Data Parallel Implementation of Extensible Sparse Functional Arrays

John T. O'Donnell
University of Glasgow, Glasgow G12 8QQ, U.K.

Abstract. Three important generalised array structures — extensible arrays, sparse arrays and functional arrays — are slow to access unless their use is severely restricted. All three can be combined in a powerful new active data structure called 'ESF arrays'. ESF arrays may grow or shrink dynamically, they can be searched quickly, and changes to them can be rolled back in a single operation. A new data parallel algorithm implements every operation on an ESF array in a small constant time, without placing any restrictions on the use of those operations. The algorithm is suitable both for massively parallel architectures and for VLSI implementation.

1. Introduction

Massively parallel architectures make possible active data structures that are infeasible to implement on conventional sequential machines. This paper applies massive parallelism to the design of active array data structures, and makes two original contributions:

1. The definition of a new data structure, the *ESF array*, combining the characteristics of three distinct data structures: extensible arrays, sparse arrays and functional arrays.
2. A fast data parallel implementation of ESF array operations. The algorithm is suitable both for massively parallel architectures and for custom VLSI implementation. The VLSI implementation is essentially an associative memory with an enhanced address decoder tree.

This algorithm is particularly significant because it performs ESF array operations *faster than possible* on a von Neumann machine, yet its VLSI complexity (both area and propagation delay) is *identical* to the complexity of a von Neumann machine.

2. Background

A crucial property of array operations is *constant time and space access*: the time and space to access or update an arbitrary element does not depend on the size of the array or the value of the index.

Many generalisations of arrays have been proposed to support the needs of diverse algorithms. Some are straightforward, such as multidimensional arrays

and arrays of records. Here we are concerned with three important generalisations that are impossible to implement (on the von Neumann architecture) without losing the property of constant time and space access.

Extensible arrays. An extensible array does not have a fixed size; it can grow or shrink dynamically as a program runs. There is a function that returns the existing lower and upper bounds of an array, as well as an operation that changes the existing bounds, making the array larger or smaller. Extensible arrays are useful for algorithms that create or destroy information dynamically, including some database and graph algorithms.

It is very costly to implement such a dynamic data structure. If the extension is implemented by allocating a new (larger) block of memory and copying the array contents into it, then the extended array can still be accessed in constant time — but the extension itself is very slow. The extension can be performed in constant time by building a node with pointers to the previous array and the block of new elements, but then accesses become slower. During the 1960s Rosenberg developed more sophisticated algorithms that reduce the total cost of extensible arrays, although individual operations are still slowed down by the extensions.

Sparse arrays. In some applications, most of the elements of an array have a default value (typically 0). A sparse array saves space by representing only those elements with interesting, or non-default, values. More importantly, a sparse array provides a function *forward*(i) that gives the index of the next interesting value beyond i, skipping over all the intervening default values. This makes traversal faster. For example, iterating from the lower to upper bound

for $i := lb$ **to** ub **do**
  if $a[i] \neq default$ **then** *Process* (i);

requires $ub-lb+1$ iterations, but processing only the interesting elements of a sparse array requires fewer iterations:

$i := lb$;
while $i \leq ub$ **do**
  begin *Process* (i); $i :=$ *forward* (i); **end**

Sparse arrays are usually represented with linked lists, where each node contains a non-default value and a pointer to the next one. This enables algorithms to traverse the interesting values quickly, but random access becomes slow.

An ideal implementation would perform all accesses — random as well as default-skipping ones — in constant time. Furthermore, its performance would not change anywhere along the continuum from fully dense arrays to fully sparse arrays. No such implementation is known for von Neumann architectures.

Dijkstra proposed *array variables* [1], combining some of the features of extensible arrays and sparse arrays. These are useful abstractions for deriving algorithms, but they are too inefficient for use in real programs.

Functional arrays. A functional array is immutable: it can be read, but never changed. Since no side-effecting operations are allowed, all accesses to such an array are performed by pure functions.

The application *lookup* a i gives the value of $a[i]$. Instead of modifying an old array, we must construct a new one by updating the old array with an index and new value. Thus *update* a i x constructs a new array identical to a, except that *lookup* (*update* a i x) $i = x$. The old array a is unaffected. Programs using functional arrays are constantly creating new arrays and discarding old ones.

Functional array updates are especially difficult to implement efficiently. In fact, nearly all past research on them has focused on methods for avoiding their use, rather than methods for making them fast!

The difficulty can be seen by considering two naïve implementations of *update*. The first idea is simply to copy the old array into a new block of memory which can then be updated destructively. This makes *lookup* fast, but *update* is prohibitively slow and wasteful of memory. The second idea, at the other extreme, is to record the updated value along with a pointer to the old array. This makes *update* fast but *lookup* becomes prohibitively slow. Various more sophisticated algorithms exist which amortise the costs between *lookup* and *update*, making both of them prohibitively slow.

The relationship between functional arrays and functional programming is surprisingly subtle. Since functional arrays are immutable, they satisfy referential transparency and equational reasoning, two of the cornerstones of functional languages. Nevertheless, functional arrays can be added just as easily to imperative languages. On the other hand, ordinary mutable arrays can be added to functional languages, as long as programs are restricted to single-threaded array access. This can be done using program analysis [2], structured access control [4] and expressive type systems [5]. Functional arrays are important because some algorithms require the facilities they provide — sharing, rollback and controlled access to alternative updates — not because of an illusory connection with functional languages.

3. ESF array operations

The last section described three important general array structures. It is conjectured to be impossible to implement any of them on a von Neumann architecture without paying a penalty, either in poor performance or in restricted access. *Extensible sparse functional (ESF) arrays* combine all three generalisations into a new data structure which is even more flexible but more difficult to implement.

The elements of an ESF array are assumed to have type *Value*. The array indices have type *Index* and an array itself has type *Array*. Some operations return a pair of indices with type *IndexPair*. The basic ESF array operations are described below, with the following naming conventions: $a, a' : \mathit{Array}$, $i : \mathit{Index}$, $x : \mathit{Value}$, and $ip : \mathit{IndexPair}$.

- *empty* : *Array* is a predefined constant array with the property that *lookup* (*empty*, i) is undefined for any index i.

- $a' := \mathit{update}\ (a, i, x)$ returns a new array a' exactly like a except that $\mathit{lookup}\ (a', i) = x$. The old array a still exists and is completely unaffected.
- $x := \mathit{lookup}\ (a, i)$ returns the value of array a indexed at i, like $a[i]$ in Pascal.
- $a' := \mathit{remove}\ (a, i)$ is similar to update. The result a' is exactly like a except that $\mathit{lookup}\ (a', i) = \mathit{Undefined}$, although a may be defined at index i. The old array a is unchanged.
- $ip := \mathit{bounds}\ (a)$ returns a structure ip containing the lower bound and the upper bound of a (i.e. the smallest/largest index at which a is defined).
- $\mathit{forward}\ (a, i)$ returns the smallest index $j > i$ (if it exists) such that $\mathit{lookup}\ (a, j)$ is defined. This is the index of the next interesting value in a starting from i.
- $\mathit{backward}\ (a, i)$ returns the largest index $j < i$ (if it exists) such that $\mathit{lookup}\ (a, j)$ is defined. This is the index of the previous interesting value in a starting from i.
- $\mathit{size}\ (a)$ gives the number of defined (non-default) elements in a.
- $\mathit{extent}\ (a)$ is the difference between the highest and lowest index of any defined value in a.

The data structure defined by these operations is purely functional since no operation modifies an existing object. The functions can only create new data structures or dispose of inaccessible ones. There is no restriction on the index used in an *update*, as with ordinary arrays, so ESF arrays are arbitrarily extensible. The *forward* and *backward* functions provide fast searching for non-default values in an array, supporting sparse arrays.

Besides the existing algorithms relying on the three generalised arrays, ESF arrays support several demanding new applications, and more are likely to be discovered.

Lambda calculus reduction illustrates the flexibility of ESF arrays. With ordinary data structures, there is no known way to implement the environment in a lambda calculus reducer so that both application and binding lookup take constant time. However, it is easy to do that with ESF arrays: each environment is represented by an array and each lambda variable is treated as an index. Thus $\rho' = \rho[E/x]$ becomes $p' = \mathit{update}\ (p, x, E)$ and $\rho\ x$ becomes $\mathit{lookup}\ (p, x)$. The ESF array algorithm ensures that application and binding lookup always take constant time, regardless of the amount of sharing present in the environment. This may lead to further complexity results for the lambda calculus.

There has been increasing activity in data parallel algorithms for time-critical database operations. ESF arrays appear ideal for databases, and they can probably be extended to supply powerful associative searching techniques. Further research is needed to assess the utility of this application.

Most algorithms using dynamic data structures with heap allocation are unsuitable for real time applications, since a garbage collection could occur at any time. Every ESF array operation takes constant time, and an update will never trigger a long garbage collection. An update will never fail unless memory

is completely full of accessible data, and in that case update will signal a 'memory full' error in constant time. So a significant application of ESF arrays may be real time algorithms on dynamic data structures.

4. The key ideas

Before presenting the actual algorithms for ESF array operations, this section presents a sequence of key ideas required to understand the algorithm. The following section presents an example showing how these ideas work in practice.

Ordinary array representations use the memory address to associate an array element with its index. That won't work here because it interferes with sparse representations and sharing. Therefore we introduce an explicit representation of the association between indices and values, which are called *bindings*. The memory is organised into a sequence of *cells*, where each cell contains an `Index`, `Value`, and a flag `Full` indicating whether that cell contains an array element.

The primary difficulty with functional arrays is the massive sharing they produce. Suppose that `a' := update (a,i,x)` and `a` contains n elements.

- If `lookup (a,i)` is defined then `a'` will also contain n elements: (the $n-1$ elements of `a` at indexes other than `i`, and the `(i,x)` pair).
- If `lookup (a,i)` is not defined then `a'` will contain $n+1$ elements: all n elements from `a` plus the `(i,x)` pair.

The heart of ESF arrays is a mechanism for exploiting all the sharing without slowing down access to elements of any array.

The next step requires a significant change of perspective. Ordinary arrays use the address of an element to determine which array that element belongs to; in effect an array 'knows' what its elements are. ESF arrays turn that around. An object of type `Array` is nothing more than the *name* of an array; its only significant property is that two distinct arrays have distinct names. An array doesn't know what its elements are. Instead, each full cell contains the name of every array that contains the element stored in the cell; this is called the element's *inclusion set*. For example,

Operation	Index	Value	Inclusion set
$a_1 :=$ *update* $(empty, 1, 3.14)$	1	3.14	$\{a_1\}$
$a_2 :=$ *update* $(a_1, 2, 2.718)$	2	2.718	$\{a_2, a_1\}$
$a_3 :=$ *update* $(a_1, 3, 42.0)$	3	42.0	$\{a_3, a_1\}$

There is no bound on the size of an inclusion set, yet we must find a way to represent an arbitrary inclusion set in constant space — one cell. Furthermore, we must be able to evaluate $a \in S$ in constant time for arbitrary array a and inclusion set S. It is impossible to represent arbitrary sets in constant space, but inclusion sets are not arbitrary. They can only be generated by sequences of updates, and that forces inclusion sets to have certain structural properties which the algorithm can exploit.

So the key to ESF arrays is the efficient manipulation of inclusion sets. We start by introducing an array code for every array name. The reason is that array names must remain constant, but the codes will need to change frequently.

For every array name *a*:*Array* there is a corresponding code *c*:*Code* which is represented as a natural number. The function *encode* (*a* : *Array*) : *Code* returns the current code belonging to *a*. This is required to be a one to one correspondence, so we can refer to an individual array unambiguously using either its name or its code. The empty array always has code 0, so *encode* (*empty*) = 0.

An inclusion set S is represented by a pair (*lo*,*hi*) such that for any array a, $a \in S$ iff $lo \leq encode\ a \leq hi$. This is good because (1) S is represented in constant space (two binary numbers) and (2) the $a \in S$ predicate is implemented in constant time by two comparisons of binary numbers.

The compact representation of inclusion sets works because, after *any* sequence of updates, there exists a mapping from arrays to codes such that every inclusion set can be represented as above. However, whenever an update is performed it is necessary to adjust the mapping from arrays to codes and the inclusion sets of existing elements. This requires a small constant amount of arithmetic for every cell.

5. An extended example

The best way to get a feel for the ESF algorithm is by watching how the memory state changes as a sequence of updates is performed. This section gives an example showing how the inclusion sets change, and the next section gives the central parts of the algorithm.

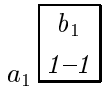
Each step in the following example is illustrated with a picture showing (on the left) the history of updates so far; (in the middle) the dependency tree resulting from that history; (on the right) the array name/code mapping resulting from the history.

The example follows some notational conventions. The actual index/value bindings are irrelevant, so they are simply identified as b_i. A cell is pictured as a box containing a binding and an inclusion set represented by *lo*–*hi*, and annotated on the lower left corner with the name of the array whose update created that binding. The dependency relations among the cells are illustrated by drawing an explicit tree structure. It is important to understand, however, that there is no explicit tree structure in the machine; the only information actually present consists of the cell contents and the name/code mapping.

The following convention is used for array codes: newly created codes are in *italics*, old codes that were modified are in **boldface**, and old ones that remain unchanged are in roman type.

We must begin by updating `empty`. The code of `empty` is always 0, so the new binding has inclusion set 1–1 and the result a_1 is given code 1.

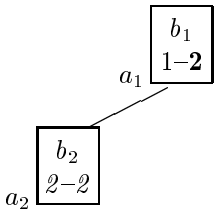
a_1 = update *empty* b_1



a_1= *1*

The code of a_1 is 1, so when it is updated the result a_2 gets code 2. The new binding has inclusion set 2–2 representing $\{a_2\}$. The inclusion set of binding b_1 is changed: its lower bound remains the same since $1 \not< 1$, but its upper bound is incremented since $1 \leq 1$. Thus the inclusion set of b_1 becomes 1–2 which represents $\{a_1, a_2\}$.

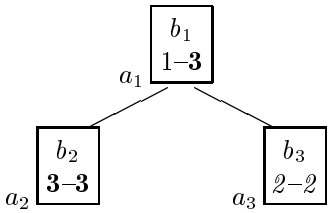
a_1 = update *empty* b_1
a_2 = update a_1 b_2



$a_1 = 1$
$a_2 = 2$

The next update illustrates the flexibility of functional arrays by making an alternative update to a_1. Since a_1's code is still 1, the new array a_3 receives code 2. But now the existing code of a_2 must be incremented, and a_2's binding b_2 must have its inclusion set adjusted. Notice how both the lower and upper bounds of b_2 are incremented (since $1 < 3$ and $1 \leq 3$) but only the upper bound of b_1 is changed (since $1 \leq 1$ but $1 \not< 1$). As a result, the *representation* of b_2's inclusion set has changed from 2–2 to 3–3, but its *value* remains $\{a_2\}$.

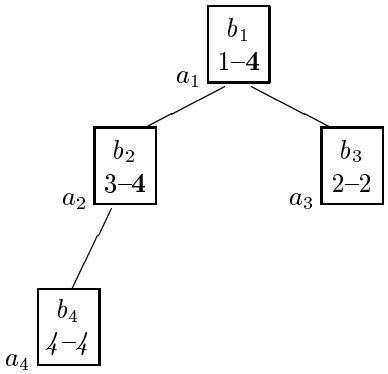
a_1 = update *empty* b_1
a_2 = update a_1 b_2
a_3 = update a_1 b_3



$a_1 = 1$
$a_2 = \mathbf{3}$
$a_3 = 2$

Now let's update something other than a_1! Notice how every binding above the new binding b_4 has its inclusion set adjusted, ensuring that a_4 contains all the bindings on the path from b_4 to the root of the dependency tree.

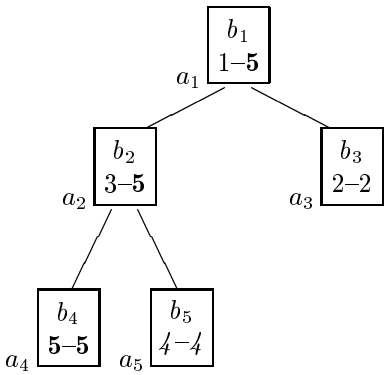
a_1 = update *empty* b_1
a_2 = update a_1 b_2
a_3 = update a_1 b_3
a_4 = update a_2 b_4



$a_1 = 1$
$a_2 = 3$
$a_3 = 2$
$a_4 = 4$

The next update is similar, and creates still more sharing.

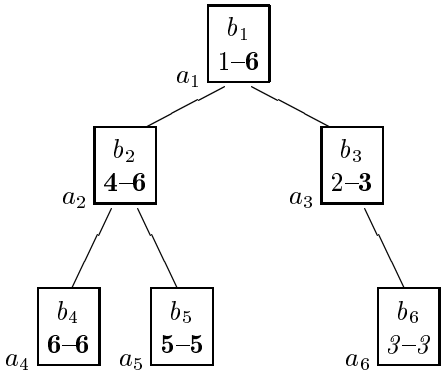
a_1 = update *empty* b_1
a_2 = update a_1 b_2
a_3 = update a_1 b_3
a_4 = update a_2 b_4
a_5 = update a_2 b_5



$a_1 = 1$
$a_2 = 3$
$a_3 = 2$
$a_4 = \mathbf{5}$
$a_5 = \mathit{4}$

Our final update demonstrates the massive quantity of computation required by update. Since we are updating a_3 whose code is currently 2, the following items must all be incremented: codes > 2; lower bounds > 2; upper bounds ≥ 2. As a result, at least one change is made to every nonempty cell in the machine! However, notice that the inclusion sets of b_2, b_4 and b_5 remain the same, while the inclusion sets of b_1 and b_3 have had the new binding b_6 added.

a_1 = update *empty* b_1
a_2 = update a_1 b_2
a_3 = update a_1 b_3
a_4 = update a_2 b_4
a_5 = update a_2 b_5
a_6 = update a_3 b_6



$a_1 = 1$
$a_2 = \mathbf{4}$
$a_3 = 2$
$a_4 = \mathbf{6}$
$a_5 = \mathbf{5}$
$a_6 = \mathit{3}$

In this example, all the elements have belonged to the same family of arrays. There is no problem if a completely unrelated familiy is created by updating `empty`. That simply produces a new array with code 1, and all existing families have their codes incremented, moving them out of the way. There is no limit to the number of independent families of arrays, and there is never any conflict among them.

6. The data parallel ESF array algorithm

This section shows how *update* and *lookup* work. There is not space to give all the details, or a correctness proof, but the most important techniques are shown here.

Each array element is stored in a cell, which is a collection of fields along with a small arithmetic processor capable of incrementing, decrementing and comparison. The `mapName` and `mapCode` fields contain an array name–code pair,

which is part of the representation of the mapping between names and codes. These fields are unconnected with the array element stored in the rest of the cell's fields.

```
processor cell =
  record
    full    : Boolean;   {cell contains an array element?}
    lo, hi  : Code;      {inclusion set = [lo..hi]}
    ind     : Index;     {index of element}
    val     : Value;     {value of element}
    mark    : Boolean;   {cell is active?}
    mapName : Array;     {name of an array}
    mapCode : Code;      {code of that array}
  end;
```

Several parallel constructs are used in the algorithm. The '`parallel` S;' construct is a map function; it causes every cell processor to execute `S` in parallel. Field names used in `S` refer to fields within the local processor. In effect, the `parallel` construct combines parallelism, a Pascal **for** loop, and a Pascal **with** statement.

The `globalOr` function calculates the Or of a bit or field in every cell in the system using a combinational logic tree network, giving it the same time complexity as accessing a word in a RAM, which requires a combinational tree decoder. A parallel minimum operation is also required; this again has the same complexity as a RAM.

Most of the array operations must find out the current code belonging to an array. This requires an associative search of the coding table, performed by `encode`. Note that `globalOr` is used to fetch the `mapCode` field from the cell whose `mark` flag is 1.

```
function encode (a:Array): Code;
  begin
    parallel  mark := mapName=a;
    encode := globalOr mapCode
  end; {encode}
```

The `update` function must create a new array and its corresponding code; adjust the inclusion sets of existing cells; store the new binding into the newly allocated cell; and adjust the array name/code mapping.

```
function update (a:Array; i:Index; v:Value) : Array;
var c, c' : Code;
    a' : Array;
begin
```

First the code `c` of the old array is calculated, and the code of the new array is defined to be `c`+1. A new cell and array name are also allocated. The `newName` function generates a new array name using an associative search of the name–code mapping for an unused name, while `newCell` performs an associative search

of the `full` fields of the cells. An important point is that allocation of names and cells always succeeds in constant time (or fails in constant time if the system is full).

```
c := encode a;        {input array code}
c' := c+1;            {result array code}
newCell (x);          {allocate new cell}
a' := newName ();     {name of result}
```

The inclusion sets of existing cells must now be adjusted.

```
parallel
  begin
    if lo>c  then lo := lo+1;
    if hi>=c then hi := hi+1;
  end;
```

The next step is to put the new array element into the allocated cell. The code of the new array is `c'`, so the element has inclusion set `c'`—`c'`.

```
with x do
  begin  lo := c';  hi := c';  ind := i;  val := v;  end;
```

Finally, the name/code mapping is adjusted.

```
  parallel
    if mapCode>c then mapCode := mapCode+1;
end; {update}
```

The `lookup` function marks every cell that meets two requirements: (1) its index matches the index being searched for, and (2) its inclusion set contains the array being accessed. Several cells may satisfy both constraints. For example, consider

```
a1 := update (empty, 5, 3.14);
a2 := update (a1, 5, 2.7);
x := lookup (a2, 5);
```

Now both the cell containing the (5, 3.14) binding and the cell containing the (5. 2.7) binding have matching indices and inclusion sets. The (5, 3.14) binding is 'shadowed' by the more recent (5, 2.7) binding, so `lookup` has to find the most recent candidate cell. This can be performed either by representing the 'age' of a binding explicitly with a cell, or by using the location of a cell within the memory to encode the relative ages of two bindings. The details are straightforward, and omitted here.

```
function lookup (a:Array; i:Index): Value;
  var  c : Code;
  begin
    c := encode a;
    parallel   mark := ind=i and lo<=c and c<=hi;
```

```
    found := globalOr mark;
    if found then
        begin
          d := globalMin age;
          parallel   mark := mark and age=d;
          lookup := fetch
        end
      else lookup := Fail;
  end; {lookup}
```

The `free` procedure decrements the bounds of inclusion sets just as `update` increments them. If the array being freed is the only array containing an element, then the inclusion set of that element becomes c—$c-1$. This represents the empty set, enabling the system to reclaim the cell.

```
procedure free (a:Array);
  var  c : Code;
  begin
    c := encode a;
    parallel
      begin
        if c<=code then code := code-1;
        if lo>c  then lo := lo-1;
        if hi>=c then hi := hi-1;
        if lo>hi then dispose
      end
  end; {free}
```

7. Parallel machine models for ESF arrays

Every time an `update`, `lookup` or `free` is executed, every cell in the system has to perform a simple computation consisting of a few comparisons, increments and decrements. The calculations can be expressed by *mapping* a function over a set of cells in parallel, and the small amount of communication required can be implemented by *scanning* and *folding* associative functions over the cells.

The mapping operations (conditional incrementing and decrementing) require no communication so they take constant time. The scanning (global Or and finding minimum ages) can be performed by a combinational network in $O(\log N)$ time, *the same time it takes a random access memory (RAM) to access a word*. The usual convention is to count a RAM word access as $O(1)$. According to that convention *every operation on ESF arrays takes only constant time*. Several implementation techniques are feasible:

- *Simulation* on a sequential machine has been completed using the functional language Haskell to specify both the algorithm and the parallel architecture.
- *Emulation* using the massively parallel Connection Machine CM-200 has been partially completed.

- *Direct VLSI implementation* would give the fastest system. A VLSI architecture similar to Apsa [3] would be suitable. Such a memory circuit has the same chip area and speed characteristics as a RAM chip, yet it can perform the ESF array algorithms in a constant number of instructions. This approach would be much faster and cheaper than emulation on the Connection Machine, since the cell processors would be very simple and a general routing network is not required.

An interesting characteristic of the ESF algorithm is that it performs far more work than ordinary array accesses seem to, but it overcomes that extra work through massive parallelism. Every time an update or lookup is performed, the memory performs $\Theta(N)$ arithmetic operations, where N is the number of cells in the memory. However, each cell must perform only a constant number of operations, and the cells can all execute in parallel.

To put this in perspective, consider that a RAM memory containing N words also performs $\Theta(N)$ computations inside its address decoder as it accesses 1 word. The point is that the RAM wastes almost all of that work, while the ESF memory performs a trivial but useful computation in each cell. A common error is to ignore the wasted work performed by a RAM (giving $\Theta(1)$ access), while counting the useful work performed by ESF (giving $\Theta(\log N)$ access).

8. Conclusion

Massive parallelism makes it possible to combine three previously unrelated generalisations of arrays — extensible arrays, sparse arrays and functional arrays — into a powerful new data structure called the ESF array. The time, space and chip area complexity measures are exactly the same for ESF array operations as those of an ordinary array operations using RAM chip.

Various extensions to ESF arrays are possible, such as mapping, folding or scanning a function over the elements of an array in parallel. The massive sharing that is possible with functional arrays requires some of the extensions to be done carefully, and the details are a topic of future research.

References

1. E. W. Dijkstra, *A Discipline of Programming.* Prentice-Hall (1976).
2. P. Hudak, A. Bloss, The aggregate update problem in functional programming systems. *Proc. 12th ACM POPL* (1985).
3. J. T. O'Donnell, T. Bridges, S. W. Kitchel, A VLSI implementation of an architecture for applicative programming. *Future Generation Computer Systems*, **4**(3) (Oct. 1988) 245–254.
4. S. L. Peyton Jones, P. L. Wadler, Imperative functional programming. *Proc. 20th ACM POPL* (1993) 71–84.
5. P. L. Wadler, Linear types can change the world! *Programming Concepts and Methods*, North-Holland (1990).