Implementation Schemes for Data Flow Procedures

Computation Structures Group Memo 138-1
May 1976
(revised February 1977)

Glen Seth Miranker

This research was supported in part by the National Science Foundation under grant DCR75-04060.

Contents

1. Introduction

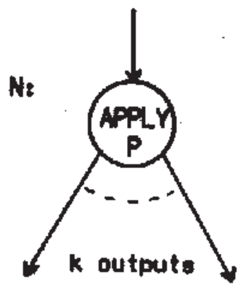
Data flow schemas have proved to be an effective model of parallel computation [10]. Important questions about parallel computations such as determinacy and proper termination have been shown to be answerable for data flow schemas by straightforward syntactic analysis on the schemata themselves [17], [25], [12]. Furthermore, the translatability of "high level" block structured language into data flow language is feasible [1]. Thus an encoding of data flow schemas would appear to be a suitable base language for a parallel processor, since such a processor would then be able to execute parallel programs that have all the desirable characteristics of this class of program schemas:

1. High degree of parallelism
2. Syntactic tests for determinacy
3. Generality - great expressive power
4. Translatability - Though no practical translator from a high level language to data flow has been built, certainly the feasibility of such translation procedures has been demonstrated. [25]
5. Modularity

Some encouraging preliminary work has been done toward the design of a data flow processor [4] [6]. However, implementation of certain features of the base language have proven difficult. Procedure application has been a particularly thorny problem. The basic difficulty with procedure invocation on current data flow machines is that the encoding of the schemas executed by the machines are inherently *impure*. That is for each actor in the schema the data values on their input arcs are stored with the actor itself. Consequently, establishment of a procedure's activation requires not simply creating a unique data area for the activation, but actually setting up a new copy of the code. The problem is further complicated by the fact that computations *including* distinct activations (such as those arising from recusive calls) of a given procedure proceed in parallel. Several implementation schemes for procedure are presented below. The discussion to follow assumes that the reader is familiar with the data flow language, [7] as well as the basic data flow processor [5]. This processor will be modified slightly to accomodate procedure invocation. This particular machine was chosen because it is representative of the class of data flow processors currently being studied in the Computation Structures Group at M I T. Furthermore, of the machines in this class it is the simplest processor with sufficient power to support the implementation scheme for procedure activation to be described. Also the ideas presented below are directly applicable to the most advanced data flow processor currently proposed [20]. In order ro handle procedures we must add one more part to the virtual memory system of the machine - a *relocation box*.

2. Procedure Invocation and Activation Names

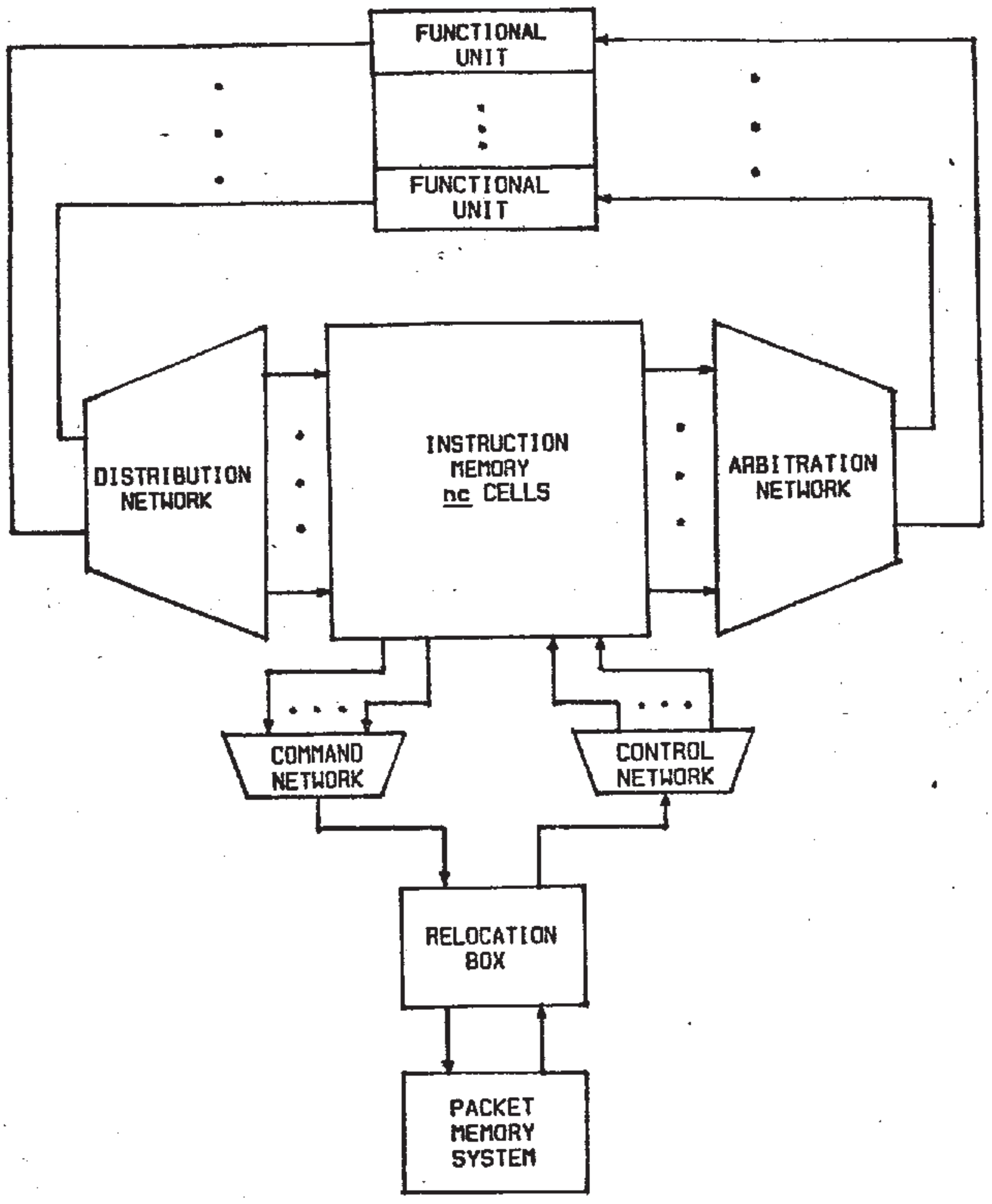
There are several ways of invoking a procedure in data flow language that are consistent with the data flow model. The simplest method is a single argument APPLY actor:



APPLY Actor
Figure 1

The effect of APPLY P is simply described. When a data token α arrives on the input arc a copy of the data flow graph for P is made and α is absorbed from the the input arc of N and a copy is placed on the output arc of the input link of procedure P. As each of the k outputs for this activation of P is produced, it is passed from its output link to the corresponding output link of the APPLY and hence to a successor cell of N. (For convenience in discussion, each cell in a data flow program is assumed to have a name, indicated next to the cell in figures as <name>:. The name of an actor is actually used in the data flow processor to identify it. For example, to route result packets to it or to retrieve it from memory.) To be syntactically correct, P must have one input link, and k output links.

The structure of the machine to be used for the following implementation of single argument/single output APPLY is similar to that desrcibed in [5]. In particular, the machine which is shown in figure 2 has a two level memory structure. The IM functioning as a cache for the cells of the program stored in the PMS.



Machine With Procedure Capability
Figure 2

Anticipating mechanisms to be proposed later, some of the functions of the *relocation box* will be described. It is assumed that every actor in a data flow program as represented in the processor has a unique cell name except for APPLY actors. Further, during the course of execution (where and how will be described later) a cell name may have a suffix appended to it - these suffixes will serve to distinguish procedure activations. At any instant during the execution of a program there will be a one to one correspondence between suffixes and procedure activation. In the following discussion we will seperate a cell name from a suffix by a ".". The relocation box's operation is quite simple. Upon receipt of a *fetch packet* from the memory control network:

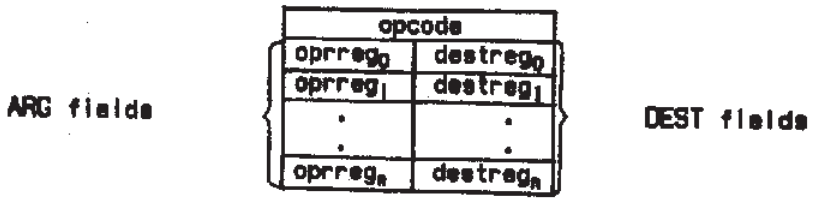
{<u>fetch</u>, $\alpha.\sigma$}

It passes the packet:

{<u>fetch</u>, α}

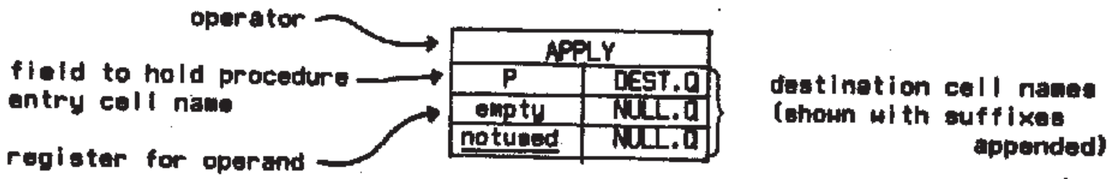
to the packet memory system. When cell α is returned by the packet memory system to the relocation box, all the names in its destination fields are changed to have suffix σ. (Null suffixes are allowable.) The relocation box then passes the cell back through the memory control network to the instruction memory. Finally, it is assumed that with the sole exception of the relocation box and one special functional unit, no other component of the data flow processor of figure 2 can distinguish if a cell name has a suffix appended or not. That is, if the distribution network for example, receives a packet with a destination cell name $\alpha.\sigma$ it sends a result packet to cell $\alpha\sigma$ (the dot seperating the name from the suffix is included merely as an aid to the reader's eye). The essential idea is that a complete cell name (i.e. a cell name plus an appended suffix, which we refer to as an *execution name*) is treated everywhere but the relocation box and the distinguished functional unit as a single entity - a cell designation.

To simplify the discussion, a cell image will be represented as:



Cell Representation
Figure 3

Thus we temporarily ignore the fixed number of ARG and DEST fields of a cell. The APPLY cell in particular has the format shown in figure 4.


operator
field to hold procedure entry cell name
register for operand
APPLY
P
DEST.Q
empty
NULL.Q
notused
NULL.Q
destination cell names (shown with suffixes appended)

An Apply Cell
Figure 4

The implementation of a single input-single output APPLY actor is straight forward. When the operand a arrives, the APPLY becomes enabled, and transmits the following packet:

(APPLY, P, a, DEST.Q, NULL.Q, NULL.Q)

which the arbitration network routes to the special functional unit that processes applies and returns. The functional unit on receipt of an *apply operation packet* creates a unique suffix e and then outputs two packets:

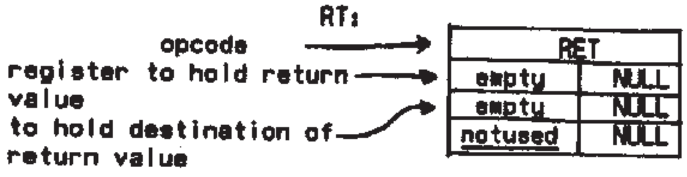
(a, P.e) and (DEST.Q, RT.e)

Packets Creating a New Activation
Figure 5

(Every packet sent to a cell must also contain *field addresses*, that is specifications of which register in a cell is to receive the value(s) conveyed by the packet. These will not be explicitly represented in the diagrams since it should be clear from the context which register of a cell is supposed to receive which value. Leaving out the field address will make the diagrams a bit less detailed and hence less confusing.)

The first packet arriving at the instruction memory causes the cache mechanism of the instruction memory to retrieve from the packet memory system a cell with name P.σ. (Since σ is unique suffix, cell P.σ can't possibly have been in the instruction memory.) Due to the action of the relocation box, a copy of cell P will be retrieved and all its destination fields given the suffix σ. Once the cell P.σ is successfully installed in the instruction memory, actor P.σ will then receive its operand σ, and become enabled. Whatever computation is specified by P (the first actor of the invoked procedure) will be carried out and the resulting values will be in packets destined for cells $D_0.\sigma$, $D_1.\sigma$ . . . $D_n.\sigma$ where D_1, . . . D_n are the contents of the destination fields of P. Clearly $D_0.\sigma$ . . . $D_n.\sigma$ will not be found in instruction memory and will be fetched from the packet memory system in the manner described above for P.σ. And so execution of the σ^{th} activation of P proceeds.

To return the value computed by P we assume that all procedures are compiled so that their output value is to be sent to a cell named RT, which is a *reserved* cell name. That is, rather that having an output link, programs are compiled to have a (uniformly named) output cell. This convention coupled with the way returns (as outlined below) are handled, allow data flow programs to be compiled independently of each other, and loaded into the packet memory system without altering any of the return destination fields. Further, it is assumed that resident in the packet memory system is a cell of the form:


RT:
opcode
register to hold return value
to hold destination of return value
RET
empty
NULL
empty
NULL
notused
NULL

Return Cell
Figure 6

that belongs to no procedure (i.e. it is a runtime support cell!). It will be retrieved by the second packet generated by the functional unit as a result of the APPLY cell firing (see figure 5) and be resident in the instruction memory in the form:

RT.φ:

RET	
DEST.Ω	NULL.φ
empty	NULL.φ
notused	NULL.φ

Return Cell in IM
Figure 7

With the above-mentioned compiler convention, when execution of the procedure is complete the following packet will be sent by the functional unit:

(Ω, RT.φ)

(where Ω is the output value of the φth activation of P).

Thus RT.φ will be enabled and create the packet:

(RET, DEST.Ω, Ω, NULL, NULL.φ, NULL.φ, NULL.φ)

which is sent to the appropriate function unit. This FU will then output the packet:

(Ω, DEST.Ω)

thus sending the result of the σth activation of P in the correct destination cell. It also returns σ to the list of free unique suffixes, and outputs the packet:

{σ, IM}

which causes

1. Every cell en-cached in the IM with a name having a suffix of σ is purged.

2. All <u>store</u>, and <u>fetch</u> packets with a <u>name</u> with a σ suffix are destroyed.

These last two "cleanup" actions are necessary to prevent unecessarily tying up of processor resources. To see why this is the case, let us consider a specific example. Suppose a procedure terminates and the suffix σ is returned to the free pool, but the IM is not cleared. Then in general there will still be cells left in the memory with suffix σ. If σ is reused for another procedure activation, then

1. If the new activation is another instance of the same procedure as the old activation, then a cell left in the IM may be referenced again. This is alright provided the DFL procedure of which the cell was part of is well behaved. For if this is the case, the cell will be in its original state and hence reusible.

2. If the new activation is of a different procedure than the previous use of σ, the cell must not be used, since in general, it will not be an appropriate encoding. One can ensure that the cell is not used by requiring the sets of names associated with procedure encodings to be disjoint. However, if the (old) cell is displaced from the IM to the PMS to make room for another cell, we have just wasted a PMS memory location.

This procedure application scheme has several attractions. First it is simple. Overhead in terms of storage, or extra packets in the system, is almost zero. Few changes need to be made to the basic data flow processor, and those that are necessary are incorporated in a smooth and natural way.

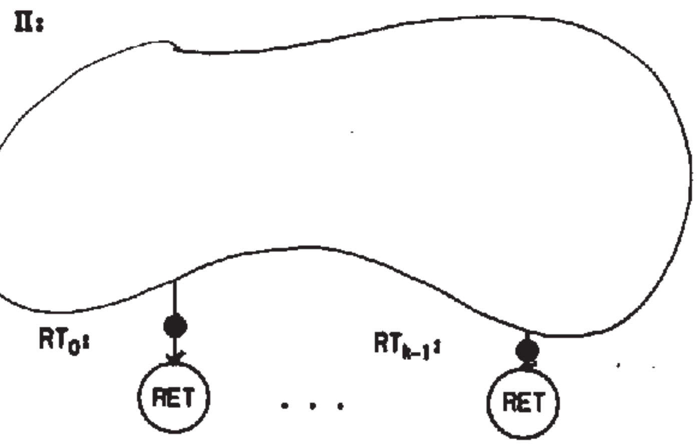
Also, notice that in this scheme the entire procedure is not copied, just the pieces of it as they become active. This is an important characteristic especially for programs with conditional constructs. For these programs, the amount of processing activity is not uniform over all program actors. In particular the predicate of a conditional schema Π, will select either the "true" or "false" subgraph of Π. It will never be the case that both subgraphs (of a given activation) execute. Thus to load both of them into the instruction memory is wasteful of instruction memory space and memory control network bandwidth. Finally, procedures are compiled no differently than programs, thus allowing (without recompilation) the use of any data flow program as a data flow procedure. This will be discussed at greater length in section 7.

3. More Elaborate Schemes.

The primary deficiency of this scheme is that it implements a rather primitive form fo the APPLY actor - only one input and one output. If this implementation of procedure call was incorporated in the data flow processor with structures outlined in D. Misunas' masters thesis then this deficiency would not be so bad since then multiple input values and multiple output values could then be encoded as structures [21]. However, this form of procedure invocation is undesirable because it limits the degree of parallelism achievable. After all, there is no inherent reason to return all the outputs of a procedure simultaneously. If a procedure produces k outputs, to wait for all of them to be computed, assemble them into a structure, return the structure to the calling routine, disassemble the structure, and then use the resulting k components, restricts the amount of concurrency achievable in a program. It also incurs the overhead of assembly and disassembly of structures. One would (ideally) like to pass each of the k output values to its destination in the calling procedure as it is generated.

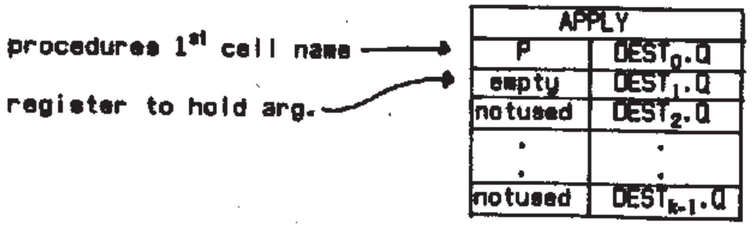
Similarly, one would like to have multi-argument functions. Passing an ℓ component structure with the ℓ argument values to a procedure as its components is undesirable. Again there is a loss of parallelism. A particular subgraph of the data flow procedure may require only a subset of the ℓ input values to start execution. Thus to inhibit passing of any argument values to the procedure until all of them are available, seriously limits the achievable level of concurrency. This view of the desired operation of the APPLY actor (i.e. as soon as an input value is available copy the procedure and begin execution, and pass each of the outputs to the calling procedure as soon as they are available) is sometimes referred to a procedure application (which we called the immediate copy rule). Some workers [25] regard this kind of behaviour for the APPLY actor as undesirable. They claim that even for multi-input/multi-output actors, one should wait for all inputs, then start the computation, wait until all the outputs are produced, and then pass them all simultaneously to the successor cells of the APPLY. This operation of the apply actor (which we called the deferred copy rule) is known as procedure call. The "semantic" distinction between these two forms of procedure invocation as discussed in chapter two is rather striking.

First we consider single input/multiple output procedures with immediate copy. Instead of having a single "floating" RETurn cell in the packet memory system, there are j return actors with names RT_0 through RT_{j-1} in the packet memory system again belonging to no routine. (j is assumed to be the maximum number of return values supported by the compiler. Note that the limit on $\int$ is not due to the processor, but a compiler limitation.] Again it is assumed that every k output procedure Π is compiled to send its output values to cells named RT_0 . . . RT_{k-1} which are not included as part of the procedure as it is stored in the packet memory system:



Procedure with Implicit Return Structure
Figure 8

The format of each return actor is precisely as in figure 6. The APPLY actor is a bit different, however:


procedures 1st cell name
register to hold arg.

APPLY	
P	$DEST_0.Q$
empty	$DEST_1.Q$
notused	$DEST_2.Q$
. .	. .
notused	$DEST_{k-1}.Q$

Multiple Output Apply Cell
Figure 9

(Extending the scheme for APPLY in immediate copy procedure invocation where the number of destinations k is too large to fit in a cell, is straightforward and its details obscure the essential features of the method. Consequently it is omitted. The curious reader who wishes to develop this extension is asked to consider using an APPLYContinued cell which the apply functional unit fetches after executing an "obese" APPLY to complete setup of the return mechanism. In the following discussion we will ignore the limitations on the number of inputs and outputs of APPLY actors imposed by the fixed number of ARG and DEST registers. The APPLY, upon receipt of α (the input value), becomes enabled and outputs the packet:

{APPLY, P, α, <u>NULL</u>, $DEST_0.Q$, $DEST_1.Q$, $DEST_2.Q$, ..., $DEST_{k-1}.Q$}

analogous to the previously described scheme. The apply funtional unit creates (k+1) packets now:
{α, P.σ} as before and {$DEST_0.Q$, $RT_0.\sigma$}, {$DEST_1.Q$, $RT_1.\sigma$} ... {$DEST_k.Q$, $RT_{k-1}.\sigma$}

Figure 10

The first packet above initiates procedure execution as described in the single input/single output apply. Each of the other packets (in a manner entirely analagous to the single output scheme) causes the cell

$RT_j.\sigma$:

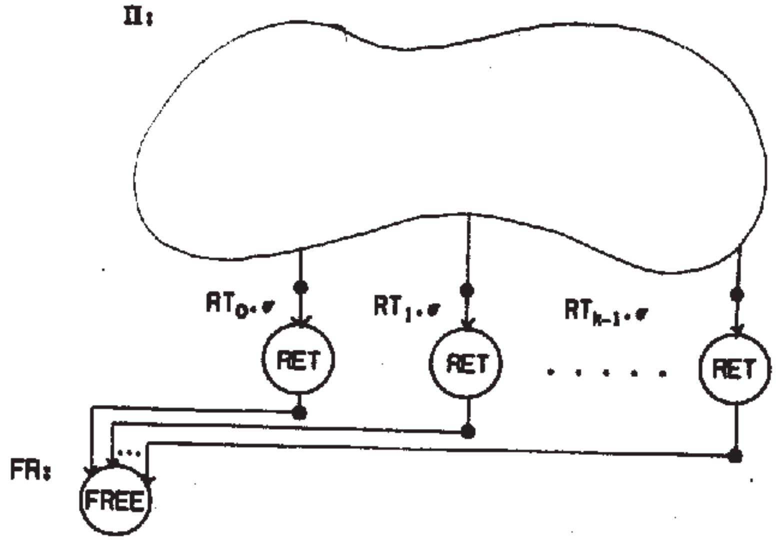
RET	
$DEST_j.Q$	NULL.σ
empty	NULL.σ
<u>notused</u>	NULL.σ

Figure 11

(for the (j+1)st packet sent by the apply functional unit) to be loaded into the instruction memory.

When the jth output value Ω_j is computed it will (by compiler convention) be sent to cell RT_j. In precisely the the same way as in the single output scheme, then the Qth activation of $DEST_j$ will receive Ω_j as required. However, the apply functional unit may <u>not</u> free the activation name σ since in general activation σ is not finished.

To allow freeing of activation names we introduce a new cell type FREE. Consider the example below:



FREE	
empty	NULL.σ
empty	NULL.σ
empty	NULL.σ

Multiple Output Return Mechinism
Figure 12

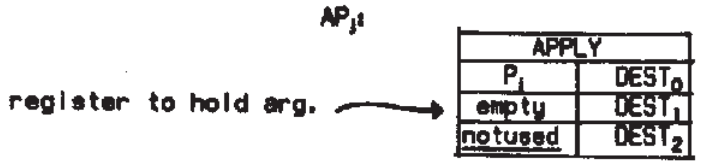
FREE should not be available to the programmer. It is a "runtime" support actor which allows proper activation name maintenence. This new actor has been introduced (just like RETurn actors) as a convenient way of showing how a procedure call and return is implemented. It is not an addition to BL, but merely a way of presenting the details of the call-return mechanism that preserves a one to one correspondence between cells and actors.

The FREE cell is enabled when all the output values of the σ^{th} activation of Π is finished since these are the values that comprise its arguments. It outputs a packet:

$$\{FREE,\ \Omega_0,\ \Omega_1,\ \Omega_2.\ .\ .,\ NULL.\sigma,\ NULL.\sigma,.\ .\ .\ NULL.\sigma\}$$

which is routed to the functional unit that handles APPLY-RETurn. Upon receipt of this packet the functional unit frees the activation name σ and outputs the $\{\sigma, IM\}$ packet as before. The reader should notice that unlike the RETurn cells, the FREE cell is explicitly included as part of the procedure application mechinism. This save us the trouble of determining at runtime the number of values the FREE cell must receive before becoming enabled. It may be objected that since the RETurn cells are not part of the procedure that the RETurn actors have no way to "know" what the name of the FREE cell is. Consequently, they cannot send result values to it. This is indeed a bug in the above scheme. The fix is rather simple however. Briefly, we simply send *two* destination names to the RETurn actors at runtime - the name of the cell that is to receive a result of the procedure call, and the name of the FREE cell. Rather than describe this alteration in greater detail we will present a more elaborate procedure implementation scheme which explicitly addresses this problem.

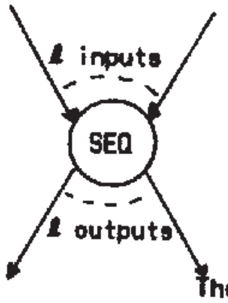
We are finally ready to attack implementation of the immediate copy rule for multiple input-multiple output APPLY. The naive approach (in the context of the previous discussion) is simply to have j apply cells for a j input APPLY actor each receiving one argument value:



APPLY Cell for Multiple Input APPLY
Figure 13

All fields are as in figure 9 except P_j is the j^th input link of procedure P. The compiler is assumed to write "code" to send the j^th argument value to cell AP_j. This method has several liabilities. One is aesthetic - a single data flow program actor maps into several cells. (I believe this to be unavoidable with the current view of cell operation. Presently, in order to keep the amount of state information required for correct cell operation, a cell is only enabled when all of its input values are present - not just some subset of them.) The other is that it does not work. For each cell AP_i the functional unit will assign a new activation name. This condition will guarantee incorrect operation for all but the most contrived programs. Even if this could somehow be patched, there is also no mechanism for correctly returning output values since each cell AP_i would cause a new set of RETurn cells to be fetched.

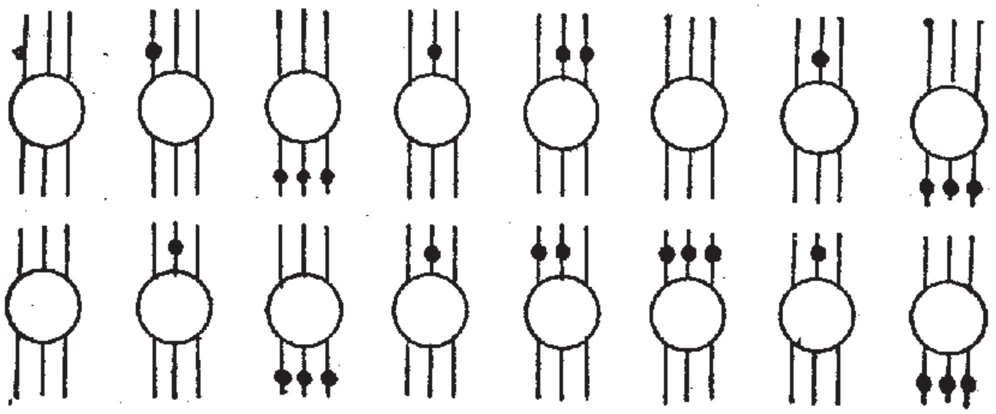
To circumvent these difficulties, we need to guarantee that for each set of input values to a particular APPLY only the first argument value causes generation of a packet which causes the functional unit to create a new activation name. Furthermore, we must make certain that the other argument values that are sent to the APPLY actor are all passed to the same procedure activation as the first. Finally, we must be sure only one return mechanism is set up. To do this we introduce a new actor which we call SEQ (for sequence) it has ℓ inputs and ℓ outputs:



SEQ	
empty	$DEST_0$
·	·
·	·
empty	$DEST_{\ell-1}$

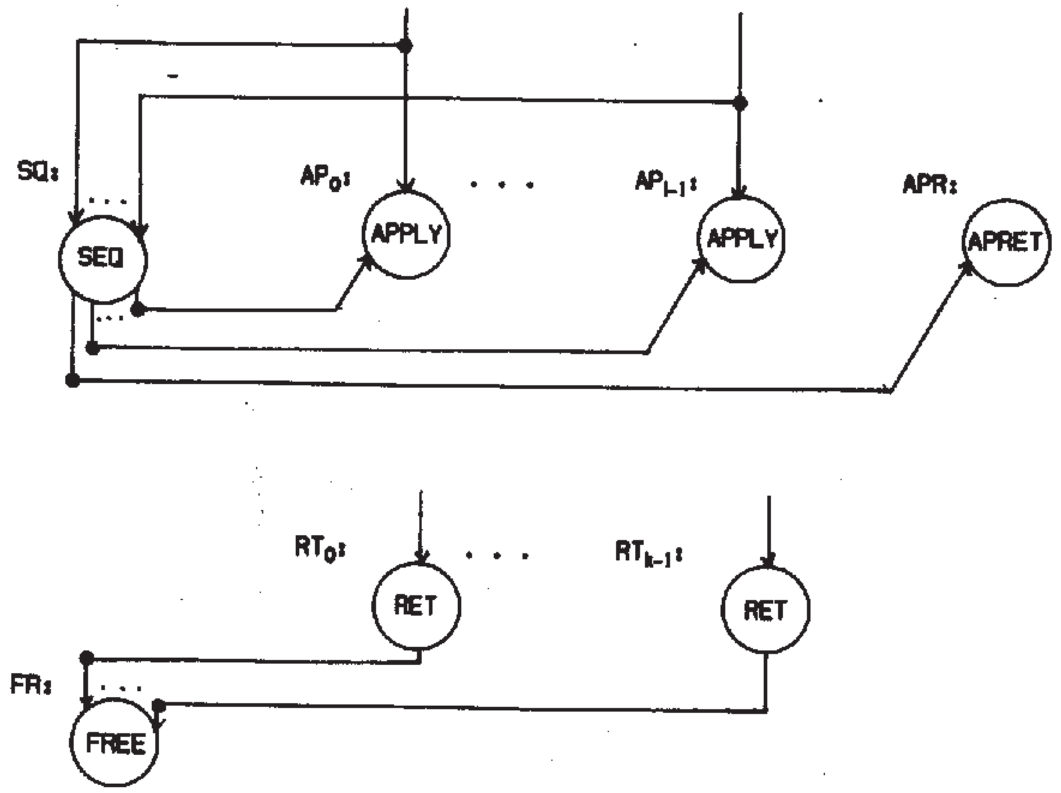
The SEQ Cell
Figure 14

Its operation is simple. Upon receipt of any one of its inputs, say on input arc j, it produces an output value which is a unique suffix name. This value is then passed out on each of SEQ's output arcs. No further action is taken until tokens arrive on inputs 0,2, . . . j-1,j+1, . . . ℓ-1. When this state occurs, one token on each of the input arcs i (i ≠ j) is absorbed and no output token is produced. The actor (and cell) then return to the initial state and the above action is repeated. For example:



SEQ Actor Firing Rule
Figure 15

Notice that while the times at which outputs are produced are quite unusual compared to the other data flow actors, only one output is produced for each set of ℓ input values. Again as with the FREE actor, SEQ is introduced as a notational convenience to help explain (in terms of data flow actors) how the activation name scheme is maintained and to provide a more direct correspondence between data flow program actors and cells. It should not be regarded as part of the data flow language. Hence it is not available to the programmer and its effect is completely transparent. The compiler which must necessarily be aware of the details of implementation makes use of them however. We implement an ℓ input k output immediate copy APPLY as:



Procedure Call Mechanism
Figure 16

The ℓ input APPLY actor now maps into ℓ APPLY cells of the form in figure 13 except that they have NULL destination field, and the third argument field is used to hold a suffix name. SEQ and FREE actors are added to maintain activation names, and RET actors to handle return values. However, at the "source" level all the user is aware of is that he is using a single actor, an ℓ input/k output immediate copy APPLY. To see how this all works, let us suppose that the 0th activation of some procedure produces the ith argument α_i for an application of Π, where Π has ℓ inputs and k outputs. For concreteness, we depict the case of $\ell = 2$, $k = 3$ in the figures.

The compiler will have generated code so that the actor that produces the value α_i has as its destinations the cells AP$_i$ and SQ. Thus the packets:

$\{\alpha_i,\ AP_i.Q\}$ and $\{\alpha_i,\ SQ.Q\}$

Figure 17

will be produced. These will cause the following two cells to be fetched into the instruction memory:

$AP_i.Q$:

APPLY	
P_i	NULL.Q
empty	NULL.Q
empty	NULL.Q

SQ.Q:

SEQ	
empty	$AP_0.Q$
empty	$AP_1.Q$
empty	APR.Q

Figure 18

where we have depicted the cells corresponding to cells AP_i and SQ just before the argument packets of figure 17 have been delivered. SQ is now enabled and will generate the packet:

$\{SEQ, . . . \alpha . . . ., AP_0.Q, AP_1.Q, APR.Q\}$

which is routed to the apply functional unit. This functional unit outputs the packets:

$\{\sigma,\ AP_0.Q\}$, $\{\sigma,\ AP_1.Q\}$, and $\{\sigma,\ APR.Q\}$,

Figure 19

The packet:

$\{\sigma,\ AP_i.Q\}$

will enable the cell $AP_i.Q$, and as a result it will output the packet:

$\{APPLY,\ P_i,\ \alpha,\ \sigma,\ NULL.Q,\ NULL.Q,\ NULL.Q\}$

which is then routed to the apply functional unit. It then outputs the packet:

$\{\alpha, P_i.\sigma\}$

and execution from the i^{th} entry cell (σ^{th} activation) procedes as described in the single input case. The other packets of figure 19 i.e. :

$\{\sigma, AP_j.Q\}$ $\qquad j \neq i \ , \ j \leq \ell$

each cause cache faults, thus bringing in cells:

$AP_j.Q$:

APPLY	
P_j	NULL.Q
empty	NULL.Q
empty	NULL.Q

$j \neq i \ , \ j \leq \ell$

Figure 20

When their arguments arrive, execution will proceed in the manner described for AP_i.

The returning of values from the σ activation of Π is handled in a similar fashion as that described in the previous section on single input/multiple output (immediate copy) APPLY. In this case it is the APRET cell that causes the RET cell to be fetched into the instruction memory. The $(\ell + 1)^{st}$ packet output (see figure 19) retrieves the cell:

APR.Q:

to hold activation name

APRET	
empty	$DEST_0.Q$
FR	$DEST_1.Q$
$\underline{\underline{\text{notused}}}$	$DEST_2.Q$

Figure 21

Notice that the APRET cell has as a (constant) operand, the name of the FREE cell. When it is enabled it produces the output packet

$\{APRET, \sigma, FR, NULL, DEST_0.Q, DEST_1.Q, DEST_2.Q\}$

This packet in turn causes the FU to output packets:

$\{DEST_0.Q,\ RT_0.\sigma\}$, $\{FR.Q,\ RT_0.\sigma\}$, $\{DEST_1.Q,\ RT_1.\sigma\}$, $\{FR.Q,\ RT_1.\sigma\}$,

etc.

Thus the RETurn cells receive as parameters the names of *both* of their destination. Otherwise, returning of values, and "destroying" an activation procedes precisely as in the previous scheme.

4. Relocation Box Revisited

To simplify the previous discussion we have purposely oversimplified part of the operation of the machine. We had said that the relocation box upon receipt of a fetch packet would always pass the packet to the PMS with the suffix stripped off the cell name. This is incorrect. If the machine really operated in this fashion, then the it could never retrieve cells that had been displaced from the IM into the PMS. There are a number of solutions to this problem. However selection among them is impossible without a more detailed model of the implementation of the packet memory system and the cache mechanism. Thus a full discussion is beyond the scope of this note. The particular solution described here was chosen for its simplicity, and should not be thought of as an "optimal" solution.

Upon receipt of a packet

{<u>fetch</u>, $\alpha.\sigma$, j}

the relocation box first checks if σ is a valid activation name. If it is not, the RB signals a runtime error, otherwise it issues *two* fetch requests to the PMS. One is for a cell with name α, and the other is for a cell with name $\alpha.\sigma$. The PMS will respond in one of several ways.

1. If the PMS returns nothing for either request, then the RB signals an error.

2. If the PMS returns something for $\alpha.r$, and nothing for α, then the RB signals an error

3. If the PMS returns something for both, then the cell image returned in response to the request $\alpha.r$, is passed back to the IM unalterred.

4. If the PMS returns something for the request α, and nothing for the request $\alpha.r$, then the cell image returned in response to the request α is passed back to the IM with its destination fields alterred as previously discussed.

The solution though simple, halves the effective memory bandwidth.

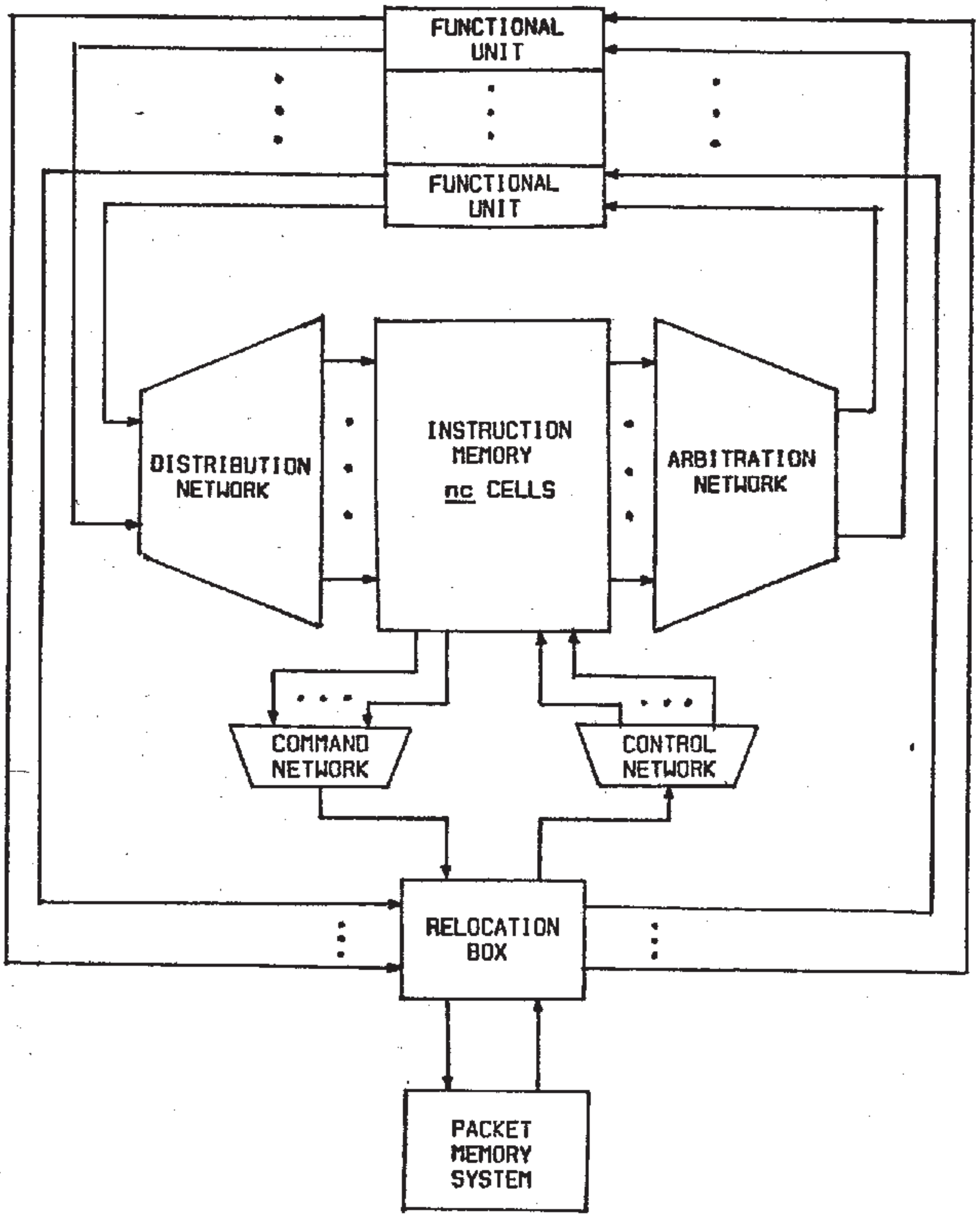
5. Names and Loading

One of the attractions of this implementation scheme for APPLY is that it does away with the need for an elaborate linking loader for data flow programs. Consider for example, a data flow program consisting of several procedures each of which has been compiled by itself. One may assume for the purposes of discussion that the cell names (and hence the contents of the destination fields of a cell) correspond in some direct way to the memory locations of the packet memory system (PMS). Thus when loading the component parts of the program (i.e. the procedures) into the PMS two things must be done. Assume for concreteness that a procedure is compiled into a linear block of cells numbered from 0 and that cell numbers are cell names. When loading a procedure, a number equal to the cell number into which the first cell of the procedure was loaded must be added to all destination fields of all cells that refer to cells that are part of the procedure. Then all external references in a procedure - that is entrance cell names in APPLY cells - must be set to the correct value. This value depends of course on the location into which the referenced procedure is loaded. Notice however, that return names need not be relocated since they are "constructed" at runtime. Thus no correcting of the destination addresses of an APRET cell need be done. Indeed, the RETurn cells which actually transfer return values to the invoking procedure are *not even part* of the invoked procedure. Thus the two tasks of loading - fixing (by adding an offset) of internal references is easy, and fixing of external references is greatly simplified. Only half of the job must be done before execution, since the entry but not the return points must be relocated.

The naming scheme provides a solution to establishing the "identity" of distinct activations amenable to efficient hardware implementation. Since a cell of an activation is uniquely identified by its name (assigned in the original compilation) and a single suffix, names are bounded in size (we assume of course that suffixes and simple names are of fixed maximum length). This is quite different from the naming scheme in the data flow interpreter presented by Arvind and Gostelow [2]. In their scheme the full name of an instance of an actor Π (during execution) consists of the original name plus k appended suffixes, where k is the activation level of Π. Thus names while always finite may be arbitrarily long. This poses some serious (hardware) implementation issues that are absent in the present model. All that is required is an enlarging of the data paths of the processor that carry names. (To handle 1000 simultaneous activations requires only 10 additional bits.) Of course one must also construct a relocation box.

6. <u>One Last Change</u>

Creation and returning to the "free pool" of activation names (i.e. suffixes) is probably not an appropriate activity for a FU. The primary reason is that one would like to have multiple FU's for processing of APPLY, RET, FREE, and SEQ, packets. Coordinating the creation and returning of suffixes to the free pool among several autonomous, asynchronously operating modules (FU's) is a messy task. It also introduces an overhead since the FU's coordination must take place through exchange of messages (packets) if we are to keep the overall machine structure consistent. Consequently, we propose the following modification to the scheme described above. We introduce an additional data path from each FU that processes applies etc. to the relocation box and from the relocation box to the those FU's. Thus the machine structure is:



Final Machine Structure Supporting Procedures
Figure 22

When a functional unit needs a new suffix rather than generating it internally, it now send the packet:

{NEWSUFFIX, j} where j is the name of the FU originating the request

to the relocation box. The RB responds by generating a new suffix name and sending the new name (as a packet over the new data path) to the requesting FU. When the FU wishes to free a suffix σ it sends the packet:

{FREESUFFIX, σ}

to the RB which then returns σ to the pool of free suffix names. Thus in the new machine, assignment and freeing of activation names takes place at a central location, hence avoiding the problems of maintaining a distributed list of free suffix names. The choice of using the relocation box to perform these functions is somewhat arbitrary, the main idea being to centralize activation name management, where it is done is not critical. Note that simply letting each apply FU manage its own activation names cannot work even if a FU's pool of names contains no elements in common with any of the others. The problem is that an additional mechanism must be provided to guarantee that a packet:

{FREE, σ}

is routed to the functional unit that created σ. Because of this complication the above scheme is inferior to the central name allocator.

7. Procedure Variables and Acknowledge Signals

It should be observed that the naming scheme for establishing unique activations in no way is dependent on the fact that the name of procedure in an apply cell that is to be invoked is constant. (Its placement in an operand field of the APPLY cell was intentional.) Consequently, without adding any additional mechanisms to the schemes proposed, procedure variables can be handled. One merely has to compile the APPLY cells with empty entries where the entrance cell names for the invoked procedure was. Of course now some other actor of the program must send a cell name corresponding to an entrance point of a procedure to the appropriate APPLY cell in order for it to become enabled.

One simple way to do this is to have each APPLY cell of the call mechanism receive the name of the procedure it is to invoke, rather than a node name of a particular entry point. When the ith one is enabled it passes to the FU a packet of the form

$$(APPLY_i, P, e, a, NULL.Q, NULL.Q, NULL.Q)$$

The FU "knows" from the opcode and the first two arguments, that processing of this packet is supposed to send the ith argument to the eth activation of procedure P. The FU can either

1. Look up in a table set up at loading time the name of the cell that is the ith entry point of procedure P.

or

2. With a suitable compiler convention, *compute* the name of the cell given i and P.

Though either approach works, the former has two advantages. First, since the call structure is set up at compile time, the number of inputs and outputs to the apply mechanism is known. This information could be incorporated in the encoding of the APPLY cells and also in the table. This would allow the FU to do a runtime check to see if the named procedure has the appropriate number of input and outputs. Second, using a table allows the FU to check if P is a defined procedure.

Finally, to be fully general, we must extend the procedure invocation scheme so that the call mechanism is incorporable in data flow (object) programs which explicitly acknowledge "absorption" of tokens [6]. For any data flow procedure, the RETurn actors are always enabled. If a successor of a (source) APPLY actor sends acknowledge signals to the APPLY, the destination of an acknowledge signal is the SEQ cell of the call mechanism. This guarantees that the SEQ cell can only initiate a new procedure instance if all the acknowledged outputs from the old procedure instance have been acknowledged. Of course the SEQ cell must be enabled in the initial configuration of the program. Similarly, if the (source) APPLY actor must acknowledge an input token, it is the FREE cell of the call mechanism that sends the acknowledge signals. This guarantees that the predecessors of the call mechanism will be enabled to send new argument values only after the previous procedure instance has been terminated. Otherwise the call mechanism is unalterred.

8. Correctness

There are a number of important things going on under the surface of this brief discusion. First, there is the question of race conditions. That is, is there some timing of argument arrivals (for a given procedure invocation) that cause more than one activation to be created. The answer of course is no, provided that SEQ is implemented correctly and that the schema containing the APPLY is safe. (That is no input to the APPLY receives its second token before all the inputs receive their first.) The second question is a bit deeper - does the implementation work correctly if the APPLY is part of a subschema which is the body of a loop? We informally argue that this is the case. We assume that the reader is convinced that the implementation correctly achieves the effect of replacing the apply node which invokes P with a copy of the graph for P provided that an activation σ initiated by an APPLY terminates before the values for the $(\sigma + 1)^{st}$ arrive.

We will make no claims for cyclic schemata in general. For although in the single input-single output case, succesive arrivals of argument values will each create a new procedure activation, each activation will attempt to return its output value(s) to the same destination node. Thus in general we have an unsafe condition the point of conflict being the destination node. However, for the special case of loops of the syntactic type allowed in BL - a subset of DFL defined in [19] - we can garantee correct behaviour. To see this, recall that such loops are safe. That is, in the petri net model of a schema of this class, regardless of the sequence of transition firings no place will carry a token load of greater than one. In the data flow schema, this means that even if an actor were to fire without first checking if its output link were empty no link would carry more than one data token. This property of safety holds regardless of the execution times of the actors.

Let $\mathfrak{a}$ be a single input APPLY actor in the body of some loop of a BL program. Suppose $\mathfrak{a}$'s input had no data dependence on any of its output value(s). Then data values would arrive at $\mathfrak{a}$ in some finite time independent of τ - the execution time of $\mathfrak{a}$. Then by letting $\tau \rightarrow \infty$ we can queue up any number of tokens on $\mathfrak{a}$'s input arc as we like - clearly an unsafe condition. Thus $\mathfrak{a}$'s input must be data dependent on its output(s) since we know that BL programs are safe. Consequently, each activation σ initiated by $\mathfrak{a}$ will finish before the $(\mathfrak{a} + 1)^{st}$ starts. Re-examining the above implementation for single input apply we see that this condition is sufficient to guarantee correct operation.

For multiple input -multiple output immediate copy (MIMIC), the above argument fails. This is due to the fact that an input to an APPLY may be dependent on only a subset of the outputs of the APPLY. To ensure the correct behaviour of the implementation of this form of the APPLY actor it is sufficient to show that when such an actor is incorporated in a WFDFS:

I) $\forall\ i,j\ |\mathcal{R}(\ell_i) - \mathcal{R}(\ell_j)| < 2 \quad 0 \le i,j \le$ number of input links of $\mathcal{A}$

where ℓ_i is the i^{th} input link of $\mathcal{A}$
and $\mathcal{R}$: {input links} $\rightarrow \mathbb{Z}^*$
and $\mathcal{R}(\ell_i)$ = number of tokens that have arrived on the input link ℓ_i

Intuitively this means that all the input values for a given activation e of a procedure arrive before any of the input values for the $(e + 1)^{st}$. This is sufficient to ensure the correct (and safe) functioning of the SEQ actor and consequently of the whole implementation of MIMIC. The claim is that this restriction on acceptable token sequences is in fact satisfied for the input links of an APPLY actor $\mathcal{A}$ in any BL program. The ideas in the proof are fairly simple. Unfortunately, presentation of a rigorous proof gets bogged down in the details of precisely describing manipulations of program graphs. A detailed proof I) can be found in [19].

9. Conclusion

We have presented here several schemes of increasing capability for implementing procedure application in a large class of data flow processors. All of the schemes implemented the immediate copy rule - that is a data flow program with an apply actor is semantically equivalent to one where the APPLY has been replaced by the program graph of the procedure it invokes. This effect can only be achieved at runtime since recursive procedures and procedure variables are allowed. The schemes presented were efficient in the sense that the overhead in terms of the number of packets required to set up and terminate an activation was small. In addition only the pieces of the invoked procedure that were active were brought into the instruction memory. This helps economize the use of instruction memory space, especially in the case of data flow procedures with conditional components.

The key to the correct operation of the a call mechanism is that there is at most one outstanding procedure activcation per call mechanism. Consequently the state information that must be kept (e.g. which arguments have been passed, the name of the invoked procedure, the activation name etc.) is bounded. Thus rather than needing external tables, potentially of unbounded size, or some other mechanism to keep track of procedure calls, we can rely on the actors of the call mechanism themselves to hold this information.

Acknowledgements

I would like to thank Jack Dennis for his continued interest and help. I would also like to thank Dave Ellis, Clement Leung and Ken Weng for their useful suggestions.

BIBLIOGRAPHY

1. Amerasinghe, S. N. *The Handleing of Procedure Variables in a Base Language.* S.M. Thesis, Department of Electrical Engineering and Computer Science, MIT, Cambridge, MA. September 1972.

2. Arvind, and Gostelow, K. *A New Interpreter for Data Flow and Its Implications for Computer Architecture.* UCI Technical Report #72, Department of Information and Computer Science, University of California - Irvine, Irvine, CA. October 1975.

3. Dennis, J. B., and Fosseen, J. B. *Introduction to Data Flow Schemas.* CSG Memo 81, Department of Electrical Engineering and Computer Science, MIT, Cambridge, MA, September 1973.

4. Dennis, J. B., and Misunas, D. P. *The Design of a Highly Parallel Computer for Signal Processing Applications.* MAC TR101, Department of Electrical Engineering and Computer Science, MIT, Cambridge, MA. August 1974.

5. Dennis, J. B, and Misunas, D. P. *A Preliminary Architecture for a Basic Data Flow Processor.* MAC TR102, Department of Electrical Engineering and Computer Science, MIT, Cambridge, MA, August 1974.

6. Dennis, J. B., Misunas, D. P., and Leung, C. K. *A Highly Parallel Processor Based on the Data Flow Concept.* MAC TR134, Department of Electrical Engineering and Computer Science, MIT, Cambridge, MA, August 1974.

7. Dennis, J. B. "First Version of a Data FLow Procedure Language". *Lecture Notes in Computer Science, 19,* G. Goos and J. Hartmanis, Editors, Springer-Verlag, New York, NY, 1974.

8. Dennis, J. B. "Design and Specification of a Common Base Language". *Proceedings of the Symposium on Computers and Automata,* Polytechnique Press of the Polytechnic Institute of Brooklyn, 1971.

9. Flynn, M. "Some Computer Organizations and Their Effectiveness". *IEEE Transactions on Computers,* September 1972.

10. Fosseen, J. B. *Representions of Algorithms by Maximally Parallel Schemata".* S.M. Thesis, Department of Electrical Engineering and Computer Science, MIT, Cambridge, MA, June 1972.

11. Fuller, R. "Associative Parallel Processing". *Proceedings AFIPS Spring Joint Computer Conference,* 1967.

12. Hack, M. *Analysis of Production Schemata.* MAC TR94, Department of Electircal Engineering and Computer Science, MIT, Cambridge, MA, February 1972.

13. Hewitt, C. *Viewing Control Structures as Patterns of Passing Messages.* Working Paper 92, MIT, AI Laboratory, Cambridge, MA, August 1976.

14. Karp, R. M. and R. E. Miller "Properties of a Model for Parallel Computations: Determinacy, Termination, Queueing". *SIAM Journal of Applied Mathematics*, *14*, November 1966.

15. Keller, R. M. "Look-Ahead Processors". *ACM Computing Surveys*, vol 7, number 4, December 1975.

16. Kosinski, P. R. "A Data Flow Language for Operating Systems Programming". *Proceedings of ACM SIGPLAN-SIGOPS Interface Meeting*, SIGPLAN Notices, 8,9, September 1973.

17. Leung, C. K. *Formal Properties of Well-Formed Data Flow Schemas*. MAC Technical Memorandum 66, Department of Electrical Engineering and Computer Science, MIT, Cambridge, MA. June 1972.

18. Miranker, G. S. *An Approach For Proving packet Communications Architectures Correct*. CSG note-27, Department of Electrical Engineering and Computer Science, MIT, Cambridge, MA, September 1976.

19. Miranker, G. S. *Design and Correctness of a Data Flow Procedure Mechanism*. S.M Thesis Department of Electrical Engineering and Computer Science, MIT, Cambridge, MA, January 1976.

20. Misunas, D. P. *A Computer Architecture for Data Flow Computation*. S.M. Thesis Department of Electrical Engineering and Computer Science, MIT, Cambridge, MA, July 1975.

21. Misunas, D. P. "Procedure Representation in a Data Flow Processor", *Proceedings of the 1975 Sagamore Computer Conference on Parallel Processing*, August 1975.

22. Rodriguez, J. E. *A Graph Model for Parallel Computation*. TR-64, Project MAC, MIT, Cambridge MA, September 1969.

23. Rumbaugh, J. *A Parallel Asynchronous Computer Architecture For Data Flow Programs*. MAC Technical Memorandum 150, Department of Electrical Engineering and Computer Science, MIT, Cambridge, MA, May 1975.

24. Thurber, K., and Wald, L. "Associative and Parallel Processors". *Computing Surveys*, vol. 7 number 4, December 1975.

25. Weng, K. *Stream Oriented Computation in Recursive Data Flow Schemas*. MAC Technical Memorandum 68, Department of Electrical Engineering and Computer Science, MIT, Cambridge, MA, October 1975.