Conf-9104425--1

*Proceedings of the Fifth International Parallel Processing Symposium Anaheim, California, April 30 - May 2, 1991*

# Input/Output Operations for Hybrid Data-flow/Control-Flow Systems*

**Paraskevas Evripidou**
Dept. of Computer Science and Engineering
Southern Methodist University
Dallas, Texas 75275

**Jean-Luc Gaudiot**
Dept. of Electrical Engineering–Systems
University of Southern California
Los Angeles, California 90089

## Abstract

Hybrid data-flow/control-flow systems combine the advantages of the data-flow model: functionality and tolerance to communication and memory latencies with the efficient instruction scheduling of the control-flow model. The absence of global state in such hybrid multiprocessors and multiprocessors in general renders the implementation of state tasks such as Input/Output operations very difficult to implement. A distributed file-pointer scheme for incorporating I/O operations onto the data-flow model has been developed. A dependency detection algorithm detects and classify cases of potential access conflicts. A conflict resolution data-flow graph is then created which at execution time safely distributes file-pointers to the I/O actors. This scheme has also been implemented on a hybrid a data-flow control-flow multiprocessor: the Decoupled Data-Driven Multiprocessor with Variable Resolution Actors.

## 1 Introduction

While a single host attached to a multiprocessor could easily serve as an interface to the outside world, timely distribution of the data may not be assured to all the processors and thus create a bottleneck (Amdahl's Law). Instead, one must assume that the I/O operations will be equally distributed and that all (or a subset) of the processors will be involved in communicating with the I/O devices. Consequently, individual I/O operations are distributed, we must also insure proper synchronization among them.

Traditionally, data-flow researchers and designers of functional languages have been faced with two diametrically opposed goals[1]:

- *Semantic simplicity*: I/O should be treated as primitives with the same functional semantics as all operations.
- *Programmer convenience*: I/O operations should be introduced as primitives with side-effects.

I/O operations have been introduced as "state" operations in the von Neumann environment as a convenient and clean way of injecting new data into a program without recompiling it. In other words, the very absence of side-effects that makes it very convenient to specify computational operations also makes I/O very difficult to implement in a functional environment. Therefore, introducing I/O instructions either in a pure functional way or as primitives with "side-effects" requires some kind of retreating from these opposing environments. A *hybrid* approach has been developed for incorporating I/O operations at the data-flow graph level [2]. The logical order for executing the I/O operations of common source/destination is the sequential ordering given by their lexicographical ordering in the source program. This compromises some of the semantic simplicity of the *"pure approach"* with some of the convenience of the *"convenient approach"*.

In this paper we provide an overview of the overall I/O scheme for dynamic data-flow machines (section 2). An in depth analysis of the output value processing in dynamic data-flow machines is presented in section 3. Section 4 presents the incorporation of I/O in the Decoupled Data-Driven Architecture.

## 2 I/O in the Data-Flow Model

The two basic I/O instructions:

$x =$ **read**$(filename)$, and
**write**$(filename, value)$

are introduced at the data-flow graph level as primitives with side-effects, i.e., the file pointer is incremented after each access. Actually in our optimized scheme, parallel I/O operations are made possible by using multiple file pointers. The file pointer always points to the file location to be read or written. Consider the following simple program.

*This material is based upon work supported in part by the U.S. Department of Energy, Office of Energy Research under Grant No. DE-FG03-87ER25043

```
function Simple_IO(InFile, OutFile : file returns ...)
  a := read(DataFile);
  b := if a <> 0 then read(InFile,A);
                else 0;
  c := read(InFile);
    d := b + c ;
      write(" The Sum is");
      write(OutFile, d);
      write(OutFile, "end");
end function
```

This program contains six I/O instructions. Notice that the file names are available to all I/O instructions at the same time. Therefore, sequencing by data availability as in "conventional" data-flow operations is not an option any longer. In the *Simple_IO* example there are numerous different combinations in which the reads and writes can occur. Obviously, this non-deterministic behavior is not acceptable for most practical cases.

We have developed a dependency detection algorithm which identifies all I/O dependencies in a program. This algorithm produces a Dependency Trace which indicates the *logical ordering* of the I/O instructions. This dependency trace is used to create the I/O Access Graphs. In the *simple_IO example* the dependency trace gives the logical order in which the IO instructions should be executed according to their lexicographical ordering in the source program:

Our approach improves on the complete sequentialization of I/O-performing functions by superimposing I/O dependencies and related operations as a separate graph (Input Access Graph (IAG) in the case of input processing; Output Access Graph (OAG) in the case of output processing). Thus, it constrains the main computation graph only in a minimum way. necessary.

Figure 1 depicts the data-flow graph and the IAG and OAG for the *simple_IO* example. As shown in Figure 1 each I/O access graph provides conflict resolution for the access of the corresponding I/O files. The purpose of the Input Access Graph is to synchronize the read accesses from a shared file. In a dynamic data-flow environment, this amounts to extracting a value from the input file, tagging it, sending it to its consumer, and finally incrementing the file pointer. An in-depth analysis of the input case has been presented in [2]. The output case is presented next.

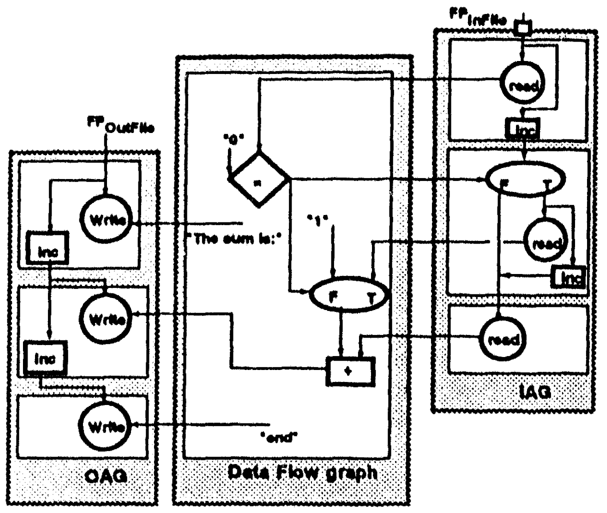


Figure 1: Data-flow graph for function simple_IO with the IAG for file Infile and OAG for file OutFile

## 3 Output Processing

Inputing values from a file into a dynamic data-flow graph is the process of determining the correct tag for an input value. The tag associated with each value is determined by (in addition to other factors), the position of the input value in the I/O file. In the output case, it is exactly the opposite. The output value has a tag associated with it. This tag will determine the position of the value in the output file.

The OAG is the interface between the Computational subgraph of a program and the file-service facility of the operating system. The OAG has a write actor associated with each write instruction in the program. The communication between the various stages of the OAG is performed by passing pointers. Stage $i$ passes a pointer to stage $i+1$. This pointer contains the file location at which stage $i+1$ starts writing in the output file. Thus, a distributed i .e-pointer scheme is employed, which in turn provides for the parallel execution of the I/O operations.

The Output Access Graph is constructed by a combination of standard dynamic data-flow actors (U-Interpreter [3] and Token Relabeling [4]) with the addition of the write actor. Some logical modules that consists of various basic actors are also defined.

The write actor $W$ is the basic output actor. It receives two inputs: a) the value to be written, and b) the file-pointer (file pointer is the 2-tuple of file name and offset within the file). It places an output value in the designated output file. There are four basic output constructs: simple write, write inside a *forall loop(s)*, conditional write, write inside a *repeat-until loop(s)*, and finally the synthesis of a number of the four basic constructs. The description of the OAG construction will be demonstrated with the aid of the function *write_matrix* shown in the following:

```
function write_matrix(...)
    if flag then write(infile,n);
        for i in  1,n cross j in i,k
            write(outfile, a[i,j])
        returns file  outfile
        end for
end function
```

The notation used in the data-flow graph used in this papers are as follows. The token format is $V_{[c.s.i]}$ where $V$ is the data value, and $_{[c.s.i]}$ is the tag. The first part of the tag $c$ is the context identifier, the second field $s$ is the destination address, and $i$ is the iteration identifier (used for loops). The following notation will be used to describe the actors used in the rest of this paper:

$$opcode : \text{input1}_{[c.s.i]}, \cdots, \text{input2}_{[c.s.i]}$$
$$\rightarrow \text{output}_{[c.s_1.i]}, \cdots, \text{output}_{[c.s_n.i]}$$

which means that the actor "*opcode*" receives as arguments the tokens $\text{input1}_{[c.s.i]}, \cdots, \text{input2}_{[c.s.i]}$ and produces the output token $\text{output}_{[c.s_j.i]}$, which sends to its $n$ consumers: $s_1, \cdots, s_n$.

## Simple Write

This is the simplest form of a write construct; it corresponds to a single write instruction. The stage of the OAG corresponding to a simple write is consists of a write actor $W$, a file-pointer increment module, and possibly a tag relabeling module. The latter is needed to manipulate the tag of the file-pointer so that it matches the tag of the output value. The three write actors in the OAG of Figure 1 are examples of simple write.

## Write inside a *Forall* loop

Nested loops (*Forall loops*) is a very common way to output data. The output values are arriving at the OAG, possibly out of order. Therefore, the *Forall stage* has to determine the order of creation of the values in order to place them in their corresponding position in the output file. To do that, it is important to observe that the iteration identifiers of the tag create a form of multi-base number system: the base of position $i$ is the maximum range of position $i+1$. Therefore, the order of creation of token $data_{[\ldots[[u.c.s.i_1].c.s.i_2]\ldots].c.s.i_n]}$ is given by:[1]

$$i_n + \sum_{k=n-1}^{1} (i_k - 1) * r_k + 1 \qquad (1)$$

[1]Formula 1 assumes that the tag generation starts at 1, not 0.

where $r_k$ is the max range at $k^{th}$ level

To perform the mapping described by equation 1, it is necessary to know the maximum range in each level of nesting. This is done by the $LR^n$ module where $n$ is the degree of nesting. The $LR^n$ module takes as inputs the maximum range of each of the nested loops. It then calculates the combined maximum range by multiplying the range of each loop. It also records the range of each level of nesting in the iteration field of the tag as shown below.

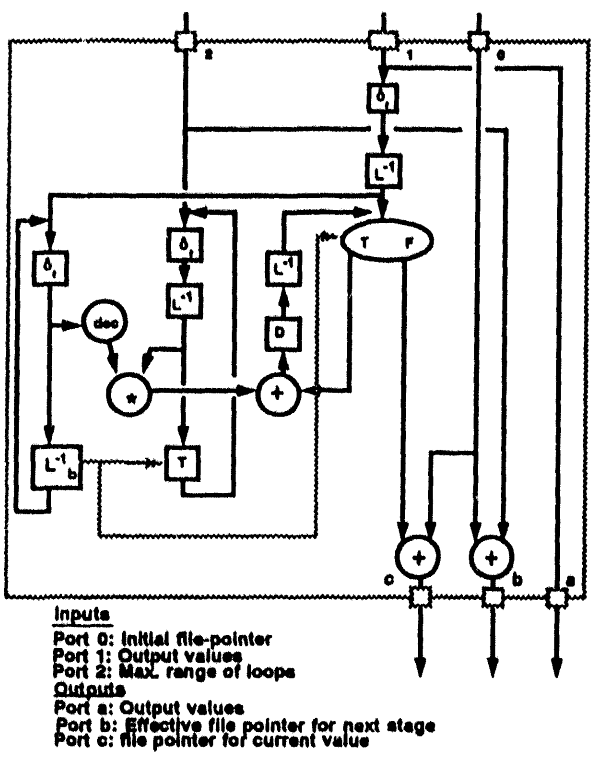
$$LR^n : r1_{[u.c.s.1]_0}, r2_{[u.c.s.1]_1}, \cdots, rn_{[u.c.s.1]_n}$$
$$\rightarrow r1 * r2 * \ldots * rn_{[\ldots[[u.c.s.r1].c.s.r2]\ldots.c.s.rn]}$$

The mapping of equation 1 is done by the $W_{for}$ module. The $W_{for}$ module takes as input the maximum range of the nested loops created by an $LR^n$ module, the output value and the initial file-pointer. The $W_{for}$ module is shown in Figure 2. A new actor, $L_B^{-1}$, is introduced which is similar to the $L^{-1}$ actor, in addition it produces a Boolean true value each time it unstacks the context part of the tag. However, when there is no stacked context to unstack, it sends a boolean false value to port b and nothing to port a:

$$L_B^{-1} : data_{[[u.c.s.i].c'.s'.i']_0}$$
$$\rightarrow \begin{cases} (data_{[u.c.s.j]})_a & , True_{[u.c.s_B.i]_b} \\ \emptyset_a & , False_{[u.c.s_B.i]_b} \end{cases}$$

The file-pointer is computed and sent to the next stage as soon as all required data becomes available. In short, the stage of the OAG corresponding to a write in nested loops is composed of a $LR^n$ module in series with a $W_{for}$ module. Function *write_matrix* presents such a situation–write inside two nested loops. Stage 2 of Figure 4 depicts this configuration. First an $LR^2$ module is needed which takes as inputs the maximum range of the two nested loops ( $2_{[c.0]}$ and $3_{[c.0]}$) and creates the maximum combined range ($6_{[[c.2].3]}$). When a token arrives at the $W_{for}$ module, it uses its tag and the combined range it got from the $LR^2$ unit to determine the effective file pointer for that value. For example, when the token $25_{[[c.1].3]}$ arrives at the $W_{for}$ module the latter calculates that its relative position is 3 and the initial file pointer (one in this case ) has to be added to this value. Thus the effective file-pointer for the value $25_{[[c.1].3]}$ is $4_{[[c.1].3]}$. Both tokens are sent to the write actor, which performs the actual write. Note that as soon as both the initial file pointer $1_{[*]}$ and the maximum combined range of the two loops ($6_{[*]}$) arrive at the $W_{for}$ , the effective file-pointer of the next stage is calculated as $7_{[*]}$[2] and passed to the next stage if one exists.

[2]As mentioned earlier, it is beyond the scope of this paper to



2
1
0
$\delta_i$
$L^{-1}$
T F
$\delta_i$
dec
$\delta_i$
$L^{-1}$
$L^{-1}$
D
*
+
$L^{-1}_b$
T
+
+
c
b
a
Inputs
Port 0: Initial file-pointer
Port 1: Output values
Port 2: Max. range of loops
Outputs
Port a: Output values
Port b: Effective file pointer for next stage
Port c: file pointer for current value

Figure 2: The write $W_{for}$ module

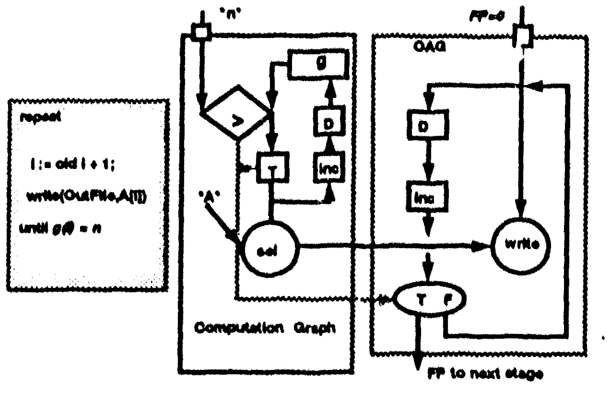


Figure 3: Outputing values with repeat until construct

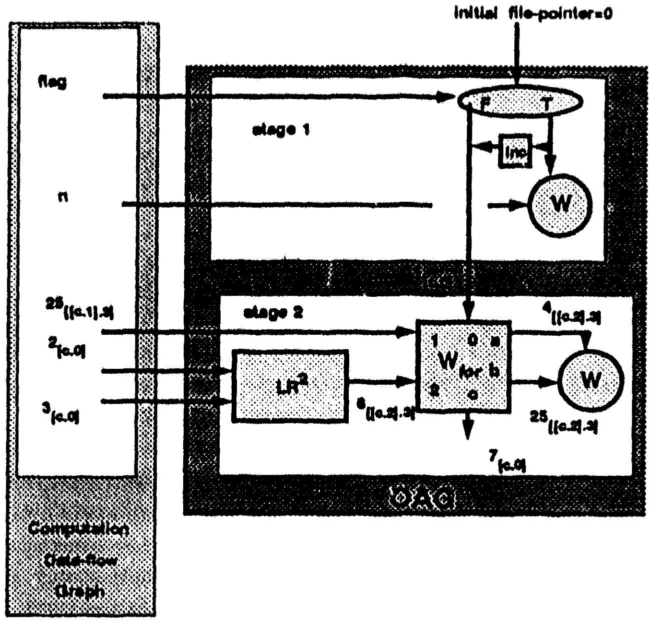


Figure 4: The OAG for function *write_matrix*

## Conditional Write

Conditional write is very similar to conditional read (stage 2 of the IAG of Figure 1). A switch actor is needed to gate the value of the file-pointer to the conditional read stage if the condition is satisfied, or to the next stage if the condition is not satisfied. However, the output values need not be gated because if the condition evaluates to false, then the computation subgraph will not send the output values to the OAG.

## Write inside a *repeat-until* loop(s)

This case is very similar to its read counterpart. A file-pointer increment module is needed to recirculate the file pointer until the condition evaluate to false. As in the conditional write case, it is not necessary to gate the output values. Figure 3 depicts the case of write inside a repeat until loop.

## Synthesis

It is possible to have compound output construct by combining the four basic constructs. This is the case in function *write_matrix* where both a conditional and a forall loop are used. In general, the corresponding modules are connected in the order the various constructs appear in the dependency trace. A general mapping algorithm from the dependency trace to the IAG is given in [5]. The mapping algorithm will not be presented here however, the various examples presented here give the flavor of how the mapping is performed. The OAG for the *write_martix* function is shown in Figure 4. Each write instruction in the program corresponds to one line in the dependency traces which in turn corresponds to a stage in the OAG. It should be clear by now from our examples that the creation of both the IAG and OAG is very similar to the creation of a dynamic data-flow graph. Both the IAG and OAG are observing the *execution-upon-data-availability-principle*.

discuss the tag-relabeling techniques. Therefore, the wild-card * character is used to denote that correct matching occurs.

## 4 I/O Operations on the Decoupled Data-Driven Architecture

The *Decoupled Data-Driven Architecture with Variable Resolution Actors* is a hybrid data-driven control driven architecture with decoupled graph and computation units [6]. Each actors is partitioned (decoupled) into two parts: graph and computation portion. The computation portion of each actor is a collection of conventional instructions (load/store, add, etc). The graph portion contains information about the executability of the actor and its consumers. Thus, a decoupled data-driven graph can be viewed as a conventional program with a data-dependency graph superimposed on it.

A decoupled processor, (depicted in Figure 5) has two engines: the Data-flow Graph Engine (DFGE) and the Computational Engine (CE). The DFGE executes all graph operations (determine executability) and the CE executes all computation operations (code fetching and execution). The two Engines execute in an asynchronous manner, i.e., the computation unit does not have to execute the computation portions of actors in the same order as the graph unit executes the graph portions. During execution the DFGE places all ready actors in the Ready Queue (RQ). The CE removes one actor at a time from the RQ and executes all instructions related with that actor. The instruction set of the CE is RISC-like. The CE performs load and stores to/from the Computation Memory (CM) (the computation cache actually). When the CE completes the execution of an actor it places its identification in the Acknowledgment Queue (AQ). The DFGE has its own hierarchical memory: Graph Memory (GM) and Graph Cache (GC). The DFGE removes one actor at a time from the AQ and updates the status of its consumer actors. During this process all executable actors are placed in the RQ.

The decoupled multiprocessor retains the dynamic data-flow principles of execution at the coarser level and employs control-flow principles at the fine level (within a macro actor). Three types of actors are supported: (I) Scalar actors, (II) Vector and Macro actors, and (III) Compound Actors (CMAs): Compiler-generated collections of scalar and/or vector instructions.

The modes of operation of the decoupled architecture and its handling of the I/O operations are described with the aid of the function *simple_IO*.

Each actor is represented by a graph subtemplate and a computation subtemplate. Figure 6 depicts the decoupled implementation of the *simple_IO* function. A load/store instruction set is used. In the figure the

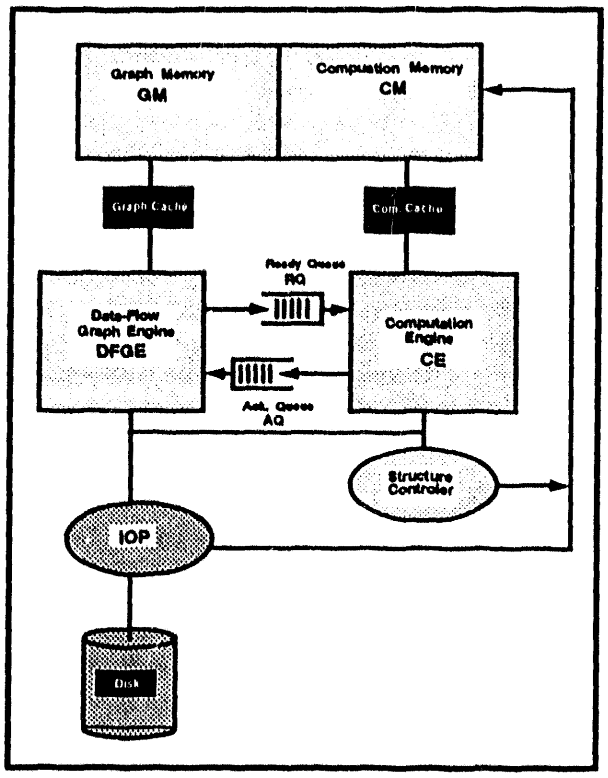
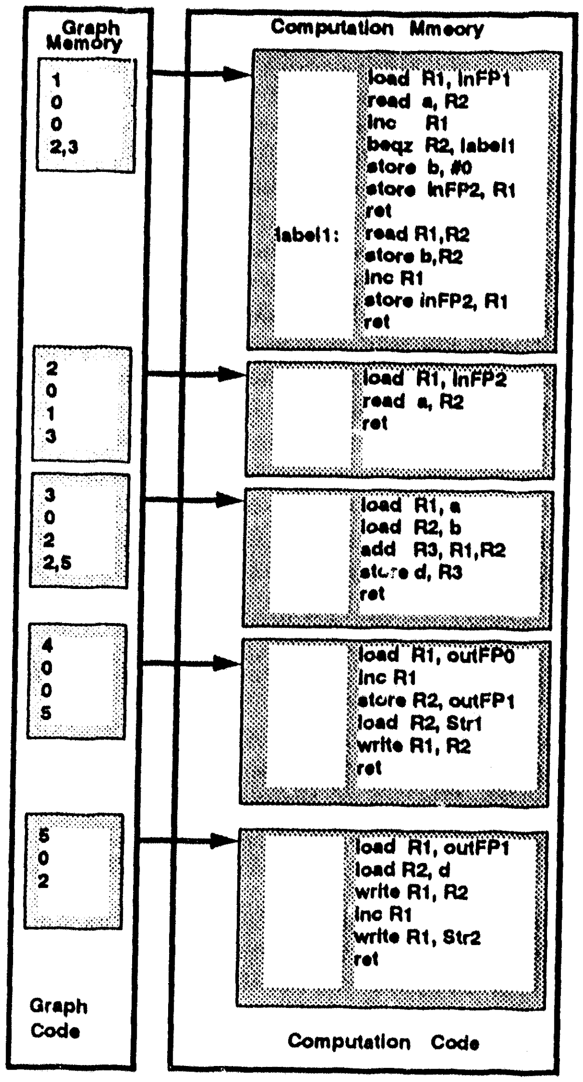


Figure 5: A Decoupled Data-Driven Processor

variables *a*, *b*, *c*, *InFP1*, *InFP2*, *OutFP1*, *OutFP2* are all memory locations. *Str1* and *Str2* are the address of the strings " the Sum is" and "end" respectively. The instructions:

*write file-pointer, value* and
*read Reg, file-pointer*

are the basic write and read instructions. The first field of the graph subtemplate is the actor ID, the second field is the context (context is 0 in this example), the third field is the status word (number of input tokens), and the last field is the consumer list. The first actor performs the first two instructions of the function ( a:= read(...) and b: = if ...). It first reads the first value from the InFile by using its initial file-pointer (load R1, InFP1) and then according to this value it sets *b* equal to zero or it reads a second value from the file. It also increments the value of the file pointer and stores it in memory location *InFp2* that is read by actor 2. Actor 2 is a scalar actor since it performs only one operation (c:= read(...) ), actor 1 on the other hand is a macro actor because it performs more than one operation. Similarly, actors 3 and 4 are scalars and actor 5 is a macro actor. The CE executes all instructions in actor and then it places its ID in the AQ. The DFGE will fetch this ID from the AQ and will update the status word of its con-



```
          load  R1, InFP1
          read  a, R2
          inc   R1
          beqz  R2, label1
          store b, #0
          store InFP2, R1
          ret
label1:   read R1,R2
          store b,R2
          inc R1
          store inFP2, R1
          ret
```

```
          load  R1, InFP2
          read  a, R2
          ret
```

```
          load  R1, a
          load  R2, b
          add   R3, R1,R2
          store d, R3
          ret
```

```
          load  R1, outFP0
          inc R1
          store R2, outFP1
          load  R2, Str1
          write R1, R2
          ret
```

```
          load  R1, outFP1
          load R2, d
          write R1, R2
          inc R1
          write R1, Str2
          ret
```

Figure 6: Graph and Computation code for the *Sim-ple_IO* example

sumers (decrement it by 1). For example after actor 1 is executed the DFGE will decrement the status of its consumer actors 2 and 3. When the status word of an actor reaches zero the actor is executable and its ID is placed in the RQ. The whole process continues until all actors are executed.

The example described here has only one context. The decoupled architecture uses dynamic data-frames for the implementation of multiple instantiations of an actor (multiple contextes). Each dynamic instance of an actor loads its operands form and stores its results into unique memory locations. Interested readers may refer to [6] for more details on this.

## 5 Concluding Remarks

A conflict resolution scheme has been developed that allows the incorporation of general purpose Input Output operations in the data-flow execution model. I/O operations are ordered according to their lexicographical ordering in the source program. Their execution however, can be performed in parallel since multiple instances of the file-pointer are allowed. Each instances of the file pointer is created and tagged according to the dynamic data-flow principles of execution. An in-depth analysis of the output case has been presented. The I/O schemes presented here can be incorporated in conventional multiprocessors if their process naming is used to emulate the data-flow tagging. We have also described the incorporation of general purpose I/O operations into the "Decoupled Data-Driven Architecture with Variable resolution Actors."

## References

[1] J.H. Williams and E.L. Wimmers. Sacrificing simplicity for convenience: Where do you draw the line? In *Proceedings of the* 18th *Annual ACM SIGART-SIGPLAN Symposium of Programming Languages*, Jan. 1988.

[2] P. Evripidou and J-L. Gaudiot. Distributed Input/Output Processing in Data-Driven Multiprocessors. In *Proceedings of the Second IEEE Symposium on Parallel and Distributed Processing*, Dec. 1990.

[3] Arvind and K.P. Gostelow. The U-Interpreter. *IEEE Computer*, pages 42–49, February 1982.

[4] J.-L. Gaudiot and Y.H Wei. Token relabeling in a tagged token Data-Flow architecture. *IEEE Transactions on Computers*, 38(9), Sept. 1989.

[5] P. Evripidou and J.-L. Gaudiot. Input/output operations in a data-driven environment. Technical Report CRI 88-45, University of Southern California, Dept. Electrical Engineering-Systems, Computer Engineering, July 1988.

[6] P. Evripidou and J-L. Gaudiot. A Decoupled Graph/Computation Data-Driven Architecture with Variable Resolution Actors. In *Proceedings of the 1990 International Conference on Parallel Processing*, August 1990.

END

DATE
FILMED

12/6/93