  return guess_val;
}
```

The last step is to binary search $undo\_list$, between the indices 0 and $upper$ (inclusive of 0 and upper). We find the log entry L with the smallest version X such that $A.version \leq X$. We return $L.value$.

The implementation of tabulate is straightforward - just allocate a new array and fill the array with the required values.

5.4 Proof of Correctness

Theorem 5.2. *Functional Arrays are linearizable.*

Proof. The linearization point of get is when we compare if A.version and AD.version are the same. The linearization point of set is the LLSC. Any instruction in tabulate can be its linearization point - we omit discussion of tabulate because the details are uninteresting.

First, we prove the linearization point of get. (fill this out)

Next, we prove the linearization point of set. (fill this out) □

Linearizability allows us to assume that array operations happen atomically. Note that all non array operations in our source language are executed in a single time step, so any well formed program is linearizable. In fact, since linearizability composes, we can show that more complicated languages that use our arrays are also linearizable.

In a previous section, we gave the evaluational (big step) dynamics of functional arrays. Alternatively, we could give the nondeterministic structural (little step) dynamics which describes the steps in the execution of a program in the source language. At each fork-join, if both sides of the fork can take a step, we could take a step on either side.

Theorem 5.3. *The non-deterministic structural dynamics and the evaluational dynamics produce the same result in the source language.*

Proof. This is a standard implicit parallelism type theorem, for example see Chapter 39 in (Harper 2012), so we omit the details. □

Now, to show that our implementation correctly implements the source language, it suffices to show that any valid execution of our implementation produces the same result as a little step transition sequence in the source language (since all transition sequences produce the same value as the evaluational dynamics).

Theorem 5.4. *For any program P in the source language, executing our implementation I of P produces the same result as some transition sequence of P. Note that I is a program in the target language.*

Proof. We prove the claim by induction. We focus on the part of the proof dealing with arrays and give a sketch of the rest.

Our inductive hypothesis is that after t time steps in an arbitrary execution of I, there exists a transition sequence S s.t.:

1. I becomes I' after this particular execution, S takes P to P'
2. I' and P' are equivalent in the following sense. Any non array value is identical in I' and P'. For any abstract array A in P' there exists a corresponding representation A_I s.t. the following holds. Consider arbitrary index i and suppose that $A[i]$ has value val. Suppose that A_I has version V and is pointing to an ArrayData object AD. If there exists some version in $AD.logs[i]$ which is greater than V then there exists a log entry (v', val) where v' is the smallest version in $AD.logs[i]$ that is $\geq V$. If no version in $AD.logs[i]$ is greater than V, then $AD.values[i]$ is val.

The base case holds since P and I are initially identical. For the inductive step, when I' takes a step, we take the corresponding step in P to get a new transition sequence S'. We know that such a step exists because I' and P' are equivalent. If the step is not an array function then the IH holds since the step will be executed in the same way in the source and target languages.

Suppose that the next step is tabulate. By the IH, tabulate is called with the same arguments n and f in the source language and the implementation. In the source language, tabulate returns an array A with $A[i] = f(i)$ for all $1 \leq i \leq n$. In the implementation, tabulate returns an array that points to a new ArrayData object AD with empty logs and $AD.values[i] = f(i)$ for all $1 \leq i \leq n$. So the inductive step holds.

Suppose that the next step is get. By the IH, get is called on equivalent arrays and the same index i in the source language and implementation. It is easy to trace through the implementation of get and see that get returns the same value in the source language and implementation.

Suppose that the next step is set. Suppose that in the target language we execute $set(A_I, i, v)$. Then by IH, in the source language the corresponding step is $set(A, i, v)$ where A and A_I are equivalent. Denote $set(A, i, v)$ by B and $set(A_I, i, v)$ by B_I. We need to show that B and B_I are equivalent, and all arrays that were previously equivalent are still equivalent.

Case 1: we create a new ArrayData object AD in set. In this case, for all $j \neq i$ we set $AD.values[j]$ to $get(A_I, j)$ which by the IH is the same as $A[j]$. Since we only modified index i, $j \neq i \Rightarrow A[j] = B[j]$. Also, we set $AD.values[i]$ to v, so $AD.values[i] = B[i]$. Since all logs are empty, this means that B and B_I are equivalent. Since we create a new ArrayData object, we do not modify any other arrays, so arrays that were previously equivalent are still equivalent.

Case 2: we modify the ArrayData object AD that A_I points to. This means that $A.version = AD.version$. We can show that all log entries in AD have version $< AD.version$, and are in increasing order. Then the IH implies that before modifying AD, $AD.values[j] = A[j]$ for all indices j. So after setting $AD.values[i]$ to v, $AD.values[j] = B[j]$ for all j, which means B_I and B are equivalent.

Consider arbitrary abstract array C in the source language and corresponding representation C_I in the implementation with $C_I \neq B_I$. If C_I did not point to AD then C_I remains equivalent to C. Regardless, we note that for all indices $j \neq i$, C_I remains the same and is equivalent to C. At index i, if the value at $C[i]$ was previously stored in a log entry, then C_I and C remain equivalent since log entries are added in increasing version numbers. If the value at $C[i]$ was stored in $AD[i]$ then we add a log entry at index i so C_I and C remain equivalent. □

6. Machine Model

Our target language is a Java-like language augmented with tabulate, fork-join, lambda functions, and link load store conditional (LLSC). Fork-join spawns two threads, one thread executing each sub-expression. Link load store conditional (LLSC) takes an (address, old value, new value) tuple as argument. LLSC first checks that the value at address is the same as old value, if it is different then LLSC returns false. Otherwise LLSC atomically stores new value at address and returns true, however if the value at address was modified between the load and store, LLSC returns false.

Our target language is run on a P processor machine. In a single time step the machine takes at most P runnable instructions and executes them. The effect of executing the ($\leq P$) instructions in a time step is the same as some (arbitrary) sequential ordering of the instructions. We assume that the machine does not reorder instructions in the target language. In reality, languages like Java have very relaxed consistency models, and we deal with this in our real implementation using memory fences.

The link load store conditional architecture deserves special mention. Call an LLSC operation pending if the load operation has completed but the store operation has not. Multiple threads can load the value at a particular address and compare it with old value in a single time step. Only a single store operation to a particular address can be executed at a time step. However, after the store operation, in each time step the bus arbiter notifies all pending LLSC operations to the same memory address that they have failed. This assumption is reasonable because all LLSC instructions are typically processed by the bus arbiter, so the bus arbiter can keep track of pending LLSC instructions to each memory address.

References

T.-R. Chuang. Fully persistent arrays for efficient incremental updates and voluminous reads. In *Symposium Proceedings on 4th European Symposium on Programming*, ESOP'92, pages 110–129, London, UK, UK, 1992. Springer-Verlag. ISBN 0-387-55253-7. URL `http://dl.acm.org/citation.cfm?id=145055.145076`.

P. R. Harper. *Practical Foundations for Programming Languages*. Cambridge University Press, New York, NY, USA, 2012. ISBN 1107029570, 9781107029576.

P. Hudak. A semantic model of reference counting and its abstraction (detailed summary). In *Proceedings of the 1986 ACM Conference on LISP and Functional Programming*, LFP '86, pages 351–363, New York, NY, USA, 1986. ACM. ISBN 0-89791-200-4. doi: 10.1145/319838.319876. URL `http://doi.acm.org/10.1145/319838.319876`.

M. E. O'Neill. *Version Stamps for Functional Arrays and Determinacy Checking: Two Applications of Ordered Lists for Advanced Programming Languages*. PhD thesis, Simon Fraser University, 2000.

J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science*, LICS '02, pages 55–74, Washington, DC, USA, 2002. IEEE Computer Society. ISBN 0-7695-1483-9. URL `http://dl.acm.org/citation.cfm?id=645683.664578`.

P. Wadler. Linear types can change the world! In *PROGRAMMING CONCEPTS AND METHODS*. North, 1990.

P. Wadler. Monads for functional programming. In *Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques-Tutorial Text*, pages 24–52, London, UK, UK, 1995. Springer-Verlag. ISBN 3-540-59451-5. URL `http://dl.acm.org/citation.cfm?id=647698.734146`.