Extensible Sparse Functional Arrays with Circuit Parallelism

John T. O'Donnell
School of Computing Science, University of Glasgow
john.odonnell@glasgow.ac.uk

Abstract

A longstanding open question in algorithms and data structures is the time and space complexity of pure functional arrays. Imperative arrays provide update and lookup operations that require constant time in the RAM theoretical model, but it is conjectured that there does not exist a RAM algorithm that achieves the same complexity for functional arrays, unless restrictions are placed on the operations. The main result of this paper is an algorithm that does achieve optimal unit time and space complexity for update and lookup on functional arrays. This algorithm does not run on a RAM, but instead it exploits the massive parallelism inherent in digital circuits. The algorithm also provides unit time operations that support storage management, as well as sparse and extensible arrays. The main idea behind the algorithm is to replace a RAM memory by a tree circuit that is more powerful than the RAM yet has the same asymptotic complexity in time (gate delays) and size (number of components). The algorithm uses an array representation that allows elements to be shared between many arrays with only a small constant factor penalty in space and time. This system exemplifies circuit parallelism, which exploits very large numbers of transistors per chip in order to speed up key algorithms. Extensible Sparse Functional Arrays (ESFA) can be used with both functional and imperative programming languages. The system comprises a set of algorithms and a circuit specification, and it has been implemented on a GPGPU with good performance.

Categories and Subject Descriptors E.1 [*Data Structures*]: Arrays; D.1.1 [*Programming Techniques*]: Applicative (Functional) programming; B.6.1 [*Logic Design*]: Parallel circuits; B.3.2 [*Memory Structures*]: Associative memories; F.1.2 [*Modes of Computation*]: Parallelism and concurrency

General Terms Algorithms, Languages, Performance

Keywords functional array, sparse array, extensible array, functional programming, circuit parallelism

1. Introduction

A longstanding problem in algorithms and data structures is the complexity of operations on pure functional arrays. This question has both theoretical and practical significance, because arrays are fundamental to much software and the complexity of their operations affects the complexity of many algorithms.

An imperative array supports two operations: fetching an array element $a[i]$ and modifying an array element $a[i] := exp$. Both operations require $O(1)$ time, according to common cost models, and they do not require any space beyond the memory originally allocated for the array. After an element of an imperative array is modified the previous content of that element is destroyed.

A pure functional program defines new values but does not perform side effects, such as modifying an existing value. Thus a pure functional array allows new arrays to be constructed but does not allow old ones to be changed. When a functional array is updated, the result is a new array that differs from the old array at one index, but the old array is still accessible. This paper generalises functional arrays to handle sparse and extensible operations as well, and the data structure is called ESFA (extensible sparse functional arrays).

Imperative arrays are trivial to implement, as they are based on basic machine instructions and addressing modes. In contrast, straightforward implementations of functional arrays are inefficient in space or time, and the most efficient implementations are complex and still asymptotically slower than imperative arrays.

Since unrestricted access to functional arrays is inefficient, there has been relatively little exploration of algorithms that rely on them. Nevertheless, functional arrays remain interesting in their own right, and they do have practical applications (Sec. 8).

Functional arrays are not simply arrays used in a functional language. Imperative and functional arrays are distinct data structures that support different operations. Both data structures can be used in both imperative and functional languages. The choice between imperative and functional arrays should be based on the needs of the algorithm using them, not on the programming language used to express the algorithm.

This paper discusses the relationship between imperative and functional arrays, and conjectures (in Sec. 6.1) that that it is impossible to implement functional arrays with $O(1)$ access time using a Random Access Machine (a theoretical model of computation). However, the main result of the paper is an algorithm that does indeed implement functional arrays with the same time complexity as imperative arrays, using the same cost models. This result appears to contradict the conjecture—but the new algorithm runs on a different model of computation which we call *circuit parallelism*. The essential idea is that the RAM model—as well as conventional computers—makes inefficient use of the digital logic components that make up the memory. By redesigning the hardware as well as the software we can sometimes beat lower bounds on algorithmic complexity without increasing the complexity in circuit elements or delay time.

Parts of the ESFA algorithm were presented in 1993 [10]. This paper briefly reviews the earlier work, and goes on to make several new contributions: operations and algorithms for memory management, correctness proofs of key parts of the system, specification of the hardware as well as software, an analysis of the complexity, specification of the operations to support sparse and extensible ar-

rays, a discussion of applications, and executable implementations of the algorithm and a simulator for the hardware.

Every one of the ESFA operations takes a small fixed number of clock cycles. No iteration is used in any of the operations; the execution time is constant, and does not depend on the past history of updates or deletions that led to the current state of the machine. No restrictions are placed on the operations that can be performed in order to achieve these perfectly tight time bounds. If an array is deleted, the deletion takes a small fixed number of clock cycles (like *all* ESFA operations), and it immediately identifies or recovers *every* memory cell that is inaccessible, in constant time. This constant time performance does not come at the cost of increased hardware complexity: the clock speed of the hardware has the same time complexity as the clock for an ordinary addressable memory, and the hardware complexity in terms of number of logic gates and flip flops is also the same. The complexity of ESFA operations is discussed in more depth in Sec. 6.

The algorithm presented here uses *circuit parallelism*. This approach originated in associative processors [6] and active data structures [1]; other examples include priority queues [2], systems with chunks of memory organised as trees [15], smart memories for multicore processors [5], associative searching [9]. Circuit parallelism is the target platform for compilation of a declarative committed-choice rule language [16]. The idea is to bring the parallelism inherent in digital circuits to bear directly on the computations required by an algorithm, rather than organising the circuit into conventional processors. In circuit parallelism, the computation is melded into the memory at the level of individual words, allowing algorithms that perform a parallel *computation within every word in the memory*; it differs from data parallelism, where an operation performs a computation on every word of a data structure (rather than every word in the machine). Current hardware trends will make this approach increasingly productive, as the number of transistors per chip continues to increase.

The algorithms presented here are fine grain and massively parallel, and they require suitable hardware in order to be usable. They do not run efficiently on a sequential computer that lacks a hardware accelerator. The fastest platform for ESFA is a direct VLSI implementation of the underlying parallel circuit, but an FPGA or GPU chip could also be used (see Sec. 5.2).

The ESFA system has been implemented and tested in two ways. First, it is implemented using a digital circuit that is specified and simulated using the Hydra hardware description language [12]. The design has not been fabricated as a physical chip, but the Hydra specification is precise down to the level of flip flops and logic gates (it is "synthesizable"), and the simulation is accurate in clock cycles and also in gate delays within a cycle. Second, the system is implemented as a program, written in C+CUDA, that runs on a general purpose GPU [4]. The GPU implementation gives good performance, and there is extremely low variation in execution time of the operations, making it especially valuable for real time applications and a good platform for research. An FPGA implementation would be faster, but the GPU program is far more portable and can run on many consumer computers.

The software is available on the web [13], including the circuit simulator, the GPU program, a random test data generator, and sample test data files. The programs have been tested using a combination of small hand-written test cases, an SECD machine interpreter that uses ESFA for the environment, and large scale randomly-generated test data. The circuit simulator has run sequences of 725,000 operations, and the GPU has run tens of millions of operations, without error.

Section 2 introduces the operations on extensible sparse functional arrays. Sec. 3 gives an overview of the algorithm, which contains three layers. Sec. 4 presents the algorithms that comprise the upper level of the implementation, and Sec. 5 describes the parallel circuit that constitutes the lower level. Related work on arrays in functional languages is described in Sec. 6, and the time and space complexity of ESFA are analyzed. Sec. 7 discusses the implementation and performance of the system Sec. 8 describes some practical applications, and Sec. 9 concludes.

$empty$	$:: AName$
$update$	$:: AName \to Idx \to Val \to Maybe\ AName$
$lookup$	$:: AName \to Idx \to Maybe\ Val$
$mindef$, $maxdef$	$:: AName \to Maybe\ (Idx, Val)$
$nextdef$, $prevdef$	$:: AName \to Idx \to Maybe\ (Idx, Val)$
$delete$	$:: AName \to StESFA\ ()$
$killZombie$	$:: StESFA\ (Maybe\ Val)$

Table 1. ESFA operations

2. Operations on ESF arrays

An array consists of a set of elements, each with an index of type Idx and a value of type a which is determined by the user program. All elements of all arrays in the ESFM must have the same type a, which can be an algebraic data type defined by the user. As far as the ESFM is concerned, an array element is just a word of bits; typing issues are up to the programming language used for the main program.

Table 1 lists the operations supported by the ESFM. The $update$ operation function takes an existing array, index, and value, and returns a new array with the given value at that index. It returns $Nothing$ if the ESFM is full, or if the "existing" array does not actually exist. Thus $update\ a\ i\ x$ corresponds roughly to the imperative notation `a[i] := x`, but there is a crucial difference: $update$ creates a new array without modifying the old one.

Array elements are accessed using $lookup$, which takes an array and an index and returns the array element defined at that index, if one exists. Thus $lookup\ a\ i$ corresponds to the imperative notation `a[i]`.

The relationship between $empty$, $update$, and $lookup$ is specified by two laws. For simplicity, these are written using the non-monadic versions and error conditions are ignored.

Law 1. (Empty array)

$$lookup\ empty\ i = Nothing$$

Law 2. (Nonempty array)

$$\begin{aligned} &lookup\ (update\ a\ (j, v))\ i \\ &\quad \mid i \equiv j = Just\ v \\ &\quad \mid i \not\equiv j = lookup\ a\ j \end{aligned}$$

The first law says that an empty array contains no elements, and the second one says that an $update$ to an existing array a gives an array that is identical to a except at index i, where it has the new value v.

The programmer does not declare the size of an array or allocate space for it, and an array of undefined elements cannot be allocated all at once, as in Fortran. Instead, elements are added one by one using $update$, and the system allocates space automatically as needed. The programmer cannot modify an existing array; $update$ creates a completely new array and leaves the old array unchanged. Every array is built incrementally through a sequence of updates, starting ultimately from $empty$. For example, $a3$ is constructed using three updates:

$a1 = update\ empty\ 1\ 101$
$a2 = update\ a1\ 2\ 102$
$a3 = update\ a2\ 3\ 103$

The values of the resulting arrays are:

$$\begin{array}{lcl} a1 & = & \{1 \rightsquigarrow 101\} \\ a2 & = & \{1 \rightsquigarrow 101, 2 \rightsquigarrow 102\} \\ a3 & = & \{1 \rightsquigarrow 101, 2 \rightsquigarrow 102, 3 \rightsquigarrow 103\} \end{array}$$

This is an example of "single threaded" construction, where there is a simple linear chain of updates. However, $update$ is not restricted just to extending the most recently created array. *Any* array can be updated at any time, allowing for a tree-structured set of relationships among arrays. Consider the following definitions, with the previous definitions still in scope:

$a4 = update\ a2\ 4\ 104$
$a5 = update\ a1\ 5\ 105$

The values of $a1$ and $a2$ have never changed, and the results are:

$$\begin{array}{lcl} a4 & = & \{1 \rightsquigarrow 101, 2 \rightsquigarrow 102, 4 \rightsquigarrow 104\} \\ a5 & = & \{1 \rightsquigarrow 101, 5 \rightsquigarrow 105\} \end{array}$$

If an update gives a new value to an index that has already been defined, the old value is shadowed: it does not appear in the new array but is still present in the old one.

$a6 = update\ a4\ 2\ 202$

$$a6 \quad = \quad \{1 \rightsquigarrow 101, 2 \rightsquigarrow 202, 4 \rightsquigarrow 104\}$$

These examples illustrate the chief characteristic of functional arrays: $update$ produces a new array, but does not change the old one. They also show why implementation of the operations in $O(1)$ time and space is difficult: it is essential to share each element among any number of arrays without using linked data structures to find them.

An extensible array is one whose minimum and maximum bound can be changed at any time. The $update$ operation gives extensibility for free, as there is no restriction on the value of the index that is provided. For example, we could define $a7 = update\ 1000000\ 7$. Naturally this can leave many indices where no element is defined.

In a sparse array, many elements may have a default value (often 0). The representation should use memory only for the non-default elements, saving space. Furthermore, there needs to be a way to traverse the non-default elements without having to iterate over all the indices. ESF arrays support sparse traversal using the $mindef$, $maxdef$, $nextdef$ and $prevdef$ operations.

Suppose we wish to iterate over all the non-default elements of an array, from the lowest to highest index. The starting point of the iteration is determined using $mindef$ to find the lowest index with a non-default value, and the iteration repeatedly applies $nextdef$ to the current index to find the next one.

It is essential to be able to reclaim memory when ESFAs are deleted. There are several ways to do this, depending on the nature of the host program and programming language. The $delete$ operation asserts that a given array will not be accessed again, so its space should be reclaimed.

There are several other ESFA operations not discussed in this paper. Some of them measure space utilisation and extract diagnostic information. Others exploit the hardware's capabilities to perform more general associative searching; for example, you can provide a value and find (in constant time) an index, if any, where that value exists in a given array.

3. Overview of the system

This section introduces the key issues and ideas behind the ESFA algorithm—the "big picture"—and the details are given in subsequent sections.

A good introduction to the implementation problem is to consider two naive algorithms.

- We implement $update\ \ a\ \ i\ \ x$ by copying the array a into a fresh region of contiguous words of memory, and storing the value of x at location i. All future lookups will take $O(1)$ time, as normally expected for imperative arrays. Unfortunately, the $update$ requires both time and space of $O(n)$ where n is the size of a. The equivalent of `for i := 0 to n-1; x[i] := x[i+q];` would require $O(n^2)$ time and allocate $O(n^2)$ words of memory.
- Since the previous approach is disastrous, we could focus on making $update$ efficient. Define an algebraic data type $\mathbf{data}\ Arr\ a = Empty \mid Update\ Arr\ Idx\ a$. Now the $update$ operation is simply an application of the constructor $Update$, and it builds a tree of nodes. Lookup requires traversing the tree, from an $Update$ node back toward the root, until the index is found. This approach makes $update$ take $O(1)$ time and space, and $lookup$ requires time that depends on the shape of the tree and the location of the index—potentially $O(n)$ where n is the size of the array.

Clearly the first approach—wholesale copying—is hopeless. The second approach invites attempts to restructure the tree to reduce the height and thereby make $lookup$ faster, but that will never produce $O(1)$ time. Many sophisticated techniques have been developed for making functional data structures more efficient [14]. Generally, a tree structure stored in the heap will give access time proportional to the height of the tree.

The result in this paper is an algorithm with the same time and circuit complexity as a conventional RAM memory—where *every* ESFA operation takes O(1) time—with no variation in execution time—and there are no restrictions on the usage of the operations. The naive algorithms above suggest that sharing and searching are the fundamental difficulties, and we now consider these issues more deeply.

Functional arrays allow an arbitrary amount of sharing. If both $a1$ and $a2$ are defined as updates to a, then every element of a is shared among all three arrays. We begin by observing that an element must be represented in the machine only once, no matter how many arrays it belongs to. To achieve constant time access, however, we cannot follow chains of pointers that link the elements together.

An imperative array is really an address that can be used to calculate the address of any element; we can think of an array as "knowing" where its elements are. That won't work for ESFA, because of the sharing. So consider a major change of perspective: consider placing each element at an arbitrary location in the memory, and storing with each element the set of all arrays to which it belongs. This is called the *inclusion set* (or *iset*) of the element. An array is no longer represented by an address, but just by a natural number called a *code*.

For this approach to be workable, we need to be able to represent every inclusion set in a fixed amount of space, and also to determine whether an arbitrary code c belongs to an inclusion set. In general, it is impossible to represent sets so efficiently, but all inclusion sets are constructed by $update$. We can represent the inclusion set for an element e by a pair $(low, high)$ such that $c \in iset\ e$ if and only if $low \leqslant c \leqslant high$.

Suppose we are updating an array with code c. It turns out that the code of the resulting array has to be $c + 1$, and that code is

likely already to be in use. That means that the codes that are larger than c must be incremented, and also the $(low, high)$ fields in many of the cells need to be adjusted. The cost would be prohibitive on a sequential computer, but by adding some logic gates to the flip flops comprising a cell, this can be done in parallel in every cell in unit time.

Since the array codes will be changing frequently, how can we represent a pointer to an array? Each array has a stable *array name*, which will never change, and the system contains a mapping from array names to codes. When an array code needs to be changed, the mapping is also adjusted so that every array name still refers to the same set of elements, even if the code has changed.

Every update or delete operations can cause wholesale changes to the representation, affecting an number of memory cells (possibly even all of them). The parallelism of digital circuits is needed to support this, so ESF arrays are implemented using "hardware/software codesign", with three layers:

1. The lowest layer (Sec. 5.1) is a parallel machine: a "smart memory" consisting of a tree-structured digital circuit (or abstract machine) that implements a basic machine operation called *sweep*. The circuit is synchronous, with a fixed speed clock. A sweep requires one clock cycle. The clock speed is determined primarily by the gate delay in combinational logic within the circuit. This gate delay is $O(k)$ where k is the depth of the tree. If the circuit is built with a balanced tree, then $k = \log n$ where there are n memory cells, allowing for n array elements to be stored. A conventional memory also has a logarithmic time gate delay, because it needs to decode the memory address.
2. The middle layer (Sec. 5.3) uses the sweep operation to implement a family of combinators that execute in one clock cycle. These include parallel maps, folds, and scans. It is also possible to program directly with $sweep$, which is more powerful than the standard scans.
3. The upper layer (Sec. 4) is software that defines the ESFA operations using the middle layer combinators. Each operation is defined as "straight line code" with no iteration of any kind. In other words, every ESFA operation takes a fixed number of clock cycles, typically 2 or 3.

ESF arrays are stored in a special circuit, the ESF machine (ESFM), a "smart memory" that uses circuit parallelism to provide a richer set of operations than a RAM (random access memory). The ESFM is an attached device, analogous to a floating point unit or a graphics processing unit, and is separate from the host processor and memory. The user program runs in a host computer with an ordinary processor and memory, and its only access to the ESFM is through the operations described below. Some of the operations change the state of the ESFM, while others only fetch data without changing the state.

There are two Haskell interfaces to the operations: a pure functional one and a monadic one that makes the ESFM machine state explicit, whose details are not discussed here. It is not necessary to use the monadic interface in Haskell, and the pure functional one may be more convenient for some users' applications. The system contains several features that support an interface with a host system, and some of these require the monadic interface; for details see the web page [13].

Each ESF array has a unique array name with type $AName$, which is represented internally as an Int. An AName is a stable pointer [8] providing a permanent reference to the array. It is an opaque reference: two names can be compared for equality, but the main program cannot perform any other useful operations on array names, such as address arithmetic. The system always has an array $empty :: AName$ that contains no elements. This is unique and indestructible: the programmer can neither create nor delete it.

Field name	Type	Purpose
cellID	Nat	array name (if mapDef)
select	Bool	control associative operation
mark	Bool	control associative operation
mapDef	Bool	cellID is name of an array
code	Nat	code of array (if mapDef)
eltDef	Bool	cell contains element
zombie	Bool	elt exists but is garbage
nfy	Bool	notify (not delete) when inaccessible
low, high	Nat	inclusion set
rank	Nat	distance from empty
ind	Word	array elt index
val	Word	array elt value

Table 2. Fields of a machine cell

4. Algorithms for the operations

In an imperative array, each array element must be stored at a specific address that is calculated from the address of the array and the index of the element. Thus the position of a word in the memory determines the position of the contents of the word within an array.

The essential difficulty in implementing functional arrays is that we need to maintain full sharing of array elements—otherwise the cost in both space and time would be prohibitive—yet this means that there is no simple relationship between the location of an element in the memory and its location within all the arrays that contain it.

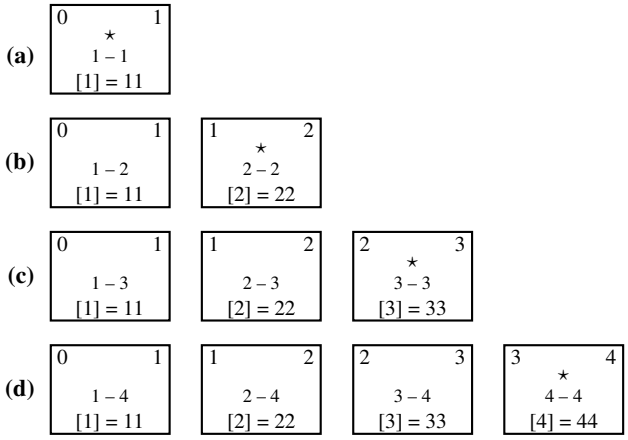
Consequently, it is necessary to abandon the idea of using machine addresses to encode indices. Instead, a technique is developed that decorates each memory location that contains an array element with a representation of the set of arrays that contain the element. This is called the *inclusion set* of the element. In general, arbitrary sets cannot be represented in a small fixed amount of space. However, the inclusion sets that appear in the ESF memory are not arbitrary: they satisfy some structural properties that are forced by the fact that the memory state must be the result of a sequence of $update$ operations.

4.1 Representation

Each array is identified by a unique natural number called an array code. An inclusion set can be represented by a pair of indices, low and high, such that an array is in the inclusion set if and only if its code lies between low and high. Every location in the array memory contains several fields, including an element value, and index value, an inclusion set, and a few more.

Suppose we are evaluating $lookup\ a\ i$. The algorithm begins by determining for each word in the memory whether a is a member of the inclusion set for that word. This calculation is performed in parallel in every word, and the result is used to set a mask bit in each word where the inclusion set contains a; this defines a set of words that might contain the right result; call these words the "candidates". We then compare the value of i with the index field in the candidates, and clear the mask where there is no match. It is possible that several candidates remain; this happens if an array has been calculated with several updates to the same index. The final step uses the rank fields of the remaining candidates: the rank is the number of updates, starting from empty, that created the element, so the candidate with the highest rank is the correct result.

There are no loops in the $lookup$ algorithm. The same is true for $update$, $delete$, and all the other operations: each requires a fixed number of steps. Each step performs a lot of work—a small



$$
\begin{array}{lcl}
a1(0) & = & \text{update empty } 1\ 11 \\
 & = & 0 \Rightarrow\ 1 \rightsquigarrow 11 \\
a2(1) & = & \text{update } a1\ 2\ 22 \\
 & = & 1 \Rightarrow 1 \rightsquigarrow 11,\ 2 \rightsquigarrow 22 \\
a3(2) & = & \text{update } a1\ 3\ 33 \\
 & = & 2 \Rightarrow 1 \rightsquigarrow 11,\ 2 \rightsquigarrow 22,\ 3 \rightsquigarrow 33 \\
a4(3) & = & \text{update } a3\ 4\ 44 \\
 & = & 3 \Rightarrow 1 \rightsquigarrow 11,\ 2 \rightsquigarrow 22,\ 3 \rightsquigarrow 33,\ 4 \rightsquigarrow 44
\end{array}
$$

Figure 1. Building an array by successive updates. The intermediate arrays are retained. Each box shows one ESFM cell. Consider the bottom right cell in the figure. The notation "a4(3)" means that array a4 has AName=3. The AName appears in the top left corner, and the corresponding code (if any) is in the top right corner. Thus the bottom right box shows that AName=3 currently has code=4. The * indicates that select=True in this cell. The lo–hi interval 4–4 is below the flags, and the bottom row shows the index and value. **(a)** The cell number 0 (upper left corner) is allocated; the cell number is the AName. The empty array is being updated, and its code is 0, so the code of a1 is 1 (upper right corner). **(b)** Cell 1 is allocated so a2 has AName=1 and code=2 (1 + code of a1). The inclusion set of the new element is 2–2 which contains {a2}; the inclusion set of the element in cell 0 is modified, so it now contains codes 1 and 2, representing {a1, a2}. **(c,d)** Each update allocates a new cell for the value of the element, and modifies the inclusion sets of the other elements.

computation in each location—but these can all be performed in parallel.

Each memory cell in the machine contains several fields and flags (Table 2).

- The *cellId* field is an Int giving the address of the cell; this is used as the name of an array if the *update* that creates the array stores its element into this cell. As long as this array still exists, the *mapDef* Boolean is True and the code for the array is held in the *code* field. If the array is deleted, then *mapDef* will be set to *False*, but if the element is shared by other arrays it will not be deleted.
- There are four Boolean flags that are used by some of the operations to keep track of specific cells. Their usage is shown in the algorithms for the operations; *select* is set in the active cell (e.g. the cell that will hold a new element during *update*); *mark* identifies a set of cells (e.g. candidates for *lookup*), *eltDef* indicates that the cell contains an index/value, *notify* indicates that if the element is deleted the host should be notified; *zombie* indicates that the cell contains garbage and can be deleted after the host is notified.
- The inclusion set is represented by low and high (sometimes abbreviated lo and hi).
- The element contains an index and value.

4.2 Implementation of update

Most of the operations, including *lookup* and the sparse array operations, are relatively straightforward. However, the *update* algorithm is delicate and non-obvious. An update not only modifies the data structures related to the array being updated; it can potentially modify every array and array element in the entire memory. This section proves the correctness of *update* using a traditional informal mathematical style.

The update and delete operations are largely similar; they all do local computations that involve arithmetic on integers that can be performed in parallel on all the memory locations.

When an update is performed, the inclusion sets and array codes need to be adjusted. Many of the memory locations will need to modify one or more of their fields, but again these operations can all be performed in parallel. It is also necessary to modify some of the existing array codes (think of a memory allocation scheme that moves data and has to note the changed addresses of objects that have moved). Since the array codes change frequently, they are useful only inside the ESF array memory. Consequently the system maintains an association table between stable array names—which never change—and the rapidly changing codes. This association table is also maintained in the array memory. It requires two more fields in each location, and all of the operations on name/code translation are parallel.

Many of the operations need to find the code corresponding to a given array name. This is performed by a function $encode :: AName \rightarrow Maybe\ (Nat, Nat)$. Encode uses an associative search to check the *mapDef* field of the cell whose *cellId* matches the array name. If there is no match then encode returns Nothing; otherwise it fetches the *code* and *rank* fields. The entire operation requires a parallel fold (one clock cycle).

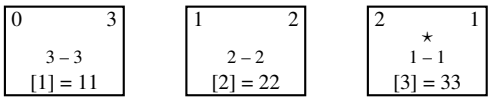
Another basic function allocates a free cell, by performing an associative search for a cell where $mapDef \equiv False$. (It is straightforward to prove that such a cell cannot contain an element.) This operation also requires a parallel fold, and takes one clock cycle.

If an *update* cannot find the code for an array, or if it cannot allocate a cell because the memory is full, an error indication is returned. The algorithms below omit the error handling, but the full algorithms on the web page [13] check and handle all error conditions.

Algorithm $update\ a\ i\ v$

1. The code for the array with $AName = a$, and the rank of the corresponding element, are found by an associative search (1 clock cycle): $(c, r) \leftarrow encode\ a$.
2. A free cell is allocated (1 cycle); this sets the *select* flag in the allocated cell. The name of the new array is the $nm = cellId$ of the new cell, and the code is $c' = c + 1$.
3. If the allocation failed (no free cell) Nothing is returned; otherwise each cell is updated in parallel (1 clock cycle) as follows:

   if *select*
   then -- this is the allocated cell
   $mapDef' = True; code' = c';$
   $lo := c'; hi := c'; rank := r + 1;$
   $ind := i; val := v$
   else -- update existing cells



a1(0) = update empty 1 11
a2(1) = update empty 2 22
a3(2) = update empty 3 33

Figure 2. Creating arrays with no shared elements

$$
\begin{aligned}
lo' &= \textbf{if } c < lo \textbf{ then } lo + 1 \textbf{ else } lo \\
hi' &= \textbf{if } c \leqslant hi \textbf{ then } hi + 1 \textbf{ else } hi \\
code' &= \textbf{if } c < code \\
&\qquad \textbf{then } code + 1 \\
&\qquad \textbf{else } code
\end{aligned}
$$

4. The AName of the new array, $Just\ nm$, is returned.

4.3 Example of update

Several examples are helpful in following how the algorithm works. We begin with a single threaded array, and then examine what happens with more complicated update patterns. Fig. 1 shows a simple sequence of updates. *Each element is represented in exactly one cell but it is included in every array which should contain that element.* Because of the single-threaded sequence of updates, none of the array codes have changed, but more complex update sequences require changes to some existing codes (see Figs. 2 and 3).

Fig. 2 shows the representation of separate arrays, where the elements are not shared. Fig. 3 shows what happens when one of the arrays in the previous figure is updated, resulting in some elements that are shared and some that are not.

4.4 Correctness of update

An update can fail due to two possible conditions: (1) if the array being updated does not exist (i.e. its name does not appear in the array name/code map); (2) if the memory is full, so a new cell cannot be allocated. These conditions could be handled by any suitable technique, such as using the Maybe or Error monad, or by raising an exception.

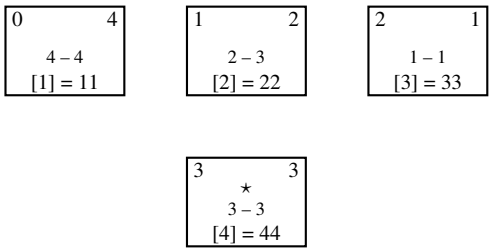
Definition of update. Consider $update\ a\ i\ v$ where array a has code c. Then the code of the new array is $c' = c + 1$. All existing cells in the entire system memory are adjusted as follows. For a cell that contains an element, the new values of the cell are:

- $lo' = \textbf{if } c < lo \textbf{ then } lo + 1 \textbf{ else } lo$
- $hi' = \textbf{if } c \leqslant hi \textbf{ then } hi + 1 \textbf{ else } hi$
- $ac' = \textbf{if } c < ac \textbf{ then } ac + 1 \textbf{ else } ac$

Invariants. Immediate corollaries of the definition are that the name and code of *empty* are always 0, and for every array element, $lo \leqslant hi$.

Correctness conditions. It is necessary to show that the new array and new element are correct, and also that all existing arrays and elements remain correct.

Theorem: Correctness of update. The $update$ algorithm locates an empty cell. The cell id (address of the cell) is used as the name of the new array (this is guaranteed to be a name that is not currently in use). The code of the new array is $c + 1$, where c is the code of the original array that is being updated, and is stored in the cell.



a4(3) = update a2 4 44

Figure 3. Both shared and unshared elements.

There are two ways to prove that all the arrays are represented correctly after an update: we could consider each array and show that it has exactly the right elements, or we could consider each element and show that it is contained in exactly the right arrays. This proof takes the second approach, focusing on the elements.

The following notation is useful: e refers to the array element contained in a cell before the $update$, and e' refers to the element after the operation. Similarly, we use names like lo to refer to the value of the field in a cell before the $update$, and lo' (with a prime) to refer to the new value after the operation. Note that c, the code of the array being updated, is a global value.

The AName/code map is adjusted: for any array with code d, the array's new code $d' = \textbf{if } d > c \textbf{ then } d + 1 \textbf{ else } d$.

First we consider cells in the memory that contain existing data, and show that after the $update$ their contents remain correct. Then we show that the new element that is allocated is correct.

There are five classes of cell. Consider an arbitrary cell; if it is not empty, then it contains an element e with lo and hi.

- **Empty:** the cell contains no element, and is not allocated by the update, so it remains empty and does not affect the representation of any array.
- **No change:** $lo' = lo$ and $hi' = hi$, so by definition $lo \leqslant c$ and $hi < c$. For $ac \in iset\ e$, $lo \leqslant ac \leqslant hi < c$, so $ac < c$ hence $ac' = ac$. Therefore $ac' \in iset\ e'$ if and only if $ac \in iset\ e$.
- **Shift:** $lo' = lo + 1$ and $hi' = hi + 1$, so $c < lo$ and $c \leqslant hi$. For $ac \in iset\ e$, $c < lo \leqslant ac \leqslant hi$ so $ac' = ac + 1$. Since $lo \leqslant ac \leqslant hi$ if and only if $lo + 1 \leqslant ac + 1 \leqslant hi + 1$, we have $ac' \in iset\ e'$ if and only if $ac \in iset\ e$.
- **Expand:** $lo' = lo$ and $hi' = hi + 1$, so by definition $lo \leqslant c$ and $c \leqslant hi$. Unlike the other cases, the interval between lo and hi has expanded. Then either $ac' = ac$ or $ac' = ac + 1$. In either case $lo \leqslant ac' \leqslant hi'$. Therefore each array that was in $iset\ e$ remains in $iset\ e'$. Furthermore, the code for the new array enters the inclusion set: this element belonged to the array that was being updated, so it also belongs to the result of the update.
- **Allocate:** the cell is empty, and is selected by $update$ for allocating the new element. Then $lo' = c' = hi'$, so $iset\ e' = \{c'\}$.

Thus it has been shown that each element in the memory is included in the right set of arrays. The dual of this is that the new array a' contains the new element as well as the elements of the original array a, and all other arrays remain unchanged. This completes the proof.

4.5 Implementation of lookup

Algorithm $lookup\ a\ i$

1. The code for the array with $AName = a$ is found by an associative search (1 clock cycle): $c \leftarrow encode\ a$.
2. The *mark* flag is set in each cell where $ind \equiv i \wedge lo \leqslant c \wedge c \leqslant hi$. This is a parallel map taking 1 clock cycle. The marked cells include the elements of the array, but they also include shadowed elements. For example, if $a = update\ (update\ a1\ 5\ 50)\ 5\ 60$, both elements $5 \rightsquigarrow 50$ and $5 \rightsquigarrow 60$ are marked.
3. The marked element with the highest rank fetched, using a parallel fold (1 clock cycle); this is the correct value at the index i.

The *lookup* algorithm is straightforward, using standard associative searching. It uses the fact that if $a2 = update\ a1\ i\ v$ then the rank of (i, v) element is greater than the ranks of the elements of $a1$.

4.6 Implementation of delete

Algorithm $delete\ a$

1. The code for the array with $AName = a$ is found by an associative search (1 clock cycle): $c \leftarrow encode\ a$.
2. Each cell is updated in parallel (1 clock cycle) as follows:

$$
\begin{array}{l}
mapDef' = mapDef \wedge cellId \not\equiv a \\
code' = \quad \textbf{if } c < code \\
\qquad\qquad \textbf{then } code - 1 \\
\qquad\qquad \textbf{else } code \\
lo' = \textbf{if } c < lo \textbf{ then } lo - 1 \textbf{ else } lo \\
hi' = \textbf{if } c \leqslant hi \textbf{ then } hi - 1 \textbf{ else } hi \\
eltDef' = eltDef \wedge (keep \vee delLater) \\
zombie' = zombie \vee (eltDef \wedge delLater) \\
\quad \textbf{where } keep = eltDef \wedge lo' \leqslant hi' \\
\qquad\qquad delLater = eltDef \wedge \neg\ keep \wedge requestNotify
\end{array}
$$

4.7 Example of delete

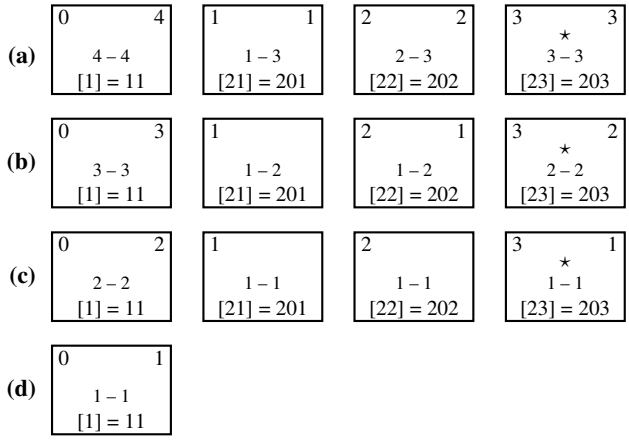
Fig. 4 shows how a sequence of deletions can leave all the array elements undisturbed until a final deletion removes the last bit of sharing; as a result many cells are reclaimed simultaneously.

4.8 Correctness of delete

We are deleting an array with $AName = a$ and $code = c$. The array name/code map is adjusted: for any array with code d, that array's new code $d' = \textbf{if } c < d \textbf{ then } d - 1 \textbf{ else } d$.

Consider an arbitrary cell. We show that if it contains an element that is identified as inaccessible then its inclusion set is empty, and otherwise its inclusion set is unchanged.

- **Empty:** the cell contains no element, and $delete$ does not change that status.
- **No change:** $lo' = lo$ and $hi' = hi$. Then $lo \leqslant c$ and $hi < c$. Consider a code d **in** $iset\ e$, so $lo \leqslant d \leqslant hi$. Since $d \leqslant hi < c$, then $d' = d$, so $lo' \leqslant d' \leqslant hi'$, hence d' **in** $iset\ e'$. Similarly, if d' **in** $iset\ e'$, then d **in** $iset\ e$. Hence $iset\ e' = iset\ e$.
- **Shift:** $lo' = lo - 1$ and $hi' = hi - 1$. Then $c < lo$ and $c \leqslant hi$. If d **in** $iset\ e$, then $lo \leqslant d \leqslant hi$ and $c < d$ so $d' = d - 1$. Since $lo \leqslant d \leqslant hi$ iff $lo - 1 \leqslant d - 1 \leqslant hi - 1$, we have $iset\ e = iset\ e'$.
- **Shrink:** $lo' = lo$ and $hi' = hi - 1$, and $lo' \leqslant hi'$ so the element e is not marked as inaccessible. Then $lo \leqslant c$ and $c \leqslant hi$. The interval from lo to hi has shrunk, so we need to verify that the only element removed from the inclusion set is the array with code c; i.e. we need to show that $iset\ e = iset\ e' \cup \{c\}$.



(a) Several arrays are created; b1–b3 share elements
a1(0) = update empty 1 11
$1 \rightsquigarrow 11$
b1(1) = update empty 21 201
$21 \rightsquigarrow 201$
b2(2) = update b1 22 202
$21 \rightsquigarrow 201, \quad 22 \rightsquigarrow 202$
b3(3) = update b2 23 203
$21 \rightsquigarrow 201, \quad 22 \rightsquigarrow 202, \quad 23 \rightsquigarrow 203$
(b) b1 is deleted but its element remains
delete b1
(c) b2 is deleted; all elements remain
delete b2
(d) Deleting b3 removes sharing; all b elements are gone
delete b3

Figure 4. Deleting an array leaves shared elements intact.

Suppose d **in** $iset\ e$, so $lo \leqslant d \leqslant hi$. Suppose that $d = c$; then this is the code of the array being deleted, and its AName no longer exists and will not give a code of d. Suppose that $d \not\equiv c$. If $d < c$ then $d' = d$ and $lo' \leqslant d' \leqslant hi'$. If $c < d$ then $d' = d - 1$ and again $lo' \leqslant d' \leqslant hi'$. The conclusion is that an array with code d is removed from the inclusion set if it is the array being deleted, and otherwise it remains in the inclusion set.

- **Inaccessible:** $lo' = lo$ and $hi' = hi - 1$, and $hi' < lo'$ so the element e is marked as inaccessible (either reclaimed immediately, or marked as a zombie for reclamation later). So $lo \leqslant c \leqslant hi$. Since $hi' < lo'$, then $lo = c = hi$. Suppose d **in** $iset\ e$, then $lo = d = hi$, so $d = c$. Since d is the only element of $iset\ e$, and d is being deleted, this element e' is inaccessible.

4.9 Implementation of sparse operations

To find the element of an ESFA with the minimum (or maximum) index, the elements are first marked, just as with *lookup*. A parallel scan then locates the extremal index in one cycle. Similar techniques are used to locate the next (previous) element in an array.

5. Parallel circuit

The hardware is an extremely fine grain parallel architecture; for best performance, there should be one processing element for each

location in the array memory. The processing elements are not full scale processors; they need only the ability to perform comparisons and increments on natural numbers, and a few bit level operations. A processing element would contain on the order of 100 bits of memory and a few hundred logic gates. The algorithm is suited for implementation on a fine grain parallel system, such as a digital circuit, an FPGA program, or a GPU.

5.1 Circuit structure and sweep

The foundation of the algorithm is a digital memory circuit with a tree structure. This structure is similar to the organisation of a standard random access memory; see Section 6 for a comparison. Each leaf in the tree contains a state of type a, and the nodes provide combinational logic functions but have no state. The state of the entire machine is modeled by the type $TreeMachine$.

$$\begin{aligned}&\textbf{data}\ TreeMachine\ a\\ &\quad = TMcell\ a\\ &\quad \mid TMnode\ (TreeMachine\ a)\ (TreeMachine\ a)\end{aligned}$$

Each leaf cell is a circuit with one input and one output port, which are connected to the parent node. A port is a set of signals (wire) organised as a tuple of fields. A cell is a state machine with type $s \to d \to (s, u)$ where s is the type of the state, u is the type of the output which goes up the tree, and d is the type of the input which comes down from the tree.

Each node is a pure function implemented by combinational logic gates. It receives three inputs coming down from the parent and up from the two subtrees, and produces three outputs. The type of the node function is $d \to u \to u \to (u, d, d)$. This is a completely general type which allows for circularities, but the specific node functions used in this paper are noncircular, so the entire tree circuit is synchronous. This has several consequences: there is no state or memory in the tree apart from the state in the leaf cells, there is a fixed depth of combinational logic between flip flops, and there is a fixed number of logic gates that must settle down in each clock cycle.

At a clock tick, each flip flop updates its state by storing the value of its input signal. As the logic gates settle down, information flows from the cells up the tree to the root, which also receives an input from the main processor, and information then flows back down the trees to the cells, preparing for the next clock tick. This general operation is called a *sweep*.

$$\begin{aligned}&sweep\\ &\quad :: (s \to d \to (s, u)) && \text{-- cell function}\\ &\quad \to (d \to u \to u \to (u, d, d)) && \text{-- node function}\\ &\quad \to d && \text{-- root input}\\ &\quad \to TreeMachine\ s && \text{-- state of tree circuit}\\ &\quad \to (TreeMachine\ s, u) && \text{-- (new state, root output)}\end{aligned}$$

A sweep causes each cell to apply a logic function to its state to produce an output to send up; later this function also calculates the new state using the current state and the incoming down message. These two logic functions are combined in the single function cf.

$$\begin{aligned}&sweep\ cf\ nf\ x\ (TMcell\ s) = (TMcell\ s', y)\\ &\quad \textbf{where}\ (s', y) = cf\ s\ x\end{aligned}$$

Each node uses its nf function to calculate the value a' to send up, and the the values p' and q' to send down to the left and right subtrees. Again, all these calculations are combined in one function nf. An alternative way to define a general tree circuit is to separate the cell and node functions into separate up and down functions, and then to define separate $upsweep$ and $dnsweep$ functions.

$$\begin{aligned}&sweep\ cf\ nf\ a\ (TMnode\ x\ y) = (TMnode\ x'\ y', a')\\ &\quad \textbf{where}\ (a', p', q') = nf\ a\ p\ q\\ &\qquad (x', p) = sweep\ cf\ nf\ p'\ x\\ &\qquad (y', q) = sweep\ cf\ nf\ q'\ y\end{aligned}$$

5.2 Parallel circuit platforms

Any implementation of $sweep$ can be used as a foundation for the higher levels of the ESFA system. However, $sweep$ requires computation in every leaf cell and every tree node during every clock cycle.

The tree machine is a *synchronous* circuit. This means that each flip flop updates its state simultaneously at every clock tick. During the time between clock ticks, the logic gates settle down to produce stable outputs that do not change again for the rest of the clock cycle (the period between ticks). The combinational logic gates (stateless devices, such as the logical $and2$ gate, that calculate pure functions) can be partitioned into equivalence classes based on the longest gate delay on any of their inputs. After d gate delays, all the logic gates in the classes below d are stable, the gates in classes above d do not have stable inputs so their outputs are immaterial, and all the gates in class d perform a useful calculation in parallel. The clock period must be slow enough for all the logic gates to settle down to a stable value; this is the maximal gate delay through the circuit, with an additional safety factor.

An alternative platform is an FPGA (field programmable gate array). This is a hardware device consisting of an array of small scale units (general logic functions, small memories, and the like) along with a programmable interconnection network that can be used to connect the small units to form a digital circuit. The advantage of an FPGA is that it is an off the shelf component; the disadvantage is that it is considerably slower and less dense than a custom VLSI chip.

In practice, the hardest part of using an FPGA is often interfacing it with a host computer. This is not at all standardised, and it requires a combination of communication software, device driver software, and interfacing hardware that is laid out on the FPGA chip itself. Some FPGAs are tightly coupled with a processor on the same chip, but others require slower I/O connections.

An FPGA platform might benefit by tiling. The idea is that k (where is a small number, such as 2) tree machine cells are mapped onto one physical cell. Each sweep operation would require k cycles to allow each physical cell to emulate its virtual ones. This increases the size of the memory without requiring the extra tree nodes and logic functions, with a commensurate slowdown.

GPGPU (general purpose graphical processing units) are a form of fine grain multicore processor targeted at a restricted form of data parallelism. Originally these devices were intended for graphics algorithms, but they have found increasing usage for more general data parallel computation, including circuit simulation. Current GPGPU chips have many small processor cores (on the order of 1000), and offer good performance for regular applications.

A GPU implementation of ESFA has been developed, and it gives good performance with extremely small variance in execution time. This is discussed in more detail in Section 7.3

It would also be possible to run the circuit simulator on a conventional multicore system, but that is unlikely to give adequate performance. A GPU, with a much larger number of processing elements, is a much better host for ESFA.

5.3 Parallel map, fold and scan

The second level in the system uses the general tree circuit to implement three key combinators: $tmap$, $tfold$, and $tscanl$. There are other algorithms that use further combinators supported by the general circuit, and it is sometimes more convenient to program directly with sweep, but this set of combinators suffices for ESF arrays.

The $tmap$ combinator maps a function f over the cells, which define a sequence similar to a list. The nodes have no role other than to fan out the signals. If map were the only operation to be performed, there would be no need for a tree, but in a synchronous circuit the clock speed is determined by the longest gate delay path. This occurs in the tree when it is instantiated to perform $scanl$, so the system would not run faster if $tmap$ were implemented without the tree sweep. Furthermore, it is common for the function f being mapped to be specified as a partial application; in the digital circuit this requires information to be broadcast to the cells, and the tree does this efficiently.

$$
\begin{aligned}
&tmap :: (s \to s) \to TreeMachine\ s \to TreeMachine\ s \\
&tmap\ f\ t = fst\ (sweep\ cf\ nf\ ()\ t) \\
&\quad \textbf{where}\ cf\ s\ \_ = (f\ s, ()) \\
&\qquad nf\ a\ p\ q = ((), (), ())
\end{aligned}
$$

It is straightforward to configure the tree circuit to perform a fold over a sequence of values that are extracted from the leaf cells of the tree. The $tfold1$ applies f to each cell to obtain a message that moves up the tree; messages are combined using g, which is usually chosen to be associative (although $tfold1$ is defined unambiguously even if g is not associative).

$$
\begin{aligned}
&tfold1 \\
&\quad :: (s \to a) && \text{-- f: get msg from cell} \\
&\quad \to (a \to a \to a) && \text{-- g: combine msgs} \\
&\quad \to TreeMachine\ s && \text{-- initial state} \\
&\quad \to a && \text{-- result}
\end{aligned}
$$

$$
\begin{aligned}
&tfold1\ f\ g\ t = snd\ (sweep\ cf\ nf\ ()\ t) \\
&\quad \textbf{where}\ cf\ s\ \_ = (s, f\ s) \\
&\qquad nf\ a\ p\ q = (g\ p\ q, (), ())
\end{aligned}
$$

The tree circuit can also be programmed to perform a parallel scan. Again, there is a function f to obtain a message from a cell, and an update function g to update the cell with an incoming message. The h function is used to combine messages.

$$
\begin{aligned}
&tscanl \\
&\quad :: (s \to a) && \text{-- get singleton from cell} \\
&\quad \to (s \to a \to s) && \text{-- update cell using singleton} \\
&\quad \to (a \to a \to a) && \text{-- h: function to be folded} \\
&\quad \to a && \text{-- initial accumulator} \\
&\quad \to TreeMachine\ s && \text{-- initial state} \\
&\quad \to (TreeMachine\ s, a) && \text{-- (new state, final foldl result)}
\end{aligned}
$$

The $tscanl$ defines a communication pattern that begins with the cells, transmits information up the tree, and then transmits further information back down the tree. If h is associative, then $tscanl$ performs a scan-from-left over the cells of the tree; see [11] for a correctness proof.

$$
\begin{aligned}
&tscanl\ f\ g\ h\ a\ t = sweep\ cf\ nf\ a\ t \\
&\quad \textbf{where}\ cf\ s\ a = (g\ s\ a, f\ s) \\
&\qquad nf\ a\ p\ q = (h\ p\ q, a, h\ a\ p)
\end{aligned}
$$

The combinators defined above specify the state explicitly, but it is easier to use monadic versions that hide the state. In order to make I/O convenient for experimenting with the simulator, the type $StateT\ (TreeMachine\ s)\ IO\ a$ is used.

5.4 Key operations

The most straightforward steps in the ESFA algorithms are parallel maps (every cell does the same computation, perhaps using a global value that is sent down the tree), and parallel folds (for example, determining whether a cell exists with a certain property). A few operations use a parallel scan to perform a calculation that uses information from all the cells, and identifies a specific cell to receive special treatment. Two typical examples are:

- $c \leftarrow encode\ (a)$ performs a parallel fold: the cell function determines whether the $cellId$ matches the array name a, and the node function transmits the result up to the root.
- $x \leftarrow newCell$ uses a parallel scan; the cell function determines whether a cell is available, and the associative function being scanned picks the first one.

6. Complexity

If it can be proved that access to a functional array is single-threaded, so that an array is never accessed again after it has been updated, then the compiler can implement a functional array update by overwriting the old array element, achieving the efficiency of imperative arrays. This approach can be supported using compiler analysis, the type system, or monads [7]. It works well for algorithms that were designed in the first place for imperative arrays, but it does not help when the flexibility of pure functional arrays is needed. This approach has been developed quite far, with efficient array accesses for parallel processors [3]. However, our concern is with general functional arrays where there is no restriction on the usage of the operations.

6.1 Functional arrays and the Random access machine

In talking about the complexity of algorithms, it is essential to be clear about what machine model we are using. The Random Access Machine is a theoretical model that corresponds closely to standard von Neumann computer architectures. The Random Access Machine has a Random Access Memory, which allows any memory location to be accessed in $O(1)$ time, regardless of the address. This differs from a Turing Machine, which takes account of locality and makes it more expensive to access distant locations.

Conjecture There does not exist an implementation of functional arrays such that every operation always takes $O(1)$ time and space on a Random Access Machine.

The conjecture is simply saying that the performance of imperative arrays cannot be attained by functional arrays, without placing restrictions on how the arrays are used.

In complexity theory, it is common to consider the costs of operations using unbounded memory size and unbounded word size. In real computer systems, there is a fixed word size k and memory size limited by 2^k. We can still analyze the complexity of an algorithm running in a bounded system; the algorithm will fail if it exceeds the bounds. In an unbounded system, many of the complexity measures have an additional factor of $\log(\log N)$ because the index size grows as the memory grows. In this section, we will consider machines of bounded size, but the essential result (unit time ESFA operations with the same circuit size and delay as for RAM) remains valid for the unbounded case as well.

6.2 Circuit parallelism

The ESFA system presented in this paper uses its massively parallel digital circuit to do exactly what the conjecture says is impossible on a RAM architecture.

The time required by a computer system to perform a computation is $k \times p$, where k is the number of clock cycles required and p is the clock period (the reciprocal of the clock speed). It is straightforward to find k. In calculating p it is essential to state clearly what cost model for the circuit is being used. It is a fallacy to say that each instruction in a RAM takes $O(1)$ time but the ESFM has a tree circuit so its clock period is $O(\log n)$, and then to use these figures to compare their speeds.

cost model	RAM	ESFM
unit time cycle	$O(1)$	$O(1)$
gate delay	$O(\log n)$	$O(\log n)$
propagation delay	$O(\sqrt{n})$	$O(\sqrt{n})$

Table 3. Comparison of cycle time for RAM and ESFM. In the gate delay model, the RAM needs a tree of multiplexers with delay $O(\log n)$ to decode addresses; the ESFM uses a tree of slightly more complex circuits to perform the folds, and the delay is also $O(\log n)$. The point is that the basic operations provided by the ESFM circuit are far more powerful than those provided by RAM, yet ESFM does not require asymptotically more hardware and does not take asymptotically more time.

Consider the number of clock cycles for $update$ and $lookup$. The ESFM performs every $update$ in a fixed number of clock cycles (3 cycles, with the current implementation), and every $lookup$ in a fixed number of cycles. Those cycle counts do not depend on the past history of operations and they do not assume any restriction on the usage of arrays. Furthermore, every one of the operations defined in Sec. 2 takes a constant number of clock cycles (and the constant is small, typically 2 or 3 cycles). Memory usage is also optimal: each $update$ allocates exactly one cell and there is never any loss of sharing. Each $delete$ identifies every cell that is inaccessible, in constant time.

Does the ESFM achieve its optimal cycle count at the expense of an asymptotically longer clock period, or at the expense of an asymptotically larger circuit? Table 3 shows that it does not, and Table 4 shows why.

The ESFM contains $O(n)$ flip flops and $O(n)$ logic gates for a system with n cells. There are many technologies for implementing a random access memory, and some do not use flip flops or logic gates, but equivalent devices are used. A RAM contains $O(n)$ "flip flop equivalents" and $O(n)$ "logic gate equivalents", so its size is asymptotically the same as the ESFM.

The clock period must be expressed according to a *cost model* that defines what aspects of the hardware are being considered, and what aspects are abstracted away. Some reasonable cost models include:

- *Unit time cycle.* Assume that each clock cycle is one unit of time. This model is commonly used in analysis of algorithms: the average case time for Quicksort is $O(n \times \log n)$ *assuming the unit time cycle model*.
- *Gate delay.* Find the critical path in the circuit and measure its gate delay, and multiply by a safety factor slightly more than 1. The RAM needs to decode the memory address, which requires a gate delay of $O(\log n)$. The ESFM has a gate delay proportional to the height of the tree, which is also $O(\log n)$. A subtle point is that $lookup$ requires a comparison of two ranks in each tree node; this would appear to require gate delay of $O(\log n \times \log n)$ with ripple comparitors or $O(\log(\log n))$ with fast comparitors. However, ESFA uses a pipelined combinational circuit that requires only $O(\log n)$ gate delay.
- *Propagation delay.* Storage elements take physical space, and accessing them requires signals to travel a distance $O(\sqrt{n})$ for a memory of size n. RAM and ESFM both have a propagation delay of $O(\sqrt{n})$.

Table 3 compares the time complexity of a clock cycle for ESFM and RAM. The choice of cost model depends on what aspects of a system seem most important. For small circuits the gate delay dominates the cycle time, and the gate delay model, which ignores the lengths of wires, is reasonable. For large circuits, the wire length dominates. The important point is that *comparisons should use the same cost model*.

	logic gate calculations	useful calculations
RAM	$O(n)$	$O(1)$
ESFM	$O(n)$	$O(n)$

Table 4. Usage of calculations in RAM and ESFM.

It is not the case that ESFM performs as well as RAM according to every cost model. For example, it is likely to require asymptotically more power per access than a RAM.

Why does ESFM achieve the same time complexity as imperative arrays, while (it is conjectured) a RAM cannot? The answer lies in the address decoder in a RAM. This is a tree of multiplexers; each level in the tree uses one bit of the address to select which subtree contains the memory word being addressed. The tree has a gate delay of $O(\log n)$ and it contains $O(n)$ logic gates, and all it does is to select one word based on an address *even though every flip flop and every logic gate in the tree performs a calculation*. In other words, the RAM performs $O(n)$ logic gate calculations in a clock cycle in order to perform $O(1)$ *useful* work. The ESFM also performs $O(n)$ logic gate calculations, but it uses all of these to calculate new values in every cell in the entire machine (Table 4).

7. Implementation, testing, and performance

The ESFA algorithm has been implemented in several ways. There is an executable specification in Haskell, which uses the semantic laws (Section 2) to implement the operations; there are circuit specifications at several levels of abstraction; and there is a parallel GPU implementation. The source code (simulators and parallel GPU program), documentation, and test data are available on the web [13].

7.1 Circuit specification

The system has been defined as a high level circuit specification, which models the machine at the level of communicating black box circuits, a commonly used level of abstraction in computer architecture research. There is also a synthesizable circuit specification which spans all the levels from register-transfer level to logic gates and flip flops, using the Hydra hardware description language. The circuit specifications are executable, and the system runs on the simulated circuits. Simulating a large circuit on a sequential computer is, naturally, slow, but it suffices for testing.

7.2 Testing

Testing has been done using hand-written test cases, which are necessarily small. The system has also been tested by using it to support some application programs.

The most robust testing is performed using a custom random test case generator. An earlier version of the ESFA software used QuickCheck, but this has been replaced by a flexible test generator that examines the state of the ESFA machine at each step, in order to generate high quality test data. The reason for this is that truly random test data is not very useful. For example, a randomly generated array name is likely not to exist, and a randomly generated index within an array is likely to be undefined. Therefore random operations are likely to return Nothing too often. The custom test generator analyzes the state of the machine to adjust the probabilities for the choice of operation and the operands.

There is a test generator that generates a test suite and saves it to a file; this is used for testing the GPU program described below. As the generator runs, it uses the state of the machine to select useful operations to perform (in order to avoid too many "Nothing"

k	N	updates	lookups	deletions
4	16	17,541	14,818	17,641
13	8196	18,452	14,713	16,835

Table 5. ESFA machine size and number of operations performed in a test run with 50,000 randomly generated operations with randomly generated (but meaningful) operands. The N column is the number of cells, and $k = \log N$. The experiment was run for $4 \leqslant k \leqslant 13$ but only figures for the the smallest and largest cases are shown.

results). The generator uses the semantic specification to calculate the correct-by-definition result, and it uses the circuit simulator to calculate what the machine will actually do. These results are compared. Furthermore, after every operation the software analyzes the entire state of the circuit and extracts the entire set of arrays (with all their elements), and compares this with the entire set of arrays according to the semantics. This ensures that a situation cannot arise where the machine state is wrong but a lookup that would expose the error happens not to be generated.

Testing via the simulator has been carried out for many sets of random test data, with machine sizes ranging from 16 cells to 64k cells, and with sequences of up to 100,000 operations.

7.3 GPU program

The ESFA algorithm has been implemented in C+CUDA and tested on an NVidia GeForce GTX 590 GPU, which has 512 CUDA Cores. The program is organised as a dialogue between the CPU, which issues ESFA operation requests, and the GPU kernel, which executes the requests and returns the results. The GPU kernel uses persistent shared memory to hold an array of ESF cells.

Well over 25 million ESFA operations have been executed without error on the GPU; the result of every operation was checked by comparing it with the result of an executable specification. The experiments show that the algorithm runs correctly on a parallel system. Furthermore, each experimental run was long enough to test the automatic memory management thoroughly. Table 5 shows that every test run executed far more updates than there are cells in the machine; most of these updates reused cells that had been reclaimed by delete operations.

Table 6 summarises the performance of the algorithm. The total execution time (the T column) grows slowly as a function of k, the log of the number N of cells. The execution is most efficient at the largest size. The measurements are highly repeatable with low variance. The times were measured by starting a CPU clock, running the entire test suite of 50,000 operations to completion, and then stopping the clock. The timing measurements include the overheads of communicating between the CPU and GPU, as well as the time for memory management.

Consider an ESFA machine with 8K cells. An efficient sequential algorithm (*not* a simulator) using a tree structure may require an average of 13 RAM accesses to perform an ESFA operation, with additional overhead for loop control. With fast memory hardware, this would amount to around 0.1 to 1 microsecond. However, if an array is long—say 5000 elements—and the tree is not balanced, the access would require 50 to 500 microseconds. For comparison, ESFA running on the GPU requires 300 microseconds. It is possible to rebalance the trees periodically, but that introduces further overhead, requires more storage, and increases the variance in execution time. The ESFA algorithm *always* requires a time of 300 microseconds, without any extra overheads, making it valuable for real time applications where each operation must be completed within a deadline.

k	N	T(ms)	stdv	d	$t(\mu s/op)$
4	16	5,071.1	32.3	104.0	101.4
5	32	5,374.3	30.9	99.9	107.5
6	64	5,606.1	25.9	117.1	112.1
7	128	6,447.8	26.0	123.7	128.9
8	256	7,946.3	32.7	122.8	158.9
9	512	7,994.0	14.0	65.7	159.9
10	1,024	8,426.2	25.1	101.0	168.5
11	2,048	12,144.0	35.9	125.0	242.9
12	4,096	13,424.4	33.5	106.5	268.5
13	8,192	15,037.2	44.7	183.1	300.7

Table 6. ESFA performance on GPU. Each line shows the configuration (k = tree depth, N = number of cells) and the total time T in milliseconds for a run consisting of 50,000 randomly-generated operations including a mix of updates, lookups, and deletes. For each configuration, the execution was repeated 25 times, and the result of every operation was compared with the result calculated by an executable specification. Thus $10 \times 25 \times 50,000 = 12,500,000$ ESFA operations were performed without error. The stdv column is the standard deviation of T; d = max-min of T. The t column gives the time in microseconds per ESFA operation. The system used is running CUDA version 4 on an NVidia GeForce GTX 590, with 512 CUDA cores (16 multiprocessors with 32 cores per mp), and a 1.22 GHz clock speed, with CUDA capability 2.0.

The fastest way to implement ESFA would be a custom VLSI design, and an FPGA would give an efficient emulation of the circuit. A GPU has a level of granularity that is intermediate between a multicore and an FPGA. The GPU implementation already gives excellent performance for applications where low variance of operation time is important, and it is poised to benefit from newer generations with more parallel processing cores. Another advantage of the GPU is that the code is far more portable than an FPGA would be.

8. Applications

Most algorithms have been developed for languages that provide imperative arrays. These algorithms use single-threaded access to arrays, and there is no particular advantage in replacing them with ESFA. The most promising applications for ESFA make essential use of sharing with multi-threaded access and/or sparse traversal and searching.

Functional arrays provide a flexible undo/redo facility. Suppose a program records transactions by updating an ESFA, keeping the most recent array as the current state. The program can revert to any previous state simply by accessing the corresponding state; there is no need to rebuild any data structures.

ESFA can represent the environment in a lambda calculus reducer. Each variable is represented by a unique integer, each environment is an ESFA, and the initial environment is $empty$. Obtaining the value of a variable is performed by a lookup, while extending the environment for a beta reduction requires an update:

$$\begin{aligned} &eval\ (Var\ x)\ env = lookup\ env\ x \\ &eval\ (App\ (\lambda x \rightarrow e1)\ e2)\ env = \\ &\quad eval\ e1\ (update\ env\ x\ e2) \end{aligned}$$

An SECD machine has been developed using this method, and has been used to test the ESFA system running on a simulated circuit.

Programming language implementations sometimes use complex data structures to represent the evaluation stack as well as variable environments. Sometimes this can be inefficient: a dynamically bound variable may require a list traversal to find its value,

while an ESFA could obtain the value in one operation. Constraint solving algorithms use backtracking or coroutines to explore several alternative paths, and functional arrays may prove useful for representing the constraint sets.

The GPU implementation is fast enough to support productive experimentation with algorithms using ESF arrays.

9. Conclusion

This paper conjectures that a standard computer based on the RAM memory cannot implement functional array operations in constant time, unless the program makes restricted use of the operations.

Despite this conjecture, the paper gives the design of a digital circuit implementing a smart memory and a set of algorithms that use that memory to achieve the goal: functional array lookup and update, as well as several other operations supporting extensible and sparse arrays, can all be implemented in unit time. Each operation requires a small constant number of steps, and there is no restriction on the past history of updates. A lookup or update always takes exactly the same time, regardless of how much sharing there is among all the existing arrays.

The algorithms are implemented in Haskell, and the circuit is specified using Hydra, a hardware description language embedded in Haskell. There is a parallel GPU implementation that gives good performance, and the algorithm also runs on the simulated circuit.

The implementation of ESFA relies on a fine grain data parallel host to hold the array memory. Each operation involves a small amount of calculation in every location in the entire memory. The system performs a lot of extra work—a small amount of arithmetic in every location—and then mitigates the extra work with massive parallelism—ideally, a processing element in every location.

However, there is a more insightful way to think of the algorithm. Consider a sequential program running on standard hardware, with a RAM memory. Programmers think of the RAM as just doing a little work on the word that is accessed (if they think of the RAM at all). However, a RAM is a digital circuit that actually has to perform an enormous amount of work on every access (not exactly a computation on every location, but that is a fair intuition). We think of the RAM as performing a small amount of work because most of its work is *wasted and not worth thinking about*. The ESFM does more work than the RAM, but only by a constant factor, and it uses this work to enable it to support a useful data structure more efficiently than a RAM can.

ESF arrays are not just a theoretical novelty. Although the fastest host for the ESFM would be an application specific integrated circuit (ASIC), which has not been implemented, the current GPU implementation is fast enough for many applications, and is easily portable, supporting future research in purely functional data structures.

Acknowledgments

Cordelia Hall implemented the GPU program and the SECD interpreter, and provided many valuable suggestions for the paper.

References

[1] G. R. Andrews and D. P. Dobkin. Active data structures. In *ICSE'81: Proc. 5th Int. Conf. on Software Engineering*, pages 354–362. IEEE Press, 1981.

[2] G. Bloom, G. Parmer, B. Narahari, and R. Simha. Shared hardware data structures for hard real-time systems. In *Proceedings of the tenth ACM international conference on Embedded software*, EMSOFT '12, pages 133–142, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1425-1. doi: 10.1145/2380356.2380382. URL `http://doi.acm.org/10.1145/2380356.2380382`.

[3] M. T. Chakravarty, G. Keller, S. Lee, T. L. McDonnell, and V. Grover. Accelerating Haskell array codes with multicore GPUs. In *Declarative Aspects of Multicore Programming*. ACM, January 2011.

[4] J. D. O. et. al. A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum*, 26(1):80–113, 2007.

[5] A. Firoozshahian, A. Solomatnikov, O. Shacham, Z. Asgar, S. Richardson, C. Kozyrakis, and M. Horowitz. A memory system design framework: creating smart memories. In *Proceedings of the 36th annual international symposium on Computer architecture*, ISCA '09, pages 406–417, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-526-0. doi: 10.1145/1555754.1555805. URL `http://doi.acm.org/10.1145/1555754.1555805`.

[6] C. Foster. *Content Adressable Parallel Processors*. Cengage Learning EMEA, 1976. ISBN-13: 978-0442224332.

[7] S. L. P. Jones and P. L. Wadler. Imperative functional programming. In *Proc. 20th ACM Symposium on Principles of Programming Languages (POPL)*, pages 71–84, 1993.

[8] S. P. Jones, S. Marlow, and C. Elliott. Stretching the storage manager: weak pointers and stable names in Haskell. In P. Koopman and C. Clack, editors, *11th International Workshop on Implementation of Functional Languages*, volume 1868 of *LNCS*, pages 37–58. Springer, 1999.

[9] T. R. Martinez. Smart memory architecture and methods. *Future Gener. Comput. Syst.*, 6(2):145–162, Nov. 1990. ISSN 0167-739X. doi: 10.1016/0167-739X(90)90030-H. URL `http://dx.doi.org/10.1016/0167-739X(90)90030-H`.

[10] J. O'Donnell. Data parallel implementation of Extensible Sparse Functional Arrays. In *Parallel Architectures and Languages Europe*, volume 694 of *LNCS*, pages 68–79. Springer-Verlag, 1993.

[11] J. O'Donnell. A correctness proof of parallel scan. *Parallel Processing Letters*, 4(3):329–338, September 1994.

[12] J. O'Donnell. Overview of Hydra: A concurrent language for synchronous digital circuit design. In *Proceedings 16th International Parallel & Distributed Processing Symposium*, page 234 (abstract). IEEE Computer Society, April 2002. ISBN 0-7695-1573-8. Workshop on Parallel and Distribued Scientific and Engineering Computing with Applications—PDSECA.

[13] J. T. O'Donnell. *ESF Arrays*. School of Computing Science, University of Glasgow, 2013. URL `www.dcs.gla.ac.uk/~jtod/research/esfa`.

[14] C. Okasaki. *Purely functional data structures*. Cambridge University Press, 1999.

[15] T. St. John, J. B. Dennis, and G. R. Gao. Massively parallel breadth first search using a tree-structured memory model. In *Proceedings of the 2012 International Workshop on Programming Models and Applications for Multicores and Manycores*, PMAM '12, pages 115–123, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1211-0. doi: 10.1145/2141702.2141715. URL `http://doi.acm.org/10.1145/2141702.2141715`.

[16] A. Triossi, S. Orlando, A. Raffaetà, and T. Frühwirth. Compiling chr to parallel hardware. In *Proceedings of the 14th symposium on Principles and practice of declarative programming*, PPDP '12, pages 173–184, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1522-7. doi: 10.1145/2370776.2370798. URL `http://doi.acm.org/10.1145/2370776.2370798`.