Flexible Algorithms and their Implementation:

Enabling Well Defined Order-Independent Execution

R. B. Yehezkael, M. Goldstein, D. Dayan, S. Mizrahi

November 2015 – כסלו תשע"ו

All authors are with the Flexible Computation Research Laboratory, Jerusalem College of Technology,
Hawaad Haleumi 21, POB 16031, Jerusalem 91160, Israel.
Here are their contact details:
R. B. Yehezkael (phone: +972 2 6751294; fax +972 2 6432145; e-mail: rafi@jct.ac.il).
M. Goldstein (e-mail: goldmosh@jct.ac.il).
D. Dayan (e-mail david674@bezeqint.net).
S. Mizrahi (e-mail: shimon@jct.ac.il).

***Abstract*—Techniques are presented for ensuring well defined flexible execution, i.e. parallel and unordered sequential execution where the values read are independent of execution order. This is done by refining the scope rules of variables and defining where they may be initialized, where they may be updated, and where they may be read. Given these refined scope rules, this approach extends or replaces once-only assignment with suitable composite assignments to ensure well defined read values. Examples of such suitable assignments are once only assignment, "or=", "and=", "+=", "-=", etc.**

A flexible algorithmic language with these characteristics is described. The "core" of this language is based on functions (or procedures) with "IN", "OUT" but no "INOUT" parameters, and conditional statements, with or without an "ELSE". Blocks, loops, case statements etc. (including nested forms) are not part of the "core" language but viewed as abbreviations for certain compound forms in the "core" language.

An implementation of an embedded flexible language (EFL) based on these principles is described in a companion paper [1].

Work is in progress on the following topics which are briefly described here.

- **Hardware support for flexible execution.**
- **Educational aspects of flexible execution.**

***Index Terms*—Flexible execution, Scope rules, Composite assignments, Hardware support.**

1 Introduction

Conventional algorithms change the values of variables during the course of execution, and so to ensure uniquely defined results, sequential execution is required. The advantage of the sequential approach is that it is easy to implement and easy to follow the execution steps. The analysis and debugging of sequential algorithms is difficult in view of the fact that the statements of the

algorithm can be quite dependent on each other and so many interactions need to be considered. Also, a particular value can depend on changes made by operations in the (distant) past, which only adds to this difficulty. Parallel programming is even harder, particularly if the programmer explicitly handles the coordination.

1.1 Independence of execution order

Our approach is to define a flexible algorithmic language which enables much more independence between statements in a program. Though writing algorithms in such a language is harder than writing conventional algorithms, the fact that there is more independence between statements has advantages:

1. Algorithms may be executed in a variety of orders, sequential and parallel, with identical end results.
2. The analysis and debugging of algorithms is easier because greater independence between the statements of the algorithm means there are fewer interactions to consider.

1.2 The notations of classical mathematics and imperative programming languages

Consider the following:

x = b - c

a = c + 2

b = 4

c = 1

y = x / (x + 3)

z = b + c

These can be viewed as definitions of variables in mathematics or as statements in a programming language.

Mathematical View: Values of variables independent of order of definitions. No concept of previous value of variable. Execution order not explicitly specified.

Imperative Programming View: Values of variables depend on order of statements and previous values of variables. Execution is sequential.

Note that though x = x+1 is acceptable as a statement in imperative programming languages its use is unacceptable in mathematics. Also, if all we needed is the value of z, mathematically only b, c, z need to be determined but in imperative programming all steps would be executed.

The subtlety of the simple mathematical variable is that it allows computations to be performed in a variety of orders sequential and parallel. It also enables only some of the variables to be computed. Thus partial computation is possible. One of the reasons for this flexibility is that mathematical variables may not be updated. Another reason for this flexibility is that a simple mathematical variable is really a function of zero arguments in programming language terms. Here is the programming equivalent of the mathematical view in *Python*-style [2].

```
def r = x( ):    r = b( ) – c( )

def r = a( ):    r = c( ) + 2

def r = b( ):    r = 4

def r = c( ):    r = 1

def r = y( ):    r = x( ) / (x( ) + 3)

def r = z( ):    r = b( ) + c( )
```

There are difficulties with using the above definitions. For example, when computing y ( ) we will compute x( ) twice even though the same value will be produced on both occasions. This can be overcome by using static storage for caching the function values and for status information about the state of the variable (i.e. unassigned, assigned, in computation, value in error). In this way, we can handle the computation flexibly without unnecessary re-computation. However, even with this improvement, functions of zero arguments will be less efficient than variables of imperative programming languages.

The approach taken in this paper is in-between the mathematical view and the imperative programming view:

1) As in mathematics (and as in functional programming), variables receive a value only once, but they are not functions of zero arguments.

2) Like mathematical notation, computation may proceed in a variety of orders, sequential and/or parallel.

3) Like the imperative programming approach, we require all steps to be performed. So partial computation is not supported.

Regarding once only assignment, this is only a starting point. Later we introduce certain composite assignments which are more general than once only assignment and enable well defined unordered sequential execution and sometimes well defined parallel execution.

It turns out that this kind of language can also be used to describe hardware block diagrams [3, 4], but we do not pursue this aspect here.

1.3 Layout of the paper

The remainder of this paper consists of the following sections:

2 A FLEXIBLE ALGORITHM

Let us now give an example of a simple algorithm in this flexible language and describe different execution methods.

```
def vOut = reverse (vIn, low, high):
#    SPECIFICATION:
#  IN - vIn is a vector
#          low and high are positions within the vector vIn.
# OUT - If low ≤ high, then within the range [low .. high], vOut is like vIn but  reversed.
#          Other elements of vOut are not given values by this function.
#          If low > high, then the function does nothing to vOut.
if  (low < high):
    vOut [high] = vIn [low]
    vOut = reverse (vIn, low+1, high-1)  # A
    vOut [low] = vIn [high]
  elif (low == high):
    vOut [high] = vIn [high]
```

```
# end reverse
```

For example, to reverse a vector [1, 2, 3, 4, 5] and put the result in rOut we write:

```
rOut = reverse ([1, 2, 3, 4, 5], 0, 4)  # B
```

2.1 Notes

1. Parameters may only be IN which are given inside the parenthesis, or OUT which are given in the left-hand-side of the assignment-like prototype definition of the function. There may be several IN parameters and several OUT parameters.
2. It is an error to assign twice to the same simple parameter or simple component of a parameter having (several) components.
3. Later on # A and # B will be used as return addresses, when describing execution methods, below.

3 EXECUTION METHODS

The restrictions described above allow the execution to be performed in a variety of orders with equivalent end results. Four execution methods are described.

1. Parallel shown in tree form.
2. Sequential using a stack, with immediate calls.
3. Sequential using a queue, with delayed calls.
4. Sequential using a current function data area, and a stack with delayed calls.

We also discuss how to use a waiting area for efficient virtual memory handling.

In presenting these methods we assume v = [1, 2, 3, 4, 5] and that we are executing:

```
r = reverse (v, 0, 4); # B
```

3.1 Parallel execution shown in tree form

This is shown in Fig 1. For simplicity we illustrate the case where computation proceeds level by level, i.e. synchronously.

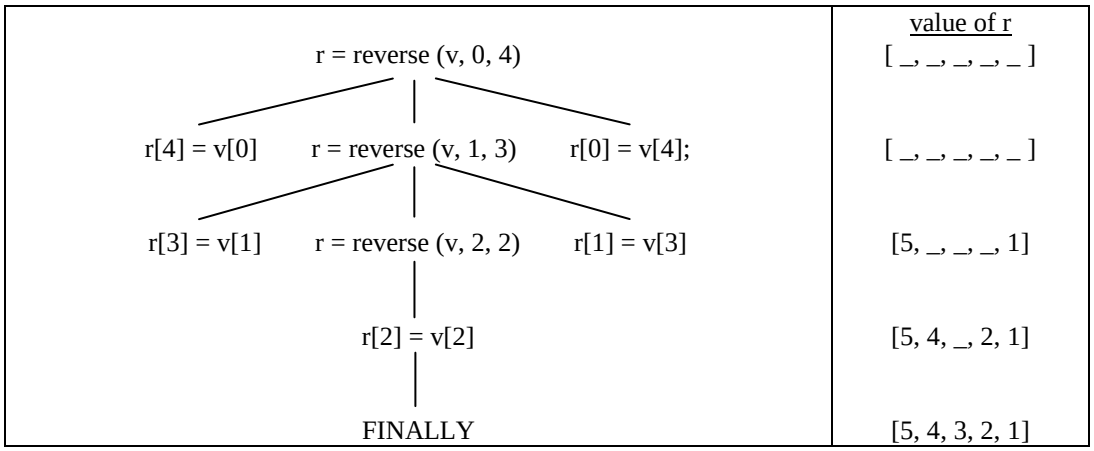


Fig. 1. Parallel execution shown in tree form.

Note that though the values of r are determined assuming that the computation proceeds synchronously, there are other execution orders possible. For example, if the calls to reverse are expanded level by level before executing the assignments, then all the assignments may be performed in any order.

The above can also be represented by a sequence of pairs of the form

set of statements::values of OUT variables as follows.

{ r = reverse (v, 0, 4); }::r=(_, _, _, _, _)

≡{ r [4] = v[0]; r = reverse (v, 1, 3); r [0] = v[4]; }:: r=[_, _, _, _, _]

≡{ r [3] = v[1]; r = reverse (v, 2, 2); r [1] = v[3]; }:: r=[5, _, _, _, 1]

≡{ r = reverse (v, 2, 2); }:: r=[5, 4, _, 2, 1]

≡{ r [2] = v[2]; }:: r=[5, 4, _, 2, 1]

≡{ }:: r=[5, 4, 3, 2, 1]

Note that though we have used sets here, the use of a multiset or bag may enable more effective use of parallelism. This is because we avoid the overhead of ensuring that an element appears only once in a set and it may be preferable to execute the same statement more than once.

3.2 Sequential execution using a stack with immediate calls

This is shown in Fig. 2. Here calls are executed immediately and a record is made in the stack of the value of variables at the time of the call. The return address is also recorded in the stack. (This is the standard way of handling calls in imperative programming languages.)

Note: Top of stack is at the right. A, B denote return addresses.

Stack			Value of r at	call/exit
r=reverse(v,0,4) |B			[_, _, _, _, _]	call
r=reverse(v,0,4) |B	r=reverse(v,1,3) |A		[_, _, _, _, 1]	call
r=reverse(v,0,4) |B	r=reverse(v,1,3) |A	r=reverse(v,2,2) |A	[_, _, _, 2, 1]	call
r=reverse(v,0,4) |B	r=reverse(v,1,3) |A	r=reverse(v,2,2) |A	[_, _, 3, 2, 1]	exit
r=reverse(v,0,4) |B	r=reverse(v,1,3) |A		[_, 4, 3, 2, 1]	exit
r=reverse(v,0,4) |B			[5, 4, 3, 2, 1]	exit

Fig. 2. Sequential execution using a stack with immediate calls.

3.3 Sequential execution using a queue with delayed calls

This is shown in Fig. 2. Here the calls are delayed and stored in a queue until the current function completes execution. (In a composition, an outer call is queued after an inner call.) There are no return addresses in the queue, but a note is made of the function to be activated in the future, and its parameters. When the current function completes, its entry is removed from the head of the queue, and processing continues with the next entry.

Note: Head of queue is at the left.

Queue		value of r at exit
r = reverse (v, 0, 4)		
r = reverse (v, 0, 4)	r = reverse (v, 1, 3)	(5, _, _, _, 1)
r = reverse (v, 1, 3)		
r = reverse (v, 1, 3)	r = reverse (v, 2, 2)	(5, 4, _, 2, 1)
r = reverse (v, 2, 2)		(5, 4, 3, 2, 1)

Fig. 3. Sequential execution using a queue with delayed calls.

This method does not handle all cases, but will handle tail recursive definitions even if there are multiple tails. The usual definition of tail recursion requires a single tail and is a recursive form which forces sequential execution and is equivalent to flowcharts. The following definition of f is not tail recursive in the usual sense (single tail). It has multiple tails - the calls to g1, g2, g3, which are functions defined by the user (not primitive functions). This kind of definition can be executed sequentially by the queue. (It also enables g1(...), g2(...), g3(...) to be executed in parallel using other execution methods.)

```
def mOut, nOut, pOut = f (i, j):
if (...):
  mOut=g1(i+1)
  nOut=g2(j+1)
  pOut=g3(i+j)
else
  mOut=0, nOut=1, pOut=2
# end f
```

3.4 Sequential execution using a current function data area, and a stack with delayed calls

Here the data for the current function are stored in a current function data area off the stack, preferably in registers. Calls are delayed and stored in a stack until the current function completes execution. (In a composition, an outer call is stacked before an inner call.) There are no return addresses in the stack, but a note is made of the function to be activated in the future, and its parameters. When the current function completes, the stack is popped into the current function data area, and the function recorded therein started.

Stack	Current function data area	value of r at exit
	r = reverse (v, 0, 4)	
r = reverse (v, 1, 3)	r = reverse (v, 0, 4)	(5, _, _, _, 1)
	r = reverse (v, 1, 3)	
r = reverse (v, 2, 2)	r = reverse (v, 1, 3)	(5, 4, _, 2, 1)
	r = reverse (v, 2, 2)	(5, 4, 3, 2, 1)

Fig. 4. Sequential execution using a current function data area, and a stack with delayed calls.

The execution is similar to that of the queue, but there can be differences in the order of the delayed calls.

This method is restricted in that it cannot for example, handle a function in the condition part of an "if" statement. However, by rewriting the "if" as follows, even this case can be handled.

```
if (G(...)):              # Original "if" statement.
  statement1
else statement2
```

The "if" statement would be rewritten (by a compiler) as an internal block and a new local variable as follows:

```
:( new_variable=G(...) )
# Local variable(s) with their initial value.
if (new_variable):
   statement1
else statement2
```

NOTES: (a) This is NOT the usual method of using a stack with return addresses. The execution of the current function is NOT interrupted when an internal call needs to be made and the current function completes execution BEFORE internal calls are made. As a result, performance can improve because there are likely to be fewer page faults. (b) For ease of explanation we have explained what needs to be done by introducing an auxiliary variable and block. However we can get the same effect by simply using the standard call with return address mechanism, for the call to G.

3.5 Different ways of writing parameters in a function call

The definition of the function reverse given previously included a call written in functional style:

```
vOut = reverse (vIn, low+1, high-1);
```

Sometimes the style of a procedure call is clearer:

```
reverse (vIn, low+1, high-1, vOut);
```

Sometimes an assignment-like style is helpful:

```
reverse (vIn = vIn; low = low+1; high = high-1; vOut = vOut);
```

This can be abbreviated showing only the changes:

```
reverse (low = low+1; high = high-1);
```

Note that there is a fundamental difference between low = low+1, high = high-1 written above and the assignment statement which updates values. Here the variables "low", "high", on the left hand side are new variables which will be created when the

call is executed and the variables "low", "high" on the right hand side are existing variables holding values i.e. the variables on different sides of the "=" are different variables. Though this may look like we are updating the values of variables, this most definitely is not the case. In fact, we are giving new variables their values. So flexible execution remains possible with this arrangement.

4 CONVENIENT EXTENSIONS

The following extensions though not essential, are convenient, more readable and briefer in many situations. We require that all such extensions can be converted to core language described earlier. This ensures that all the flexibility of execution is retained.

4.1 Blocks and local variables

It is often useful to use a local variable to prevent re-computation of values. Suppose the value x+y-z is used several times in the definition of the function below.

```
def rOut = f(x,y,z):
# SPECIFICATION …
# statements of f
… x+y-z…
…
… x+y-z…… x+y-z…
…
# end of f
```

Then to avoid re-computing this value we can use a local variable in a block and write f as follows:

```
def rOut = new_f(x,y,z):
# Local variable(s) with their initial value.
:( xyz=x+y-z )
# statements of f with xyz in place of x+y-z
… xyz…
…
```

```
… xyz…… xyz…
…
# end of new_f
```

Note even though this is not written explicitly, the local variable of a block is of IN type only and is given a value with its declaration. This is to allow conversion to the core language.

Here is how this will look in the core language without a block and local variable but with an auxiliary function having an extra IN parameter.

```
def rOut = new_f(x,y,z):
rOut = block(x,y,z,x+y-z);
# end of new_f

def rOut = block(x,y,z,xyz):
# statements of f with xyz in place of x+y-z
… xyz…
…
… xyz…… xyz…
…
# end of block
```

This view treats a block as a function which has only one external call. This means of course that everything we have said so far about the flexibility, will remain true if we add blocks and local variables, since this notation can be viewed as a convenient abbreviation.

4.2 Labeled blocks (or *Loop Blocks*)

We allow blocks to be labeled (really a convenient abbreviation for an auxiliary function definition and its call). This notation allows us to write in a style similar to loops ("for", "while"). We require the same scope rules for labels as for functions, variables and other names. Here is how we can square every element in a vector using a labeled block (similar to a while loop).

```
def vOut = squares(vIn):
```

```
# SPECIFICATION: vIn, vOut are vectors having the same length.
#                    Each element of vOut is the square of the corresponding element of vIn.

:loop ( i = 0 )
   if (i < vIn.length):
         loop(i = i+1); # for next "iteration"
         vOut(i) = vIn(i)**2
# end of squares
```

Note that "loop" is the name of the labeled block (or loop block) and "i" is its local variable. As with the block, the labeled block loop can be translated to the core language by using an auxiliary function as follows.

```
def  vOut = new_squares (vIn):
   loop(i = 0)
# end of new_squares

function vOut = loop(vIn):
    if (i < vIn.length):
         loop(i = i+1); # for next "iteration"
         vOut(i) = vIn(i)**2
# end of loop
```

Again, this view treats a labeled block as a function which may have one or more external and internal calls. Again this addition will retain the flexible execution capability. In particular, parallel execution of several iterations of the loop is possible by using different local variables for each iteration.

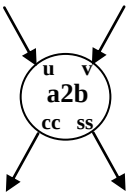
Note that for the purpose of definition, we have rewritten blocks and labeled blocks in terms of functions. This does not mean that we must implement them in this way.

More kinds of loops are described in the companion paper paper [1].

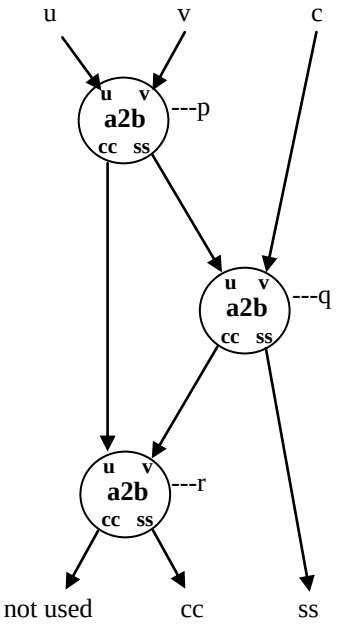
4.3 Explicit bindings

Explicit bindings are a direct way of showing how values pass between various functions. This is explained with an example, where the arrows show how values are passed; i.e. how arguments are bound to the appropriate formal parameters.

Let us assume that there is a hardware operation "a2b" for adding two bits giving their carry and sum respectively (i.e. a half adder). This operation can be represented by the following diagram.



Let us now construct a full adder "a3b" for adding three bits giving their carry and sum respectively. Here is a block diagram for "a3b" where for the sake of illustration, we only use the operation "a2b" in the diagram.



Note that because of the particular situation above, the last operation "a2b ---r" may be replaced by an "or", which is the usual way of making the construction.

The above block diagram can be expressed using the features we have so far described as follows.

```
def cc, ss = a3b (u, v, c):
#
SPECIFICATION:
IN - u, v, c, are the bits to be added.
OUT - cc is the carry and ss is the sum.

:( c1,s1=a2b(u,v) )
  :( c2,s2=a2b(s1,c) )
    _,cc=a2b(c1,c2);
    # "_" on the left denotes an anonymous variable,
    # i.e. a result returned from a2b is not used.
    ss=s2
# end of a3b
```

This is not as direct as the diagram and there is a need for additional variables. To overcome this limitation of the textual form we use the brackets "[ ]" to make bindings explicitly.

```
def cc, ss = a3b (u, v, c):

# SPECIFICATION:
# IN - u, v, c, are the bits to be added.
# OUT - cc is the carry and ss is the sum.

[
  a2b p, q, r; # Like declaring three "objects" or "components" of "type" a2b.
  p.u=u; p.v=v;
  q.u=p.ss; q.v= c;
  r.u=p.cc; r.v=q.cc;
  cc=r.ss; ss=q.ss;
```

]

end of a3b

Notes

1) In the function definition above, p, q, r correspond to the labels p, q, r in the diagram. We call a variable such as p.u a labeled input. We call a variable such as r.cc a labeled output.

2) All labeled inputs are required to receive a value but not all labeled outputs need be used. In the diagram and in the definitions of "a3b", the output r.cc from the third "a2b" is not used.

3) The use of the underscore "_" above is similar to its use in Prolog to indicate an anonymous variable. An anonymous variable may not be read, that is, it may not be used on the right hand side of an assignment statement. Each occurrence of "_" is considered to be a distinct anonymous variable.

4) Bindings in algorithms may not be nested. This is not restrictive as any number of operations may be bound together in a single binding.

5) Note that in assignments, labeled input variables occur on the left and labeled output variables occur on the right. Also, in this example we started with a directed acyclic graph, and expressed it in textual form using explicit bindings. It is also possible to start with the textual form of explicit bindings and produce a directed graph showing the bindings. We require the graph to be acyclic and consider it a syntax error if this is not the case. This prevents a loop occurring in the graph of the explicit binding, which is important for avoiding circular wait situations i.e. deadlock.

6) Explicit bindings can provide some of the mechanisms of objects, including a form of inheritance as well as omission of components. However, further work is needed to design an object mechanism compatible with flexible execution.

4.4 Generalized call statement and scope rules

The generalized call statement uses the assignment style of parameter passing to provide additional possibilities. This statement has three components: an entry block followed by the function to be called followed by an exit block. We explain this further by example.

Suppose we wish to call a function f whose IN parameter is a vector v, and whose OUT parameter is a vector vOut. Suppose that we wish to pass to f a vector of a hundred elements in which v(i) is i^2 and receive the result in aOut. The usual way would be to declare a vector sq, put the squares in sq, and apply f to sq. If we were to use the assignment style of parameter passing we could write at length:

f(aOut=vOut; v(0)=0*0; v(1)=1*1; v(2)=2*2; ...; v(99)=99*99)

Here is a briefer and clearer way of doing this using the generalized call statement.

```
call
      :  loop(i=0)                    # Entry block
        if (i<100):
            loop(i=i+1); v(i)=i*i
      f                               # Function to be called
      :  aOut=vOut                    # Exit block
# end of call
```

Here is something similar with OUT parameters. Suppose we wish to apply f to a vector "u" and put the first fifty elements of f(u) in aOut and the last fifty elements in bOut. Here is how this can be done without declaring an auxiliary vector.

```
call
      :  v=u                          # Entry block
      f                               # Function to be called
      :  loop (i=0)                   # Exit block
        if (i<50):
            loop(i=i+1); aOut(i)=vOut(i); bOut(i)=vOut(i+50)
# end of call
```

Scope rules for IN and OUT parameters handled this way:

The entry block would be typically executed before the function call. The exit block would be typically executed after the function results are evaluated. Note that as this is a call, the IN formal parameters may only be assigned in the entry block and the OUT formal parameters may only be referenced referenced in the exit block. (The OUT formal parameters are not accessible in the entry block and the IN formal parameters are not accessible in the exit block.) This is the opposite to what is allowed in the body of the function.

This notation avoids the need to declare and initialize an additional vector. It is also suggestive, in that the entry and exit blocks may be executed either by the process which executes f or by the process which makes the call.

Generalized call statements may be nested.

4.5 Other features

We can add other features to such as "for loops", "while loops", "case statements" etc. We must be careful however that these extra constructs are equivalent to compound constructs in the core language so as to allow flexible execution. For example, here is how a "for loop" could be written as a labeled block which of course can be converted into the core language.

```
for (i=0; i<n; i=i+1):
statement list
```

This can be rewritten as:

```
:for (i=0)
    if (i<n):
        for(i=i+1); statement list
```

5 A More Complex Example

The bottom up algorithm for merge sort is described. This algorithm sorts a vector by repeated merging. For simplicity we assume that the length of the vector is a power of 2. For example suppose we wish to sort a vector of 8 elements. We view this as 8 single elements, and of course each single element is sorted.

(8, 1 , 7, 2 , 6, 3 , 5, 4)

We merge pairs of single elements to obtain:

(1,8 , 2,7 , 3,6 , 4,5)

Now we have four sorted pairs, so we merge two pairs and two pairs to obtain:

(1,2,7,8 , 3,4,5,6)

Now we have two sorted runs of four elements, so we merge them to obtain:

(1,2,3,4,5,6,7,8)

Now we have a sorted vector.

Note that at each stage the size of the sorted run in the vector is doubled with respect to the previous stage. Also the number of sorted runs in the vector is halved with respect to the previous stage. Here is what happens to these values.

Number of sorted runs	*Size of sorted run*
8	1
4	2
2	4
1	8

Assume that there is a function m2 with specification as follows.

```
def vOut=m2 (vIn, size, place):
# SPECIFICATION:
# vIn is a vector having two runs which are sorted in non-decreasing order.
# These ranges are of length "size" and at position "place" onwards in vIn.
# These two ranges are merged into a single range of length "2*size"
# and are put at position "place" onwards in vOut.
```

Let us now use the function m2 to define the function mergesort. For expository reasons, we first present a draft version of the definition followed by the final version of the definition.

```
# DRAFT VERSION - Part of the definition is in algorithmic style.

def vOut=mergesort(vIn):
# SPECIFICATION:
# vIn, vOut are vectors having the same length which must be a power of 2.
# vOut will be like vIn but sorted in non-decreasing order.

:loop(size=1)
# The variable "size" is the size of the sorted run.
if (size=length of vIn):
vOut=vIn
else
```

```
:loop(size=1)
  if (size=length of vIn):
     vOut=vIn
   else
     loop (
            size=size*2;
            vIn=m2( vIn, size, 0 );
            vIn=m2( vIn, size, 2*size );
            vIn=m2( vIn, size, 4*size );
            vIn=m2( vIn, size, 6*size );
            .
            .
            .
            vIn=m2( vIn, size, length of vIn – (2*size) );
          );

# FINAL VERSION - the algorithmic part has been replaced by a for loop.

def vOut=mergesort(vIn):
# SPECIFICATION:
# vIn, vOut are vectors having the same length which must be a power of 2.
# vOut will be like vIn but sorted in non-decreasing order.

:loop(size=1; number=length of vIn)
# The variable "size" is the size of the sorted run.
# The variable "number" is the number of sorted runs.
  if (size=length of vIn):
     vOut=vIn
   else
```

```
loop (
        size=size*2; number= number/2;
        for (i=0; i< number; i=i+2):
            vIn=m2( vIn, size, i*size );
      );
```

6 Flexible Execution - Two Forms

Flexible execution takes two forms, parallel execution or unordered sequential execution. In both cases we require that function values are uniquely defined. Here we show how introducing local variables may be used to allow more parallelism and how using suitable composite assignments enables unordered sequential execution.

6.1 Introducing local variables to enable parallelism

Consider the following fragment of a *sequential* program in the style of Python.

```
i=5
# statement list 1
i=i+7
# statement list 2
i=i+1
# statement list 3
```

In converting this to our flexible language we could write

```
:( i=5 )    # The value of this variable i is 5.
   # statement list 1 suitably modifed to single assignment form
   :( i=i+7 )    #    The value of the new variable i is 12.
      # statement list 2 suitably modifed to single assignment form
      :( i=i+1 )    # The value of the new variable i is 16.
         # statement list 3 suitably modifed to single assignment form
```

So there are three variables called "i", and this allows the execution of the three modified statement lists to be overlapped and executed in parallel. A similar approach can be used with labeled blocks (loops) so as to allow several "iterations" to be overlapped and executed in parallel.

In the same spirit, new variables are created for parameters when a function call is executed, also enabling parallelism.

6.2 Unordered sequential execution using composite assignments with a single function

Languages such as "C" or "Java" [5, 6] allow composite assignments of the form "x f= y", meaning x=(x f y), where f is a binary operator or function of two arguments. A typical example is x += y. It also supports abbreviated forms of composite assignments such as x++ meaning x += 1 or x=(x + 1). Can composite assignments (and their abbreviations) be incorporated while retaining well defined unordered sequential execution? For which kinds of functions and situations is this possible? Solutions to these questions are important as they will allow programmers to use the familiar algorithmic style, and they will enable flexible execution with reduced storage requirements. (The approach taken here differs from the approach of the previous subsection, in that we wish to allow flexible execution for certain composite assignments, *without* the need for multiple copies of a variable.)

For example, consider the function:

def xOut = g(…); xOut=e_0 : # initial value for xOut is e_0

xOut f= e_1;

...

xOut f= e_n;

Here e_0, ..., e_n are expressions and f is as above. As in regular assignments, the IN and OUT restrictions must be observed in composite assignments. The variable on the left hand side of the assignment must be an OUT variable and the variables on the right hand side of the assignment must be IN variables.

Here are some conditions, which enable execution flexibility while ensuring a uniquely defined final result for xOut.

1) It is required that ((a f b) f c) = ((a f c) f b). This is called the function condition for Unordered Sequential Composite Assignment Execution (function condition for USCAE). While the first parameter and the value of "f" must be of the same type, no restriction is placed on the second parameter of "f". Examples of functions satisfying this condition are "and", "or", exact arithmetic operations +, -, *, /, **, addition and subtraction with respect to a fixed modulus, etc. (Associative/commutative conditions or the existence of a unit element are not required.)

2) In this specific case, e_0, ..., e_n may be evaluated in any order, sequentially or in parallel. (In other situations it can be necessary to evaluate e_0 first.)

3) The initialization xOut= value of e_0, must be carried out before the composite assignments.

4) The composite assignments xOut f= value of e_1; ...; xOut f= value of e_n; must be performed in any sequential order.

5) Care must be taken that only the final value of an output variable is read or used in subsequent computations.

For now we assume that only one kind of assignment can be used with each variable. In the next subsection, this condition is relaxed.

The final value of xOut, equals value of the expression $(...(((e_0\ f\ e_{i_1})\ f\ e_{i_2})\ f\ e_{i_3})\ ...\ f\ e_{i_n})$ in an execution according to the above. Here $i_1, i_2, ... i_n$ are a permutation of 1, 2, ... n and are determined by the order in which the composite assignments are performed.

Condition (1) the USCAE function condition, allows consecutive e's in the aforesaid expression, except for e_0, to be swapped, without altering the value of the expression. As it is possible to sort by swapping consecutive elements, we can rearrange $e_{i_1}, e_{i_2}, ... e_{i_n}$ to $e_1, e_2, ... e_n$ by sorting on the subscripts of the e's using "bubble sort" for example, there will be no change in the value of this expression by this sorting. So this means that the final value equals the value of $(...(((e_0\ f\ e_1)\ f\ e_2)\ f\ e_3)\ ...\ f\ e_n)$ and so is uniquely defined.

Furthermore, we shall show that the USCAE function condition cannot be relaxed any further. Suppose that there are only two composite assignments and the final result is uniquely defined. If the first is made before the second, then the final result will be $((e_0\ f\ e_1)\ f\ e_2)$. But if the second is made before the first the final result will be $((e_0\ f\ e_2)\ f\ e_1)$.

As the final result is uniquely defined, it follows that $((e_0\ f\ e_1)\ f\ e_2) = ((e_0\ f\ e_2)\ f\ e_1)$, which is the USCAE function condition.

(Incidentally, conditions similar to the above apply for detecting violations of once only assignments during a parallel execution. Expression evaluation may be in parallel but assignments to a given variable must be performed must be performed in any sequential order so as to guarantee detection of such violations. In a sequential implementation, this difficulty does not arise.)

Note: Composite assignments where the function satisfies the USCAE function condition, bear similarity to certain features of synchronous systems [7].

6.3 Unordered sequential execution using composite assignments with several functions

Consider a more general case:

def xOut = g(…) xOut=e_0 : # initial value for xOut is e_0

xOut f_1= e_1;

...

xOut f_n= e_n;

where f_1,...f_n are functions, which may or may not be different. (They may even all be the same, which is the case discussed above.) The USCAE function condition needs to be modified as follows:

1) It is required that ((a f_i b) f_j c) = ((a f_j c) f_i b) for i≠j.

A similar argument to the above shows that the final value of xOut equals the value of (...(((e$_0$ f_1 e$_1$) f_2 e$_2$) f_3 e$_3$) ... f_n e$_n$) and is uniquely defined, subject to the other conditions enabling flexible execution. (In the sorting on the subscripts described above, the only change is to swap consecutive function expression pairs in the sort process and not just consecutive expressions.)

6.4 Allowing any composite assignment by restricting flexibility

We can allow the use of any composite assignment by restricting the flexibility. For example, the expressions on the right hand sides of the above assignment statements may be executed in parallel but the composite assignments would be executed sequentially in the order they are written. This ensures well defined read values providing all composite assignments to a variable are executed before its value is read. This is an interesting possibility as it provides a simple interface between sequential and parallel execution. Further study of this topic is needed, in particular, the efficiency of an implementation.

(Note that regular assignment is a special case of composite assignment.

Defining (x f y) = y, makes x f= y the same as x=y. Thus our remarks here apply to regular assignment too.)

7 AN EMBEDDED FLEXIBLE LANGUAGE (EFL)

We implemented an Embedded Flexible Language (EFL) in which well defined flexible execution order is implemented in EFL blocks. In the current implementation, execution in the host language is restricted to be sequential, i.e. parallelism is implemented only via EFL blocks. In future implementations we may relax this restriction. For host languages that support pointers, our first implementation of EFL does not allow pointers in EFL blocks, because this can give read and write access to the same variable, and other side-effects that should be very carefully studied. The first host language for which we implemented these EFL blocks is Python, and in the future we hope to implement EFL blocks in more host languages. We discuss EFL in detail in the companion paper [1]. For now, we only present the main features of an interface between an Embedded Flexible Language and its host languages:

7.1 Main features of an interface between embedded and host language

We envisage three subsets in an implementation of EFL and its host language interface.

Basic subset: Here there are no declarations of variables, functions or convenient extensions in EFL blocks. Declarations of variables are made in the host language only. In an EFL block, a host variable is accessed either as an IN variable or as an OUT variable, i.e. no INOUT access. Only pure functions (i.e. no side effects) of the host language may be called from within EFL blocks; it is assumed that the called functions are pure without checking this. New variables can be created by means of (recursive) function calls using call by value parameter passing. As the host language is (restricted to be) sequential, and without pointers, this will ensure well defined flexible execution in EFL blocks.

Intermediate subset: The convenient extensions such as blocks, labeled blocks and various kinds of loops would also be included but there are no function declarations in EFL blocks. (Variable declarations are part of these convenient extensions.)

Full subset: Function declarations in EFL blocks would also be included.

7.2 More on side effects

1) If a sequential read is performed, a side effect occurs. This is because the file position is changed to the next record. So, reading sequential files is not compatible with flexible execution. If there are only random access reads, where the file position is a parameter of the read, there may be a benevolent side effect and this will be compatible with flexible execution. More precisely, the file pointer is updated, but it is never read when there are only random access reads.

2) Herlihy and Shavit discuss benevolent side effects in their book [8]. Are benevolent side effects compatible with flexible execution? In our discussion of composite assignments in the section VI above, we showed that certain composite assignments, even to the same variable, must be performed in any sequential order. If a function in the host language updates a global variable only with such assignments, read values will be well defined providing that care is taken so that all updates complete before performing a read. Flexible execution is possible and the Embedded Flexible Language (EFL) which we implemented, uses this notion. We are grateful to Maurice Herlihy who made us aware of the possibility of benevolent side effects.

8 Survey Of Previous Work

We have presented an algorithmic language which supports flexible execution. Blocks declarations and conditionals are written in the style of the Python language [2]. It differs from conventional algorithmic languages in that variables are either IN or OUT but not INOUT. Composite assignments where the function satisfies the USCAE function condition may be used, of which once-only

assignment is a special case. On the other hand, pure functional languages [9, 10] completely avoid the assignment statement. If we were to write a program for reversing a vector in a functional language, at each swap a new vector would be created causing gross inefficiency. As was demonstrated above, in our approach, only one new vector was created. It also differs from other functional languages in that they are higher level languages which make more use of higher order functions. Our language, on the other hand, is more algorithmic in style.

An early work by J.L.W. Kessels [11] presents a design of a language having both sequential and nonprocedural (parallel) components. There are descriptive (nonprocedural) blocks and sequential blocks which may be freely nested. No assignments may be made to global variables in both kinds of blocks. There are also multistate blocks with associated global variables and statements for explicit synchronization. There are features similar to our approach but in ours well defined access to global variables is provided via enhanced scope rules and suitable composite assignments.

Our language is close to an early version of LUCID though LUCID later developed into a data flow language [12]. In LUCID, a well formed program cannot cause a multiple assignment, i.e. it is a compile time check, but in our language it is a run time error. As mentioned above regarding the reverse example, this can give significant execution improvements when processing vectors and other multi-component data. It also allows greater expressiveness. It is less efficient however, for handling single component data.

Our language handles multiple assignment at run time in a way similar to which the "Id-", "pH" and "SISAL" languages [13, 14, 15] handle it. Unlike the "pH" language, our language does not have mutable structures (M-structures).

There is a significant difference with logic programming languages [16, 17] in that logic programming languages support non-determinism (multiple results). Common features include once only assignment and support for unassigned (unknown) elements. Also, some logic programming languages support IN and OUT parameters.

Configuration languages [18, 19] are used for specifying the interconnections of distributed systems and our language bears similarities to configuration languages in the way in which variables and assignments are handled. We therefore expect that by adding appropriate data structures, this language can be used like configuration languages, for describing the communication links of distributed systems. However, further work is needed here. In any case, our language is a good companion to configuration languages in view of its flexible execution and implementation possibilities.

Vishkin's paper [20] describes a comprehensive approach to parallel programming. It is based on the Parallel Random Access Machine (PRAM) model for describing algorithms and eXplicit Multi Threading (XMT) approach for programming. The programmer is responsible for converting PRAM to XMT and a Work Depth (WD) methodology is described to aid the programmer do this task. A hardware implementation of PRAM is described too. Educational aspects are also discussed. It differs

from our approach in that there are no scope rules or use of suitable composite assignments to ensure a well defined result, this being the responsibility of the algorithm designer or programmer. Our approach is based on implicit parallelism instead.

In [21, 22], Thornley presents a declarative language based on the ADA programming language incorporating parallel composition, parallel for loops, and single assignment variables for synchronization. This language has an algorithmic style and there is much in common with our work. It does not include as general a construct as labeled blocks (for generalized looping). Also, it does not make use of composite assignments or the assignment style for parameter passing, or the generalized call statement.

9 WORK IN PROGRESS

Work is in progress on the following matters.

- Hardware support for flexible execution.
- Educational aspects of flexible execution.

Here we briefly describe these works and we intend to report further on these matters as these works mature.

9.1 Hardware support for flexible execution

In the previous section we saw that certain composite assignments give well defined read values when executed in any sequential order. Are there such composite assignments which are easily implementable in hardware and give well defined read values? Are there such assignments which give well defined read values when executed on suitable hardware, in parallel or sequentially in any order?

As we know, dynamic memory is capacitor based. The advantage of this type of memory is its very high density. The disadvantage of this type of memory is its need for a refresh, and because of the refresh it has a lower speed. A significant advantage of this kind of memory, is that it facilitates parallel and unordered sequential execution when used with suitable composite assignments. For more information on dynamic memory see [23].

The simplest composite assignments which can be implemented with capacitor based memory are "or=" when capacitor empty represents 0, capacitor full represents 1. Similarly, "and=" can be implemented using capacitor full to represent 0, and capacitor empty to represent 1. These composite assignments may be executed in any sequential order or even in parallel. The reason why these composite assignments give well defined read values, even when executed in parallel, is because these operations would be performed at the capacitors themselves (and because of the electrical properties of the semi-conductors in the electrical circuits).

Therefore, there is no need to read the data stored in the capacitors. It is interesting to ask if there are any other composite assignments which may be performed in parallel or in any sequential order, giving well defined read values for suitable kinds of hardware. These hardware aspects of composite assignments need further development and we intend to deal with this matter elsewhere.

9.2 Educational aspects of flexible execution

There is a need to educate/encourage programmer to write code which enables parallel execution if the full benefits of parallel execution are to be reaped. This is a consequence of Amdahl's law which states that the speedup to be gained by parallel execution is limited by the sequential part of the program.

So suppose for example that half of the program must be executed sequentially and that all the processors are identical. Then regardless of the number of processors, program execution can be speeded up by a factor of two at most. Further speedups are possible by rewriting the program and this requires educating/encouraging the programmer to write code enabling parallel execution.

In [2, 3], educational material for beginners about flexible algorithms has been reported. Further development of the educational material is desirable, for example:

- An integrated course on flexible algorithms and digital logic.
- An advanced course on flexible algorithms.
- A course for secondary schools.

The development of an educational system for teaching such materials is currently in progress. Based on the reflective approach [24, 25] to teaching and learning, that system is being developed upon the MOODLE [26] educational software platform We intend to report on this matter elsewhere.

10 CONCLUSIONS

Achieving well defined final values independent of execution order, depends on the following:

- IN parameters are assigned when calling a function, and may be read in the function's body..
- OUT parameters may be assigned in the body of a function, and their values may be read from their destination variables after exiting the function.
- There are no IN/OUT parameters.

- The function condition for Unordered Sequential Composite Assignment Execution (function condition for USCAE) enables unordered sequential execution of composite assignments to a variable and ensures a well defined final value.
- By restricting the flexibility, any composite assignment may be used and the final value of a variable is well defined. The expressions on the right hand sides of assignment statements would be executed in parallel but the composite assignments themselves would be executed sequentially based on their textual order.

We have also taken into account the following broader considerations:

- There is a need for well defined parallelism so as to simplify its use.
- There is a need for a notation not too far from imperative programming languages.
- There is a need to educate/encourage programmers to write code which enables parallel execution if the full benefits of parallel execution are to be reaped.
- It is desirable to have a notation where writing parallel algorithms is not harder than writing sequential algorithms.
- For the programmer, using implicit parallelism is simpler than using explicit parallelism. Implicit parallelism will enable more programmers to reap the benefits of parallel execution.
- Further gains are possible with hardware supporting well defined parallelism.

To sum up, our language enables well defined flexible execution and has an algorithmic style familiar to programmers and engineers. We believe that its flexibility and many faces are important in education and design.

REFERENCES

[1] "Flexible Algorithms and their Implementation: An Embedded Flexible Language (EFL)", Technical report, Jerusalem College of Technology, 2013. Available at http://flexcomp.jct.ac.il/TechnicalReports/Flexalgo&Impl_2_full.pdf

[2] "The Python Language Reference", Release 2.7 2011, https://docs.python.org/2/reference/

[3] "Flexible Algorithms: Fragments from a Beginners' Course", IEEE Distributed Systems Online, vol. 8, no. 2, 2007, art. no. 0702-o2001.

[4] "Flexible Algorithms: Overview of a Beginners' Course", R.B. Yehezkael, IEEE Distributed Systems Online, vol. 7, no. 11, 2006, art. no. 0611-oy002.

[5] "The C Programming Language", B.W. Kernigham and D.M. Ritchie, Prentice Hall 1978

[6] "The Java Language Specification, Version 1.0", J. Gosling, B. Joy and G. Steele, Addison Wesley 1996

[7] "The Esterel Synchronous Programming Language: Design, Semantics, Implementation", Gérard Berry and Georges Gonthier. Science of Computer Programming vol. 19, no. 2, pp 87-152, 1992.

[8] "The Art of Multiprocessor Programming", M. Herlihy & N. Shavit, Morgan Kaufmann, 2008.

[9] "Functional Programming: Languages Tools and Architectures", S. Eisenbach, Imperial College, London, Halsted Press: A division of John Wiley, 1987

[10] "Report on the Programming Language Haskell, A Non-strict Purely Functional Language", Paul Hudak et.al., Yale University Research Report No. YALEU/DCS/RR-777, 1st March 1992.

[11] "A Conceptual Framework for a Nonprocedural Programming Language", J.L.W. Kessels, CACM Vol. 20 No. 12 December 1977, pp.906 - 913.

[12] "LUCID, the Dataflow Programming Language", W.W. Wadge and E.A. Ashcroft, Academic Press, 1988.

[13] "Compilation of Id-: a subset of ID", Z.M. Ariola and Arvind, MIT Computer Structures Group Memo 315, revised 1 November 1990

[14] "pH Language Reference Manual, Version 1.0-preliminary", R.S. Nikhil et al., MIT Computer Structures Group Memo 369, 31 January 1995

[15] "SISAL 1.2: A Brief Introduction and Tutorial", David C. Cann, Research Report, Lawrence Livermore National Laboratory, May 1992.

[16] "Programming in Prolog", W.F. Clocksin, and C.S. Mellish, Springer Verlag, 2nd edition 1984.

[17] "Concurrent Prolog - Collected Papers Vols 1,2" edited by Ehud Shapiro, MIT Press, 1987.

[18] "An Introduction to Distributed Programming in REX", J. Kramer, J. Magee, M. Sloman, N. Dulay, S.C. Cheung, S. Crane, and K. Twiddle, in "Proceedings of Esprit, Brussels, 1991.

[19] "Structuring Parallel and Distributed Programs", J. Magee, N. Dulay, and J. Kramer, in "Proceedings of the International Workshop on Configurable Distributed Systems, London, 1992.

[20] "Using Simple Abstraction to Reinvent Computing for Parallelism", U. Vishkin, CACM Vol. 50 No. 1 January 2011.

[21] "Integrating parallel dataflow programming with the Ada tasking model", J. Thornley, TRI-Ada '94, Proc. conference on TRI-Ada, ACM 1994, pp. 417-428.

[22] "Declarative Ada: parallel dataflow programming in a familiar context", J. Thornley, CSC '95, Proc. 23rd annual conf. on Computer science, ACM 1995, pp. 73-80.

[23] "An Embedded DRAM for CMOS ASICs", J. Poulton, Proceedings of the 17th Conference on Advanced Research in VLSI, IEEE Computer Society Press, Sept 15-16, 1997, pp. 288-302. PDF version of paper available at http://www.cs.unc.edu/~jp/DRAM.pdf

[24] "Educating the reflective practitioner: Toward a new design for teaching and learning in the professions", D.A. Schon, Jossey Bsss - Wiley 1990.

[25] "On the Epistemology of Reflective Practice", M. van Manen, Teachers and Teaching: theory and practice, vol. 1, nr. 1, pp 33-50, 1995.

[26] "MOODLE", see the website moodle.org