1 of 1

Distributed Input/Output Processing in Data-Driven Mutlprocessors*

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

Data-flow principles of execution provide an elegant way to ensure at runtime that instructions can be executed asynchronously in a parallel environment. However, while the conventional von Neumann model of interpretation has a very rigid ordering of instructions, it is the very asynchronous character of the data-flow model of execution that introduces conflicts when "state" tasks (such as I/O operations) must share common data objects. In order to execute I/O operations safely and in parallel, an algorithm to detect and classify cases of potential conflicts (hazards) has been developed; it is described in this paper. It is based upon localizing the effect of I/O operations by splitting the data-flow graph into two subgraphs: a) the computation subgraph, and b) the I/O subgraph. The scheme presented in this paper thus enables the creation and interaction of both subgraphs, which in turn yields a deterministic execution. Furthermore, the proposed scheme enables the distributed execution of I/O operations as permitted by data dependencies.

1 Introduction

Data-flow principles of execution provide an elegant way to synchronize many parallel processes. The "execution-upon-data-availability" mechanism exhibits tolerance to synchronization and memory access latencies[1]. This makes data-flow very suitable as an execution model for distributed multiprocessor systems where large latencies are unavoidable. Indeed, data-flow execution, coupled with the functional programming style, provides a side-effect-free environment. However, the lack of an internal state renders the implementation of Input/Output (I/O) op-

erations highly complicated. Furthermore, the situation is more complex when various program segments share an I/O source, e.g., an input file. The von Neumann model of execution is very liberal in its treatment of I/O. It allows virtually any combination of accessing and sharing of I/O objects. No complete and convenient model for state operations (such as stream manipulation, efficient distributed managers calls, or I/O) has yet been developed for the data-flow model of execution that has been already remarked in [2]. In order for the data-flow model of execution to be considered a serious alternate choice to the von Neumann model for general purpose computing, it must allow similar treatment of I/O[3]. However, the I/O synchronization problem is not unique to the data-flow model. Conventional multiprocessors have similar problems. Indeed, if multiple instances of an I/O-performing process are allowed to co-exist in machine, it is very difficult to synchronize file accesses.

The goal of this paper is to solve the problem of inputting (outputting) data values so that they can be injected (extracted) to (from) the data-flow program graph. In order to keep our multiprocessor busy, it is clear that an adequate supply of input data must reach the processors. While a single host attached to our multiprocessor could easily serve as an interface to the outside world, timely distribution of the data tokens may not be assured to all the processors. Instead, one must assume that the I/O function will be equally distributed and that all (or a subset) of the processors will be involved in communicating with the I/O devices. Consequently, this means that when individual I/O operations are distributed, we must also insure proper synchronization among them. Sequencing by data availability as in "conventional" data-flow operations is not an option any longer, for a file read operation requires only knowing the name of the file. This name is already available to all read actors concurrently, which renders them executable simultaneously regardless of any intended precedence. Therefore, additional synchronization mechanisms must be

*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 and by the Defense Advanced Research Project Agency via NASA Cooperative Agreement No. NCC 2-539

DISCLAIMER

This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor any agency thereof, nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof.

provided to ensure appropriate data allocation. As a solution we propose introducing the basic I/O instructions **read** and **write** as history sensitive primitives, i.e. the file-pointer is incremented after each access. The introduction of operations with side-effects into the data-driven environment necessitates an ordering of these instructions. I/O instructions that access the same file will be ordered by their lexicographic order. This sequential ordering or serialization of I/O operations has been also independently proposed for the *ld* compiler at MIT [2] and for the RMIT/CSIRO project [4].

In our approach, however, the effect of I/O is isolated and localized by splitting the data-flow graph into two subgraphs: a) the *computation subgraph* which solely consists of all actors not involved in the I/O process, and b) the *I/O subgraph* which performs all the actual I/O operations. The techniques presented thus not only allow for deterministic execution of I/O operations but also allow for their parallel/distributed execution. In fact, an optimal level of parallelism can be achieved as allowed by data dependencies. Although the techniques described here use files as the basic I/O unit, they can also be applied to streams in order to enable distributed stream processing and communications outside the tagging system.

Section 2 states the I/O problem in an asynchronous data-driven multiprocessor. Section 3 presents, our dependency detection algorithm. Section 4 presents the *Input Access Graph* (IAG) construction. Concluding remarks and future research directions are discussed in Section 6.

2 I/O in the Data-flow model

The conventional notion of I/O is not compatible with data-flow environments in particular and the functional style of programming in general. Therefore, incorporating I/O operations in the data-flow model is a tradeoff between functional principles and "convenient" (presence of side-effects) I/O operations. In the scheme presented here, the basic I/O instructions:

$x = \text{read}(\text{filename}),$ and
 $\text{write}(\text{filename}, \text{value})$

are introduced as primitives with side-effects, i.e., the file pointer is incremented after each access. The file pointer always points to the file location to be read or written. The following function "simple_io" written in a variation of SISAL [5] illustrates the use of these two I/O extensions.

function simple_io(infile : file returns integer)

```

let
  flag := read(infile);
  x := if (flag == 1) then read(infile);
        else 1;
  y := read(infile);
in
  x - y
end let
end function

infile: 1 10 20

```

The data-flow graph for this function is depicted in Figure 1. Notice that the input file *infile* is available to all three *read* actors at the same time. The data dependencies between the "first" read (actor 1 of Figure 1) and the "second" read (actor 4 of Figure 1) precludes them from being executed out of order. However, there are no dependencies between the "third" read (actor 6 of Figure 1) and the other two. There are thus three possible orders in which the three reads can be performed:

$\text{flag} := 1$	$X = 10$	$Y = 20$	$\text{simple_IO} := -10$
$\text{flag} := 1$	$X = 20$	$Y = 10$	$\text{simple_IO} := 10$
$\text{flag} := 10$	$X = 1$	$Y = 1$	$\text{simple_IO} := 0$

Figure 1: Data-flow graph of function "simple_io"

Obviously, this non-deterministic behavior is not acceptable for most practical cases. Thus, some total ordering on the I/O operations must be introduced. The introduction of I/O operations as primitives with "side-effects" now makes the potential for hazards and

paces real as demonstrated with the example of the function *simple_io*.

In the remainder of this chapter, more details on the issues involved in introducing I/O operations in dynamic data-flow architectures will be presented.

2.1 I / O Hazards

Two kinds of hazards can occur in an asynchronous data-flow environment augmented with the I/O extensions described earlier: *Read out of order* (RO) from an I/O file, and *Write out of order* (WO) to an I/O file. The necessary condition for either hazard is that at least one I/O file must be shared. This, however, may not be a sufficient condition.

The *read out of order* condition corresponds to different values to be read from or written to a common I/O file. Similarly, a *write out of order* condition occurs when values are written in an I/O device in an order other than the one intended by the programmer.

Sharing an I/O file occurs in two different cases:

1. Sharing of an I/O file by instructions of different program segments: The I/O dependencies that result from this situation are called *Shared by Different program Segments* (SDS) (static dependencies).
2. Sharing of an I/O file by different instantiations of the same instruction: The dependencies resulting from this condition are called *Shared by Different instruction Instantiations* (SDI). SDI dependencies are found in loop bodies and can also result from multiple invocations of the same function (dynamic dependencies).

2.2 Sequential Ordering

The basic principles of data-flow execution dictate that instruction executability must be solely determined by the availability of the arguments of an instruction. This makes the relative position of instructions in the source code totally irrelevant as far as the *meaning* of the program is concerned. On the other hand, this is not the case in the conventional von Neumann environment: the executability of an instruction is dictated by its relative position in the source code. This ordering is strictly enforced by the program counter(s). The same is obviously true for I/O tasks: various program segments are allowed access to shared I/O data in the order given by the program counter. Therefore, in a von Neumann environment, treating I/O instructions as any other instruction guarantees deterministic behavior. Yet, data-flow principles cannot directly guarantee this deterministic treatment of

I/O because of the asynchronous execution of operations.

As discussed above, an I/O access protocol must be established. The most natural solution is a protocol which will allow the same freedom in the I/O access patterns as in the von Neumann environment. This notion is formalized in the concept of *sequential ordering of the I/O protocol*:

The I/O instructions of common source/destination should be executed in the same relative order as they would be in a single sequential processor (reduction order is left to right).

In other words, the I/O instructions should be performed in the order in which they appear in the program (left to right). However, ordering need not be applied to function invocations and instructions not directly related to I/O. The sequential ordering protocol gives the programmer the ability and responsibility of synchronizing the sharing of common I/O files. This now brings the data-flow model of execution directly in line with the von Neumann model of execution as far as the I/O access pattern is concerned.

One way to implement the sequential ordering in the data-flow environment is by introducing data dependencies, i.e., by changing the file name each time the file is updated. This is however very tedious. It is especially difficult to synchronize the I/O accesses in the presence of conditionals. For example, the function *simple_io* can be rewritten as *simple_io_2* by explicitly introducing data dependencies:

```
function simple_io_2(infile : file returns integer)
  let
    flag, infile2 := read(infile);
    x ,infile3 := if (flag == 1) then read(infile2);
                  else 1, infile2;
    y := read(infile3);
  in
    x - y
  end let
end function
```

It is clear that although implementing this method is feasible, it is very cumbersome and tedious for the programmer to provide explicit synchronization. In essence, it is similar to the explicit synchronization required in imperative languages, a practice which makes writing parallel programs tedious and limits the portability of such programs. As we shall see later, a better approach (our scheme) would be to have the compiler introduce the data dependencies. Furthermore, manually introducing data dependencies (flag,

`infile2 := read (infile) )` would not suffice in the case of I/O inside loops. Consider for example the function `input_vctr`, which (as the name implies) reads a vector of size `n` from the input file `infile`.

```
function input_vctr(infile: file; n integer
    returns onedim)
    for i in 1,n
        x = read(infile)
    returns array of x
    end for
end function %input_vctr
```

Introducing data dependencies in the form of changing the file name is not feasible in this case because file names would need to change dynamically for each instance of the loop.

3 Dependency detection

We demonstrated how to synchronize I/O operations by introducing data dependencies. However, this approach does not offer any parallelism and it will eventually lower the performance of any I/O-intensive program. An I/O scheme should deliver as much parallelism as possible (as allowed by the true data dependencies). The processing of an I/O value can be summarized as: determining the correct tag for an input value, tagging the input value, and sending it to its consumer. Associating a tag to an input value is very similar to Token Relabeling techniques [6]. The tag associated with an input value is determined by the context and the iteration to which the consumer actors belong. Therefore, one way to parallelize the I/O operations is to determine what portion of an I/O file an actor consumes and to process each portion in parallel. Remember that in the example function `input_vectors`, four instructions share the file `infile`. After the first value is input (`n`), it is possible to determine which portion of the file belongs to each read instruction depending on the value of the conditional `flag==1`. Therefore, different portions of the file can be processed in parallel. This is achieved by distributing “file-pointers” to the various I/O accessing constructs.

The optimization presented here is a combination of compile and execution time techniques. The first part of the process is the dependency detection. The second part is the construction of the I/O subgraph. All I/O operations are performed by the I/O subgraph. The dependency detection is a compile time process. Its purpose is twofold:

1. Identify I/O dependencies present in the dynamic data-flow program and

2. Collect information relative to the I/O instructions needed for the creation of the Input/Output Access Graph.

If no I/O instruction is part of a conditional block, then the SDS dependencies can be completely resolved at compile time. The presence of I/O instructions inside conditional blocks makes the SDS dependencies detection and resolution a combination of compile time and runtime mechanisms. Henceforth, the term *Conditional I/O* will refer to the case when I/O instructions belong implicitly or explicitly to conditional blocks. An I/O instruction implicitly belongs to a conditional block when it appears in a function that is invoked from a conditional block. The function `input_vectors` presented in Section 2 is rewritten without any explicit dependencies, and it will be used as the example to illustrate the dependency detection algorithm.

```
function input_vectors(...)
let
    n := read(infile);
    flag := read(infile);
    a:= if (flag==1) then input_vctr(infile,n);
        else for i in 1,n
            returns array of 0;
        end for,
    b:= input_vctr(infile,n);
in
    ...
end function %input_vectors
```

The algorithm operates in three passes: **Bla Bla**

Dependency Detection Algorithm

First Pass:

Perform the *lexico-graphical analysis* for the graph, following the function invocations. Whenever an I/O object is accessed, record the function name and instruction offset.

Second Pass:

For all function invocations in the trace of the *First Pass*, follow the nested functions invocations, if any, and record them to the trace. Finally, record to the trace the actual I/O instruction.

Third Pass:

For all instructions on the SDS trace, determine whether they belong to a loop or a conditional block. If they belong to:

- (a) a FORALL loop, mark them "For." Also mark their range and position.
- (b) a REPEAT-UNTIL block, mark them "Repeat" and record their position.
- (c) an IF block mark them "IF" and record the position of the IF statement. Also indicate the branch it belongs to (true (T) or false (F)).

Apply *Third Pass* recursively until it produces no changes to the trace.

By applying the dependency detection algorithm to function *input_vectors* presented earlier, the following trace is produced (for clarity, the function name and line within the function are used to identify the instructions involved in the I/O operations):

First Pass:

- 1: (SDS *input_vectors*-2)
- 2: (SDS *input_vectors*-3)
- 3: (SDS *input_vectors*-4)
- 4: (SDS *input_vectors*-6)

Second Pass:

- 1: (SDS *input_vectors*-2)
- 2: (SDS *input_vectors*-3)
- 3: (SDS *input_vectors*-4 $\rightarrow$ *input_vctr*-2)
- 4: (SDS *input_vectors*-6 $\rightarrow$ *input_vctr*-2)

Third Pass:

- 1: (SDS *input_vectors*-2)
- 2: (SDS *input_vectors*-3)
- 3: (SDS *input_vectors*-4 { If T }-4 $\rightarrow$ *input_vctr*-2)
 $\leftrightarrow$ (SDI { For 1,n }-*input_vctr*-1)
- 4: (SDS *input_vectors*- $\rightarrow$ *input_vctr*-2) $\leftrightarrow$ (SDI { For 1,n }-*input_vctr*-1)

Evidently, the output of the dependency detection algorithm is only the trace of the third pass. The file *infile* has to be accessed by the various I/O instructions in the order in which these instructions appear in the trace. Each I/O instruction receives one value from the I/O file unless it belongs to a loop or to a conditional block. If it belongs to a loop construct, then the range of the loop(s) determines the number of I/O elements accessed by it. If an I/O instruction belongs to a conditional loop, then it accesses the I/O file if the condition is satisfied. For example, the

first line means that the read on line 2 of the function *input_vectors* accesses the first value of the file, etc. The purpose of the information encrypted in the trace will become more apparent in the next two sections, when the construction of the Input and Output Access Graphs is presented.

The dependency detection and the compiler implicit ordering of I/O instructions presented in this section made the assumption that the whole program is presented during the analysis. This, however, is not a required condition when dealing with languages like SISAL (augmented with I/O extensions) that are strongly typed and explicitly define the type of the function arguments. In such languages, the techniques presented here can be used for incremental compilation as well. During incremental compilation, the compiler can easily decide if the function in question accesses an I/O file (or channel) and create its local IAG if needed. At link time, the local IAGs are linked together to form the global IAG in the same way the individual data-flow graphs of functions are linked together to form the "global" data-flow graph.

4 Input Processing

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 as much as absolutely necessary. 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. For each read instruction in the program, there is a corresponding stage in the IAG. This stage is composed of a read actor, a file-pointer increment module, and possibly some tag manipulation actors. When a stage has enough information to determine how many input values it consumes, it creates the effective file-pointer for the next stage and passes it to it. Thus, not one but a number of file pointers are distributed through the IAG. This notion of distributed file-pointers in the IAG departs from the sequential environment of *the file-pointer* that is kept with the file. Parallel execution of I/O operation is thus made possible with this distributed file-pointer scheme. We will return to elaborate on the parallel nature of this scheme at the end of this section after the various I/O constructs are individually presented. The computa-

Figure 2: The relation between computation graph and IAG for function *simple_io*

tion data-flow graph and the Input Access Graph for function *simple_io* are depicted in Figure 2. Each read instruction in the code corresponds to one stage in the IAG. Figure 2 demonstrates that the order in which these stages appear in the IAG is the same as the lexico-graphical order of the read instructions.

The Input and Output Access Graphs are constructed with actors from the union of the U-interpreter[7] actor set and the Token Relabeling [6] actor set with the addition of read and write actors. These new actors are also functional, thus preserving the data-flow principles of execution.

The read actor *R* obtains from the I/O file the value pointed at by the file-pointer, tags it, and sends it to its consumer in the dynamic data-flow graph. For example, in Figure 2, the output of the read (value *flag*) in the first stage goes to the compare actor (*=*) actor of the computation graph, etc. The read actor is the basic input actor and can be used for all input instructions. The construction of the IAG will be presented in five parts. The first four correspond to the four basic input constructs found in a data-flow graph: simple read, read inside a *forall* loop(s), conditional read, and read inside *repeat-until* loop(s). The fifth and last part is the synthesis of a number of the four basic constructs.

4.1 Simple Read

This is the simplest case of input instruction: a read instruction within the scope of an acyclic constructs. The stage of the IAG corresponding to this case is just a read (*R*) actor and increment (*inc*) actors. The increment actor (*inc*) is the simplest form of a file-pointer increment module; it increments the value of file pointer by one and passes it to the next stage of the IAG (if it exists). Simple reads correspond to the

first and third read actors of the *simple_io* function shown in Figure 2.

4.2 Read inside a Forall loop

A very common way to input arrays is by using nested loops. This is precisely the case in the function *input_vctr*. It is sufficient here to use the same configuration as in the simple read plus a mechanism for incrementing by 1 the value of the file pointer "*fp*" *n* times, where *n* is the range *n* of the loop (For *i* in 1,*n*). This also requires that the *n* Boolean values (used for the creation of the indices of loop) be received from the computational subgraph. Figure 3 depicts the data-flow graph of function *input_vctr* that uses a forall loop for inputting a vector (1-D array) of *n* elements. The mechanism used to create all *n* file pointers is identical to the index generation mechanism of the dynamic data-flow graph. The only difference is the absence of any decision ("compare" (>)) actor. The Boolean values are instead sent by the computation data-flow graph to the IAG as depicted in Figure 3. The stage of the IAG corresponding to the *read-inside-a-forall* in the general case needs a *file-pointer-increment-module* in order to create and pass the effective file pointer to the next stage. This is just a "plus" actor that adds the value of the file pointer received from the previous stage to the range of the forall *n* and passes it to the next stage. This is depicted in Figure 5.

Figure 3: IAG of function "*input_vctr*"

4.3 Conditional Read

The conditional read is encountered whenever an I/O instruction is part of a conditional block. Each conditional introduces an If-block actor in a stage of the

IAG. Basically, the If-block passes the file-pointer to the read actor of its own stage if it receives a Boolean "True" (port 1), or to the next stage if it receives a "False." In its simplest form, the IF-block is a "switch" actor as depicted in stage 2 of Figure 2.

4.4 Read inside a repeat-until loop(s)

A stage corresponding to a *repeat-until* loop uses a switch, an increment actor (*inc*) and a D actor in order to recirculate the file-pointer to the read actor of the same stage if the predicate is satisfied or to the next stage if not satisfied. In nested *repeat-until* constructs, each additional one introduces a switch actor that gates the file-pointer. An example of nested *repeat-until* loops is given in Figure 4. This example demonstrated how to implement the very popular approach in imperative languages of reading a whole line until the end-of-line (*eol*) mark is encountered and terminating the input process when the end-of-file (*eof*) mark detected. The module shown in Figure 4 employs some amount of look-ahead. The Boolean value of the predicate of the repeat-until is used to initiate the read of the next iteration and not the read of the current iteration. Therefore, the actual read operation can be performed before it is requested. This look-ahead improves the performance of the otherwise sequential *repeat-until* construct. Recursion has the same effect as *repeat-until* loops as far as I/O is concerned. Therefore our scheme treats recursion in the same way as a single *repeat-until* loop.

Figure 4: An example with nested *repeat-until*

4.5 Synthesis

It is possible to perform an input construct by using a combination of the four basic constructs. This

Figure 5: The Data-flow graph and the IAG for function *input_vectors*

has been demonstrated in function *input_vectors* 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 [8]. The mapping algorithm will not be presented here but the various examples presented here give the flavor of how the mapping is performed. Figure 5 depicts the data-flow graph, the IAG, and the communication between the two for the function *input_vectors*. The IAG has four stages (one for each read instruction in the program). The first and second stage correspond to the unconditional reads of the main function *input_vectors*. The third and fourth stages correspond to reading values by invoking the function *input_vctr*. As mentioned in Section 3, the IAG of individual functions can be constructed during incremental compilation. At link time these individual components can be linked together in a fashion similar to the linking of the computation graphs of the individual functions, as shown in Figure 5.

The only communication between the various stages of the IAG is done by the passing of *file-pointers*. Stages 3 and 4 have a *tag-relabeling* module. This is

necessary because these stages correspond to I/O operations by the function *input_vctr* and therefore belong to different contexts. The tag-relabeling block uses the token-relabeling techniques [6] to copy the tag of the token that belongs to the function argument to the incoming file pointer. This is equivalent to having a function call with multiple arguments.

As demonstrated in Figure 5, computing the file pointer of the next stage is a simple operation and involves only a few actors. For example, in stage 1 a single increment actor is adequate. Even in stages corresponding to nested Forall constructs, it only takes ($k - 1$ multiplications and 1 addition), where n is the degree of nesting. Therefore, since the file-pointers of each stage are computed quickly relative to the I/O operations, it is possible that many (or even all) of the stages of the IAG are active in parallel, thus yielding parallel I/O operations. The IAG construction follows the exact dynamic data-flow principles as the computation graph. This scheme provides a similar amount of parallelism as is achievable in a computation graph with similar constructs.

The treatment of output values in the data-flow environment poses a somewhat different situation than the treatment of input values. In the output case (write), a value has a tag associated with it. This tag will determine the position of the value in the output file. The techniques for creating the Output Access Graph (OAG) are similar to the ones for the IAG and are not discussed here. The interested reader can refer to [8] for a detailed description of the treatment of output in the dynamic data-flow environment.

5 Concluding Remarks

We have presented a compile time scheme that enables the detection and prevention of the various I/O hazards in a data-flow environment, thus making it possible to incorporate general purpose I/O functions in a data-flow environment. Furthermore, a scheme with distributed file-pointers has been adopted that enables the parallel execution of I/O operations. Maximum level of parallelism, as allowed by data-dependencies, is made possible by the proposed scheme. The two basic I/O instructions, *x := read(filename)* and *write(filename, value)*, have been introduced as primitives with side-effects. The concept of *sequential ordering* (logical ordering) of the I/O instructions has been adopted. I/O dependencies have been classified by the way an I/O object is shared. Thus, two kinds of I/O dependencies have been identified: (I) I/O object Shared by Different Program Segments and (II) I/O object Shared by Different Instantiations of the same

Instruction (SDI). The case of input (reading values) has been extensively analyzed.

The effect of I/O operations is isolated and localized by splitting the dynamic data-flow graph into two subgraphs: a) the *computation subgraph* and b) the *I/O subgraph*. We have developed a dependency detection algorithm that identifies all I/O dependencies in a program with or without conditional statements. This algorithm produces a Dependency Trace that indicates the *logical ordering* of the I/O instructions. The Dependency Trace is used to create the I/O Access Graphs that performs all I/O related transactions.

References

- [1] Arvind and R. Iannucci, "A critique of multiprocessing von Neuman style," in *Proceedings of the 10th Annual Symposium on Computer Architecture*, (Stockholm, Sweden), June 1983.
- [2] R. M. Soley, "Implicit serialization in Dataflow programs: Input/Output in id," Tech. Rep. Computations Structures Group Memo 277, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, Massachusetts, December 1987.
- [3] J. McGraw, "Data-Flow computing: the VAL language," *ACM Transactions on Programming Languages and Systems*, vol. 4, Jan. 1982.
- [4] D. Abramson and G. Egan, "The RMIT Data flow computer," *The Computer Journal (British computer Society)*, 1988.
- [5] J. McGraw, S. Skedzielewski, S. Allan, D. Grit, R. Oldehoeft, J. Glauert, I. Dobes, and P. Hohensee, "SISAL-Streams and Iterations in a Single Assignment Language, Language Reference Manual, Version 1.2," Tech. Rep. TR M-146, University of California - Lawrence Livermore Laboratory, March 1985.
- [6] J.-L. Gaudiot and Y. Wei, "Token relabeling in a tagged token Data-Flow architecture," *IEEE Transactions on Computers*, vol. 38, Sept. 1989.
- [7] Arvind and K. Gostelow, "The U-Interpreter," *IEEE Computer*, pp. 42-49, February 1982.
- [8] P. Evripidou and J.-L. Gaudiot, "Input/output operations in a data-driven environment," Tech. Rep. CRI 88-45, University of Southern California, Dept. Electrical Engineering-Systems, Computer Engineering, July 1988.

**DATE
FILMED**

12 / 7 / 93

END