Emulating a Data-Flow Machine using a Network of Transputers

Moez Ayed and Jean-Luc Gaudiot

CENG 91-31

Department of Electrical Engineering - Systems
University of Southern California
Los Angeles, California 90089-2562
(213) 740-4484

April 26, 1991

Programming Multiprocessors

Emulating a Data-Flow Machine using a Network of Transputers

Ayed, Moez
April 26, 1991

1 Introduction

Thanks to the advance in VLSI technology, it is now possible to build Multiprocessors (MIMD) computers with many processors at a cheap cost. The major obstacle to the success of these highly flexible and powerful machines will be in terms of software cost. Indeed, an enormous effort is required to assure correctness and reliability of a large number of asynchronous communicating processes executing in parallel. The programmer has to deal with the very difficult task of partitionning the algorithm into parallel processes, map these processes to the processors and assure correct synchronization of the processes and data passing between them.

OCCAM, a powerful parallel programming language is a considerable advance on conventional programming languages. However, it still relies on the programmer to identify the logical parallelism in the algorithm and express it explicitly. Also, synchronization and data passing has to be written by the programmer explicitly. Finally the programmer has to find an efficient mapping of OCCAM processes to processors and take care of the routing between PEs.

If we can find a way to program Multiprocessors automatically without having the programmer suffer all the burdens mentioned above, then parallel architectures will be used much more efficiently. This requires a tool that identifies and implements parallel processes automatically and independently of the software engineer. It has been shown that the data-flow principle of execution which uses a data-flow graph as the machine language is a solution to the above programming problem.

In this project, we programmed a Multiprocessor of Transputers using OCCAM as a low level language.

The approach is to execute a variable resolution data-flow graph on a Multiprocessor of Transputers. More precisely, we constructed an emulator for a data-flow machine using our Multiprocessor. The data-flow computer that we emulated is influenced by the Monsoon Explicit Token Store (ETS) designed at MIT. We added some more features such as macro-actor processing, generation of stream of tokens used for parallel loops, vector processing and vector tokens. Each transputer in the multiprocessor is emulating a data-flow processing element.

2 Emulation vs Simulation of a computer system

2.1 Simulator

It is a software that behaves like the machine being simulated.
It can be run on any computer.
The simulation time is the one that we would get using the simulated machine.

2.2 Emulator

It is a combination of a software and a computer system (target machine) which behaves like the machine being simulated.
The software has to be run on the target machine.
The execution time is that of the target machine and not of the machine being simulated.

3 Token Format

token = <data,tag>
tag = <context,Instruction Pointer (IP),Iteration Identifier (II)>
context = Frame Pointer (FP) = frame base address
IP = instruction address = <instruction number (actor number),input port number>
data = scalar or vector
vector = an ordered sequence of scalers of the same type, with a maximum length of 100 elements.
type = character, integer or real

4 Overall structure of the project

The block diagrams are shown in figures 1 through 6

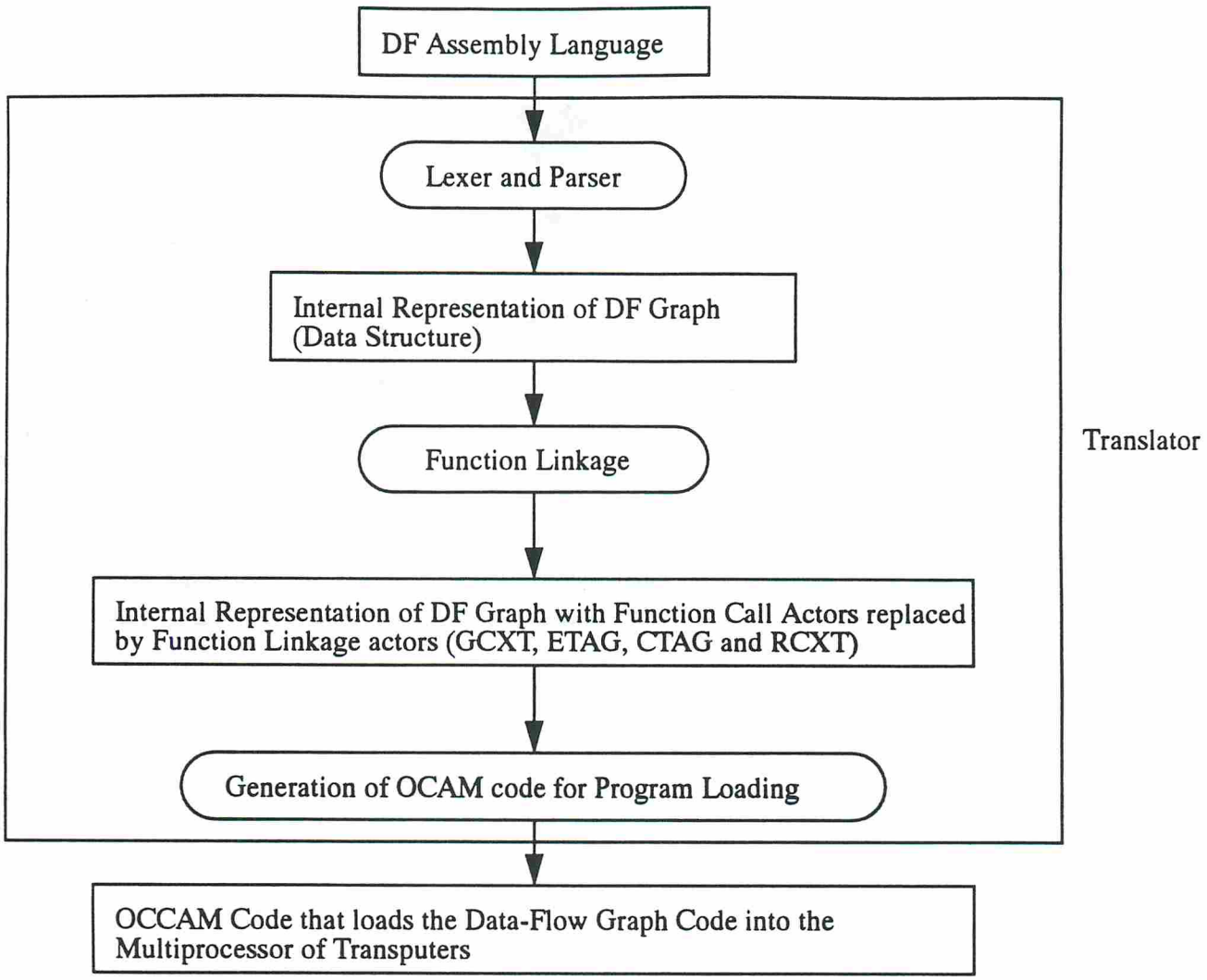


Figure 2: The Translator

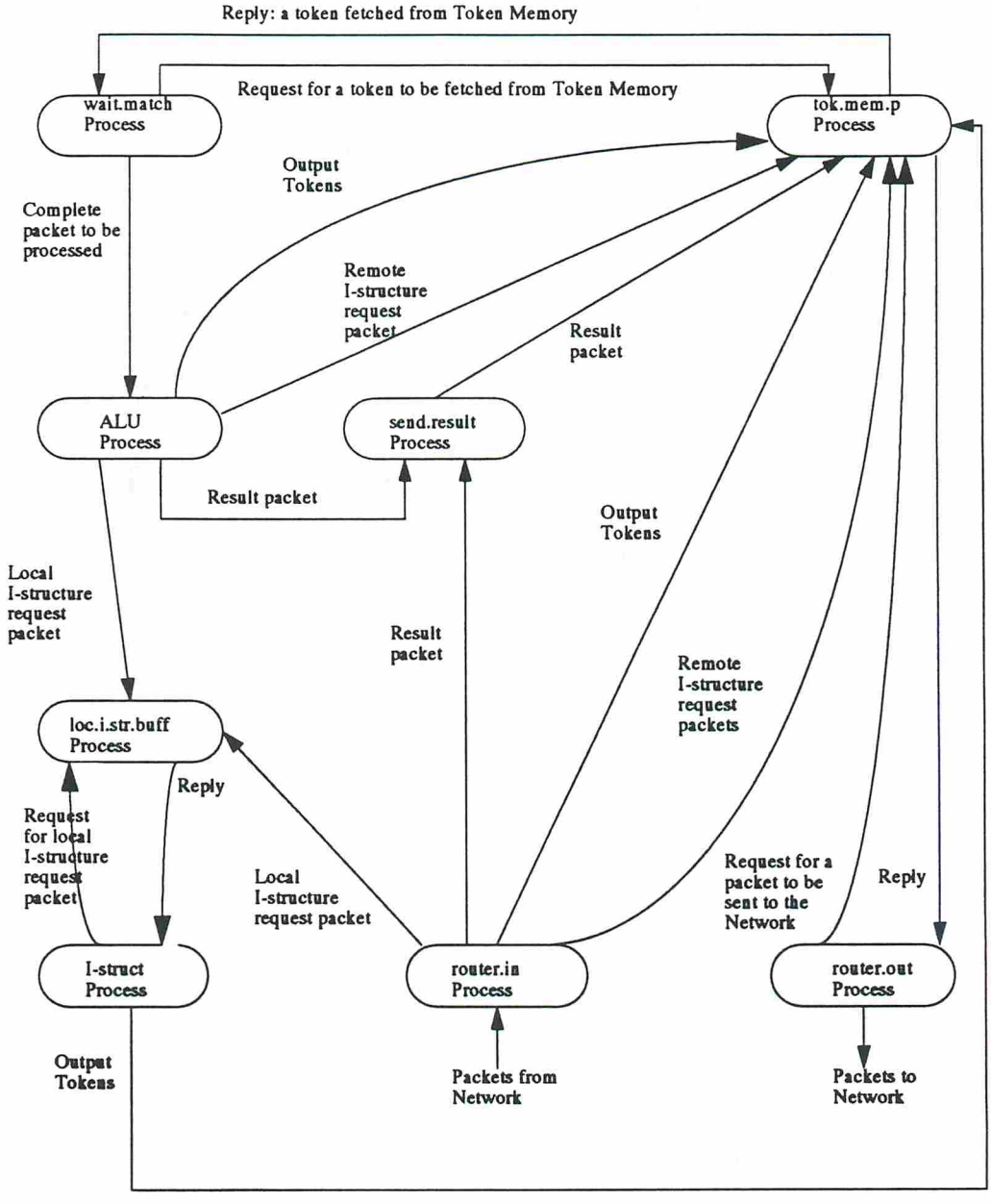


Figure 4: Emulator Engine

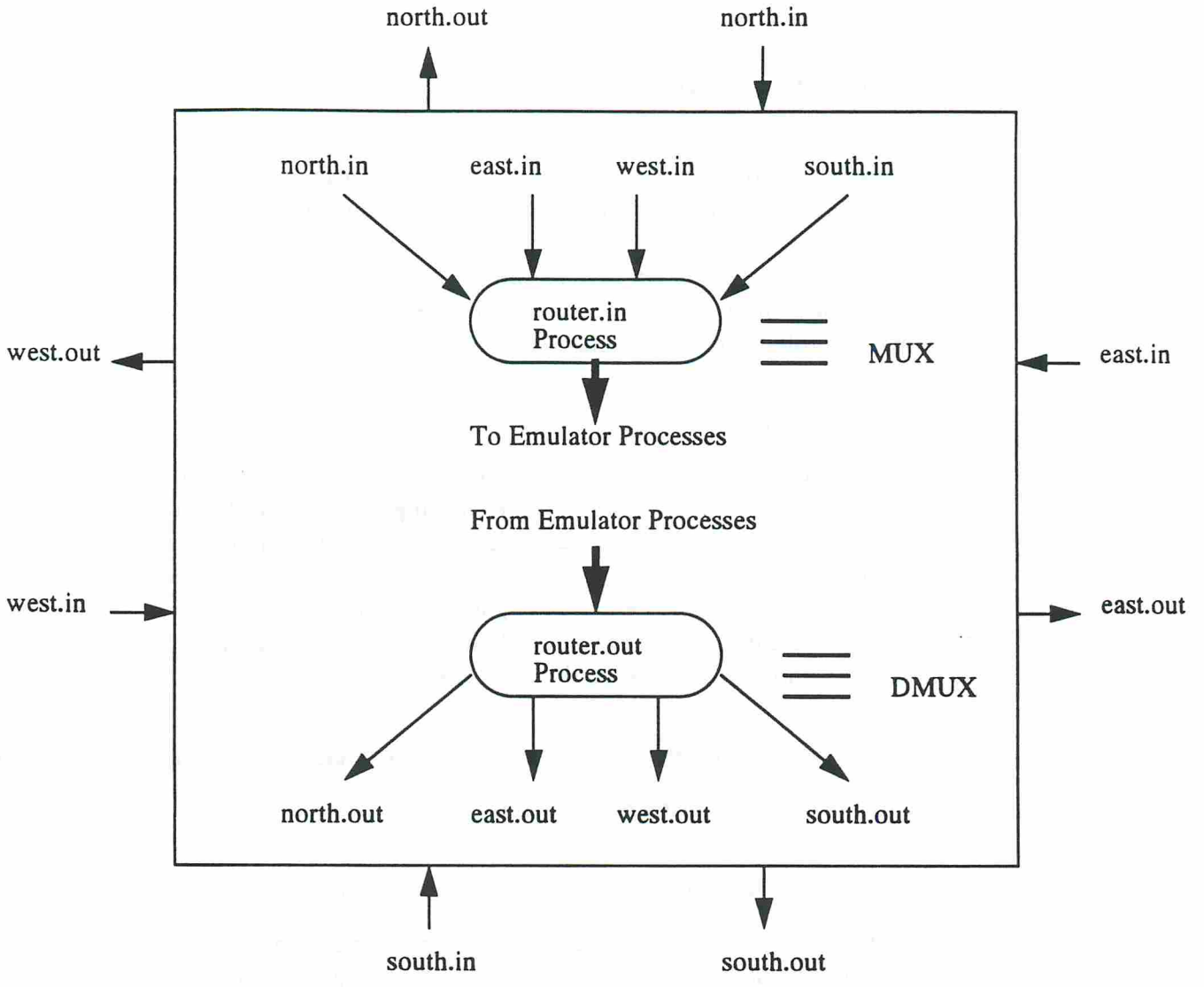


Figure 6: Router Processes as MUX and DMUX

4.5 Emulator Engine

All the special units used by the data-flow computer being emulated (Wait-Match unit, ALU unit, etc.) are implemented in software. There are 8 OCCAM processes that constitute the Emulator Engine. All of these processes are active in parallel by being combined by the OCCAM PAR construct and communicate via OCCAM channels.

4.5.1 wait.match process

This process implements the Wait-Match unit. In order to avoid deadlocks, the wait.match process receives a new token from the Token Memory only upon request. When it finishes processing a token, it then sends a request for a new token to the tok.mem.p process which is in charge of the access to the Token Memory.

4.5.2 ALU process

This is the process that implements the ALU unit. It receives a ready to execute packet from the wait.match process, processes it and then sends the result(s) to either the tok.mem.p process, the loc.i.str.buff process or the send.result process. The packets sent to the loc.i.str.buff process are local I-structure request packets. The packets sent to the send.result process are result packets which carry the output of the TPR actor which is the actor that prints out the contents of tokens for debugging purpose and to check the results of the program. The packets sent to the tok.mem.p process are either remote I-structure request packets to be sent to the network or output tokens to be stored either in the Token Memory or in a buffer containing output tokens destined to a remote PE and therefore should be sent to the network.

4.5.3 tok.mem.p process

This process controls all accesses to the Token Memory and all buffers storing packets to be sent to the network. There are 3 buffers that store packets to be sent to the network. One buffer stores the result packets. These packets are destined to the Root Transputer which doesn't contribute in any emulation work. Its main function is to act as an interface between the user through

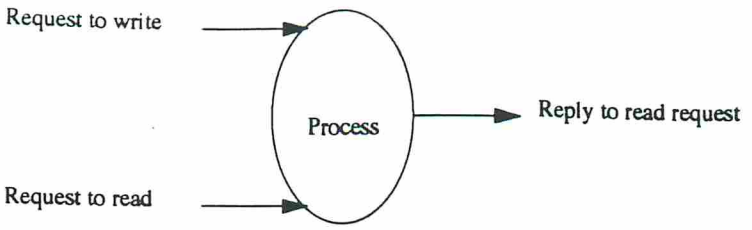


Figure 7: Process handling access to a shared data-structure

4.5.8 router.in process

This is the process that receives packets from the network. Depending on the kind of packet received, it is sent to loc.i.str.buff process, send.result process or tok.mem.p process. Result packets are sent to send.result process. Local I-structure request packets are sent to loc.i.str.buff process. Finally, remote I-structure request packets and output token packets are sent to tok.mem.p process.

4.5.9 Some OCCAM issues

- In OCCAM, 2 or more processes running in parallel cannot access the same data-structure if at least one of them writes to it. The main emulator engine processes (wait.match, ALU, etc.) are running in parallel and need to access some common data-structures (e.g. Token Memory). Therefore, we need to create a process which handles the access to the shared data-structure. Messages are sent to this process to request a read or a write access to the structure. In the emulator, tok.mem.p and loc.i.str.buff are two such processes.

- A process that handles access to a shared data-structure has the following OCCAM structure (see figure 7):

```
ALT
  write.request ? x1; x2; ... ; xn
    p1(x1,x2,...,xn)  -- process that handles the writing.
  read.request ? request
```

```
  p2(y1,y2,...,ym)  -- send the packet to the right process.
west.in ? z1; z2; ... ; zp  -- receive a packet from the
                            -- western physical channel.
  p3(z1,z2,...,zp)  -- send the packet to the right process.
south.in ? w1; w2; ... ; wq  -- receive a packet from the
                             -- southern physical channel.
  p4(w1,w2,...,wq)  -- send the packet to the right process.
```

5 Architecture of the Multiprocessor of Transputers

The network of transputers is shown in figure 8.

The dotted lines are physical connections to transputers that are not part of our multiprocessor, and therefore are not used. Our multiprocessor is part of a bigger multiprocessor that has a tree like configuration. Since it is being used by some other research group for some face recognition project, we could not use it in its entirety.
The transputers that are being used are of type T800. The architecture of a T800 transputer is shown in figure ?.

The Host Transputer does not participate in the emulation. It is mainly used as an interface between the user through a PC and the multiprocessor. Each of the 16 other transputers emulates a data-flow processor.

6 Software implemention of the various memories of the data-flow machine

Since arrays are the only data-structure provided by OCCAM, all memories are implemented using arrays.

6.1 Instruction Memory

6.1.1 Micro-Instruction Memory

The micro-instructions inside an actor are translated into an integer code with each component of the micro-instruction coded into an integer. A two dimensional array MICRO represents the Micro-Instruction Memory and stores all micro-instructions of all actors in the data-flow graph. MICRO[i] is an array of 4 elements of type INT16 storing one micro-instruction (see figures 9 and 10).

MICRO[i][0] = code for opcode = opcode number
MICRO[i][1] = code for operand1 if any, otherwise 0.
MICRO[i][2] = code for operand2 if any, otherwise 0.
MICRO[i][3] = code for operand3 if any, otherwise 0.
Note that there are no character or real operands. Only integer operands.

Coding of operands:

- Code for Ri[jk] is $(ijk)_{10} = i * 100 + j * 10 + k$
- Code for (Ri[jk]) is $(1ijk)_{10} = 10^3 + (ijk)_{10}$
- Code for immediate is the immediate itself. Hence the immediate has to be an element of the INT16 type.
- Code for a label is the number of micro-instruction preceeded by the label inside the actor. Numbering of micro-instruction starts from 0.
- Codes for conditions:
  EQ0 $\rightarrow$ 0; GT0 $\rightarrow$ 1; LT0 $\rightarrow -1$;
  GE0 $\rightarrow$ 10; LE0 $\rightarrow -10$; NE0 $\rightarrow$ 11.
- Code for (k) is k.
- Code for a number x is x itself (e.g. CON x,y).

6.1.2 Instruction Memory

Each actor in a data-flow graph has a unique number (no 2 actors even in different function bodies can have the same number). Actor numbers start

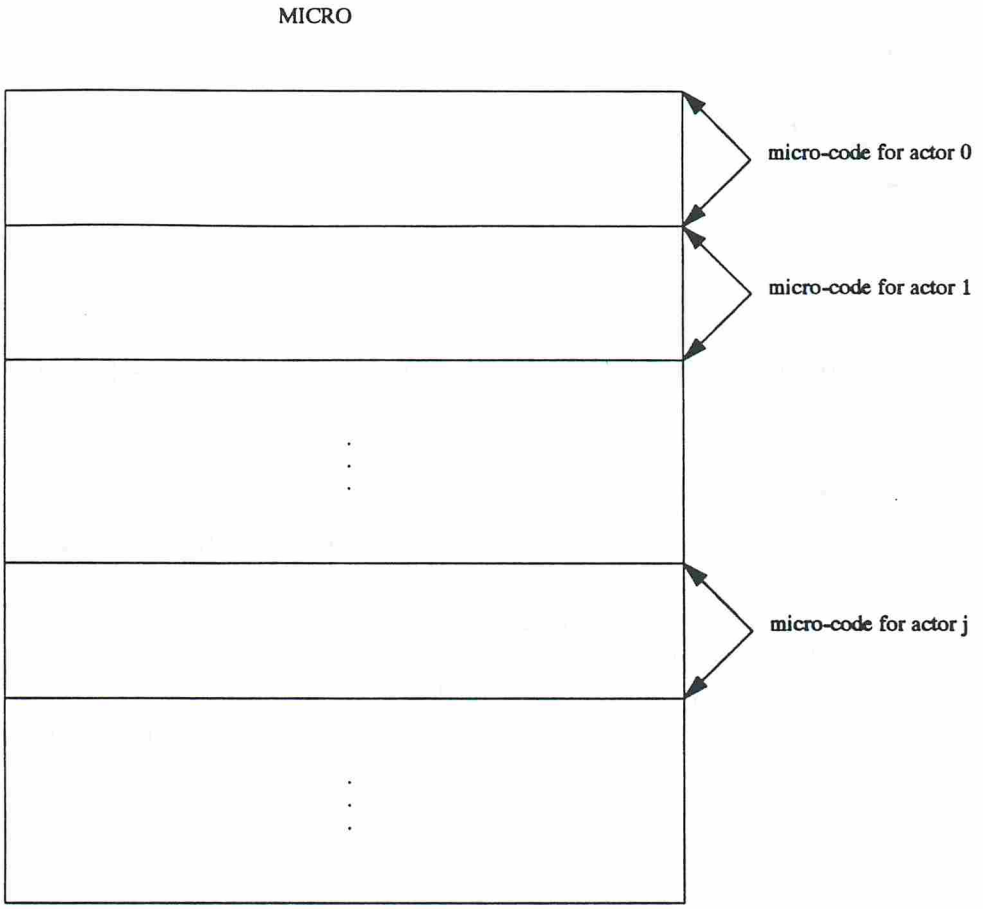


Figure 10: Micro-instruction Memory

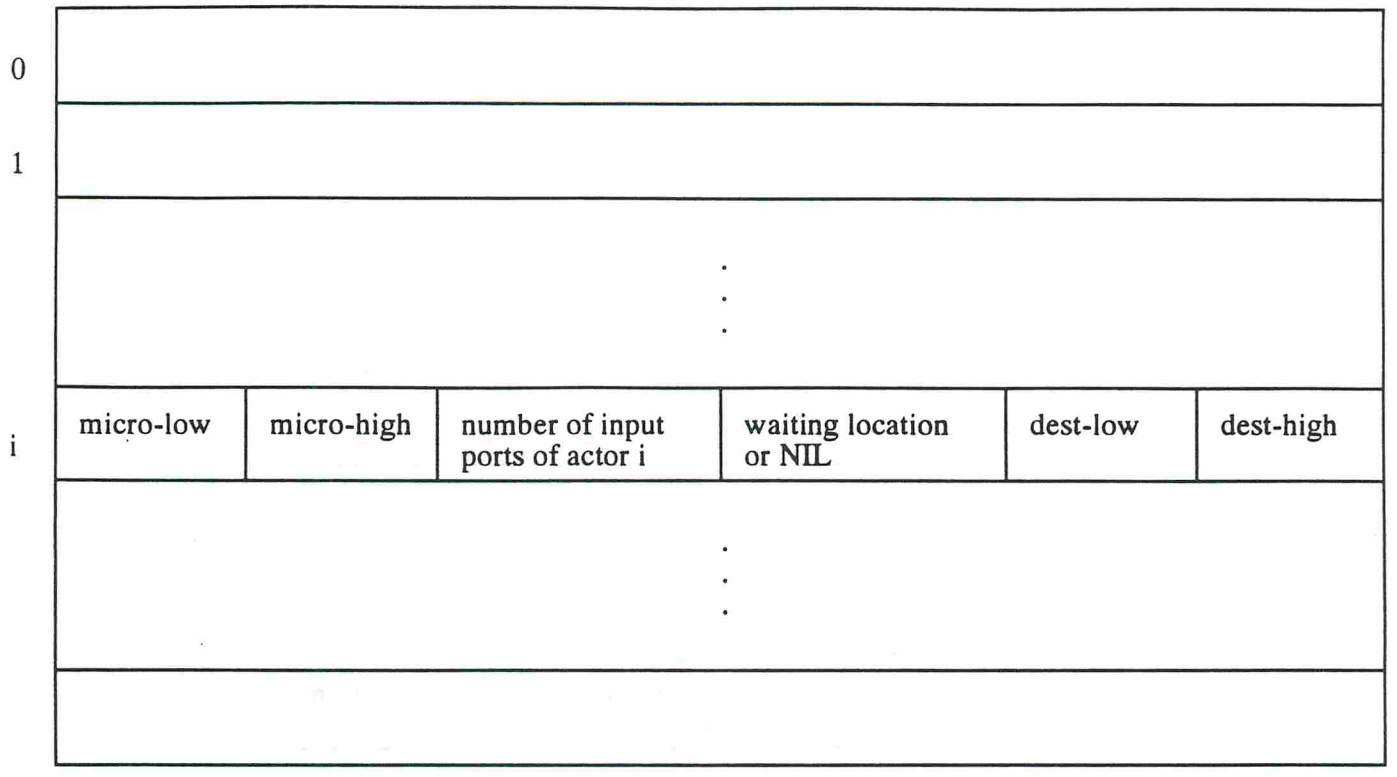


Figure 11: Instruction Memory

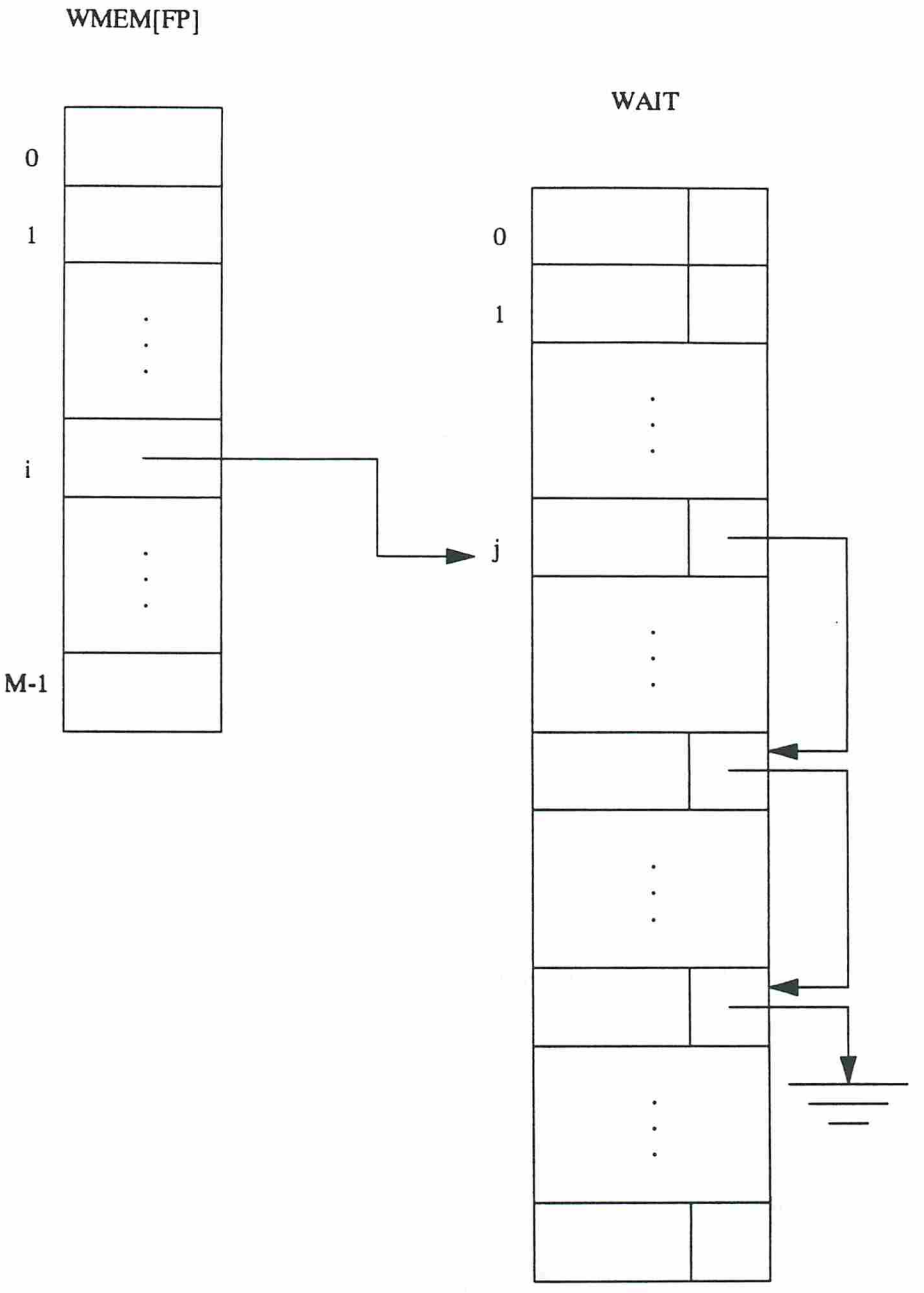


Figure 13: Waiting Memory

WAIT[j] = a 13-element array of type REAL32 storing a set of input tokens of some actor having the same II (and of course belonging to the same context).

Note:

- This mechanism of storing the waiting tokens causes an expensive list search when the matching is done.
- No hashing instead of list implementation was done because of the limitation in the memory size of the transputer. Hashing would create many unused holes in the memory space.

6.2 Token Memory

It is represented by 3 2-dimensional arrays TMEM0, TMEM1, TMEM2 (see figures 15 and 16).

Each TMEMi stores tokens of type i, where i is 0 for CHAR, 1 for INT and 2 for REAL. Each TMEMi is implemented as a *circular buffer*.
TMEMi[j] = a 6-element array that stores a token of type i.

Access of TMEMi:

- Storing a token:
  if high.ptr < (Tmemsize - 1) **then**
  1. add the token at location (high.ptr + 1);
  2. increment high.ptr;

  else if low.ptr > 0 **then**
  1. add the token at location (low.ptr - 1);
  2. decrement low.ptr;

  else TMEMi is full ⇒ overflow;
- Retrieving a token:
  if low.ptr > high.ptr **then** TMEMi is empty;
  else

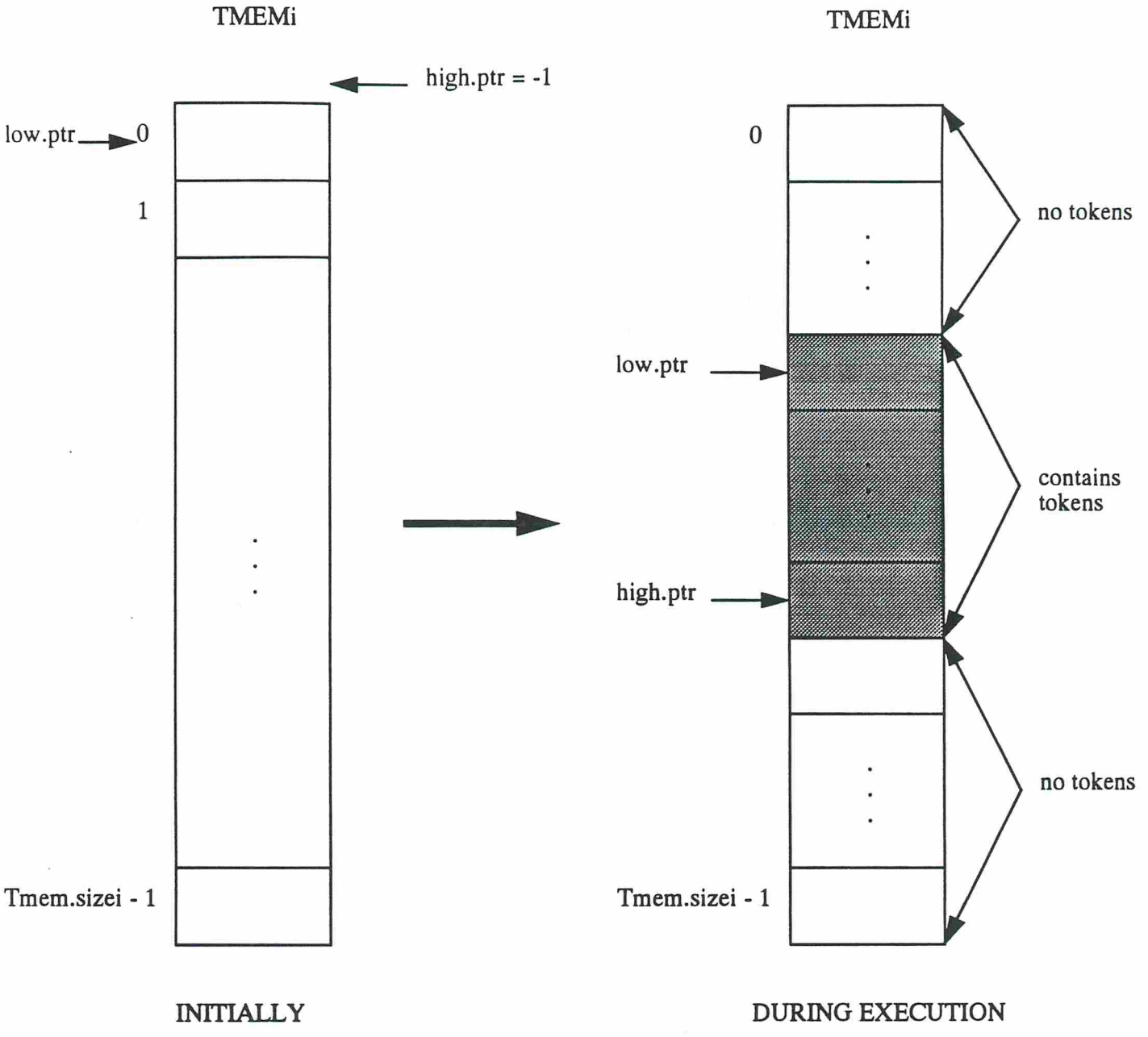


Figure 16: Token Memory as a Circular Buffer

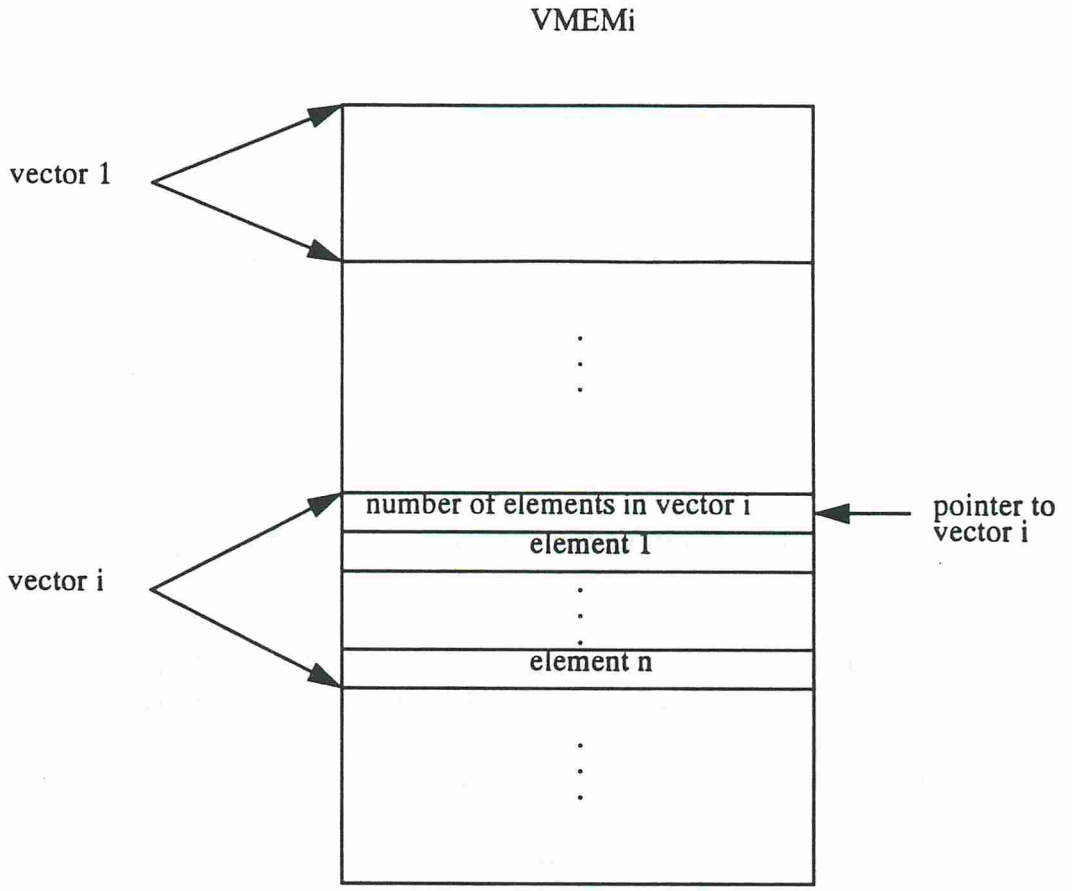


Figure 17: Vector Memory

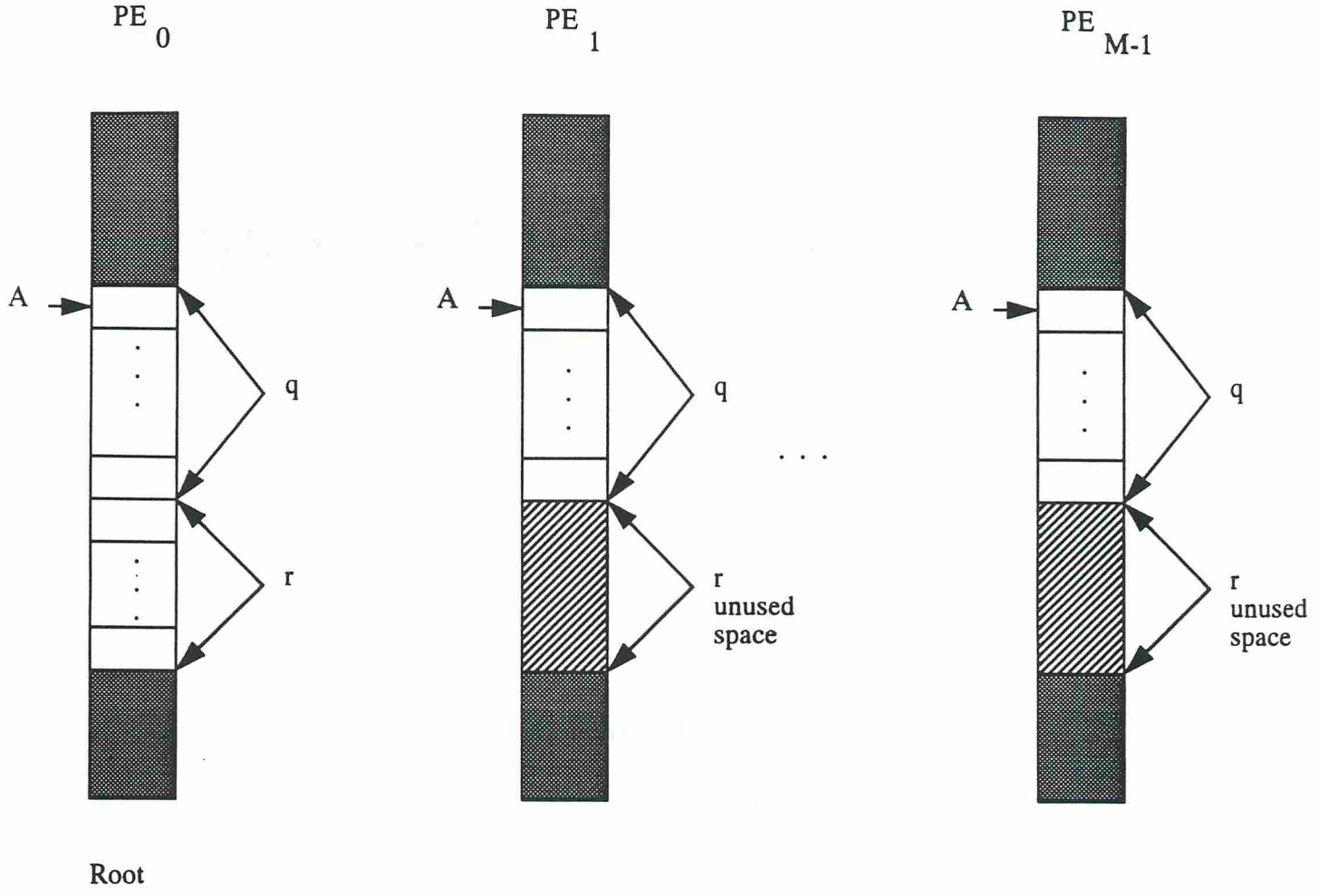


Figure 19: Distribution of an I-structure Array over the PE's

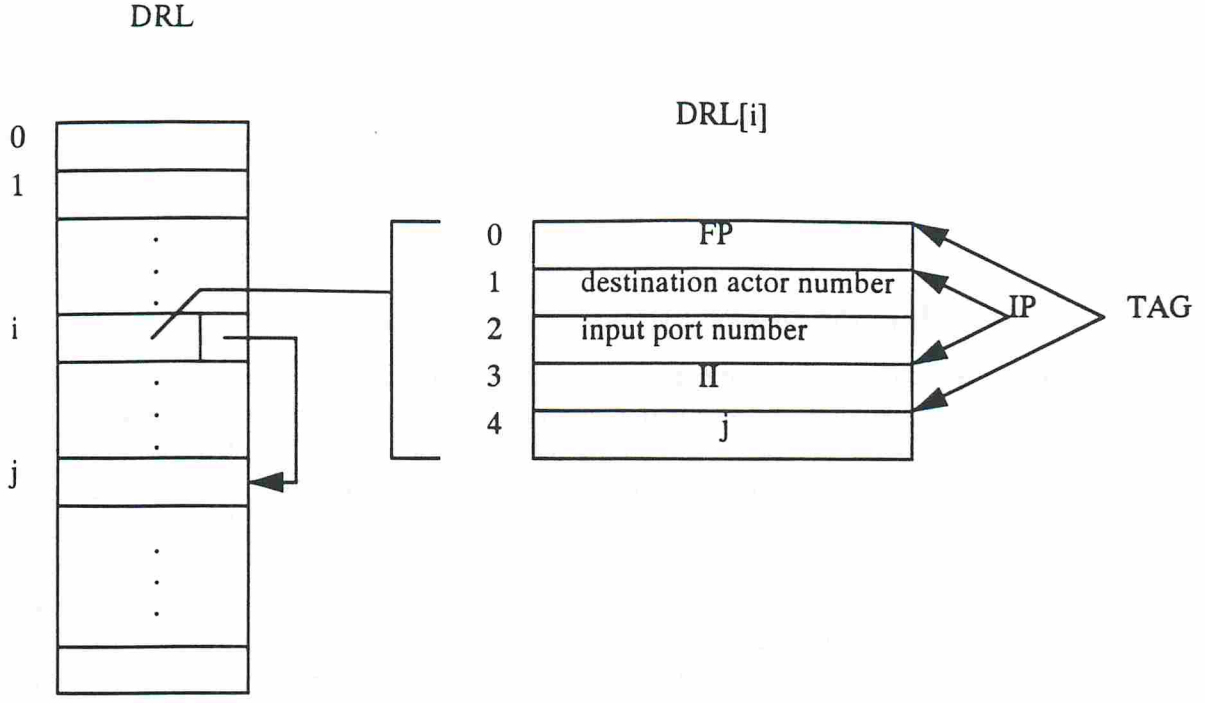


Figure 21: Deferred Read List

6.4.1 Deferred Read List

It is represented by a 2-dimensional array DRL that stores all the deferred read lists (see figure 21).

DRL[i] = a 5-element array storing a tag belonging to some deferred read list.
DRL[i][0] = FP.
DRL[i][1] = destination actor number.
DRL[i][2] = input port number.
DRL[i][3] = II.
DRL[i][4] = pointer to next tag (index within array DRL) belonging to the same deferred read list. If this is the last element in the list, then pointer = NIL (-1).

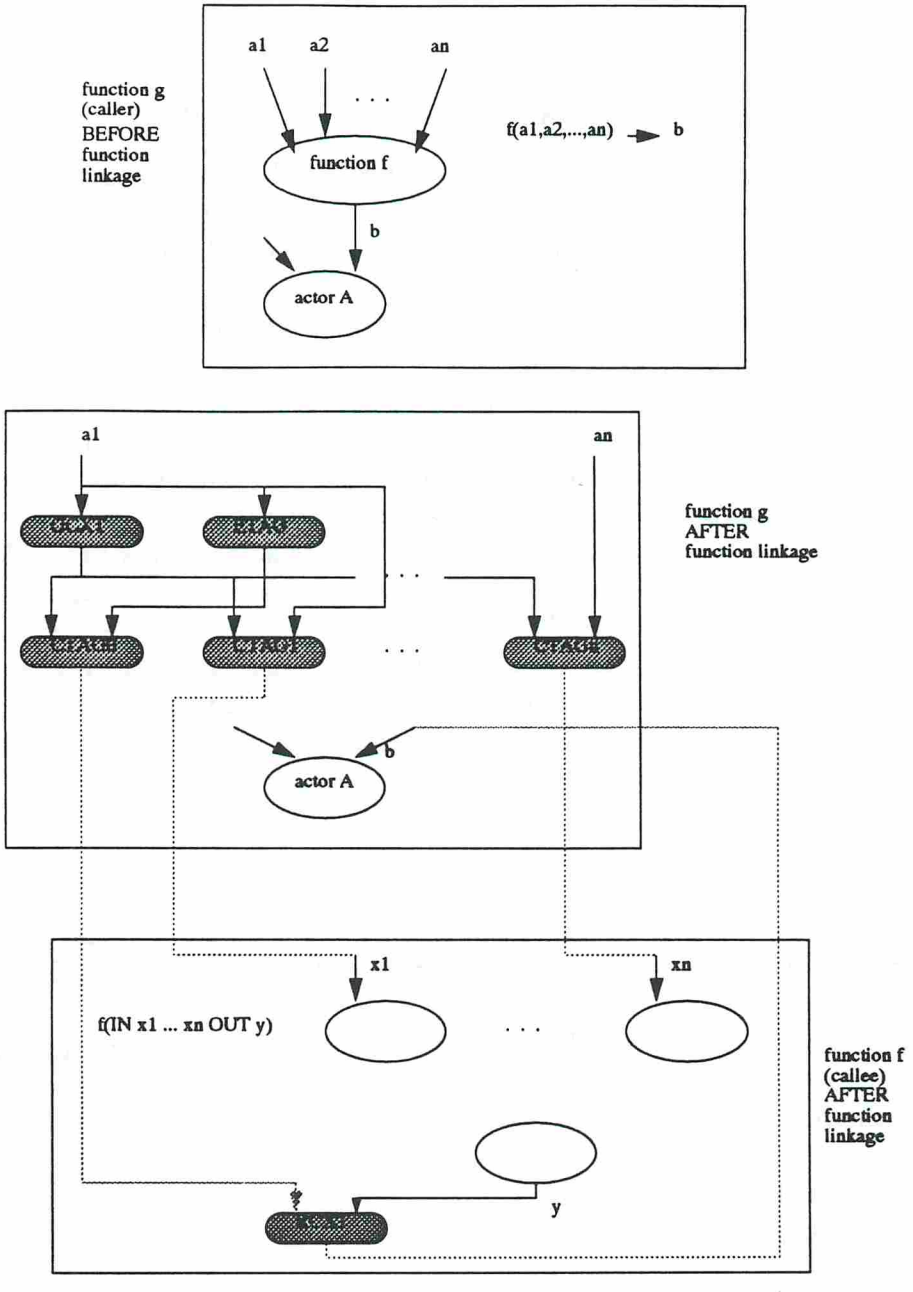


Figure 22: Function Call Linkage Mechanism

IP_r = address of actor where the result of the function call goes (actor A in the figure).

7.3 CTAGi actor

CTAG0 is the actor that sends the return address of the function call to the RCXT actor in the callee function body.
CTAGi for i $\neq$ 0 is the actor that sends actual input argument i to the corresponding input arc in the callee function body.

- input port 0:
  <new-FP,<FP,IP_1,II>>
- input port 1:
  <data,FP,IP_2,II>
- dynamic output:
  <data,new-FP,IP',II'>

$$II' = \begin{cases} -1 & \text{if } i = 0 \text{ (for wild matching)} \\ 0 & \text{otherwise} \end{cases}$$

$$IP' = \begin{cases} \textit{address of RCXT actor in callee body} & \text{if } i = 0 \\ \textit{address of actor in callee body where argument } i \textit{ goes} & \text{otherwise} \end{cases}$$

7.4 RCXT actor

It is used by the callee function to send the result token back to the caller function.
If dynamic memory management is done then this actor is also used to call the context manager to deallocate the frame used by this function invocation in the Waiting Memory.

- input port 0:
  <<FP,IP_r,II>,new-FP,IP_1,-1>
- input port 1:
  <result,new-FP,IP_2,any iteration number>
- dynamic output:
  <result,FP,IP_r,II>

OUT[i] is used to hold data destined to output port i in the same way as R[i] holds data on input port i.
A 2-dimensional array NUM is used for the output ports.
OCCAM declaration: [5][2]BYTE NUM:
NUM[i] is used for output port i.
NUM[i][0] = type (CHAR, INT or REAL) of data on output port i.
NUM[i][1] = number of elements in data (1 for scalar data and n for vector data of length n) on output port i, if there is such a port; otherwise it is 0.

8.1 Special purpose VECTORIZE actor

This actor is used to collect a number of scalar data elements and output a vector consisting of the elements collected. The vector has a maximum length of 100.

- input port 0: number of elements in the vector.
- input port 1: scalar elements to be vectorized.
- ouput port: vector.

The firing rule of this actor is a special one. As soon as a token on any of the 2 inputs arrives, the actor fires. This may or may not generate an output token (the vector being accumulated). The actor outputs a vector token when all elements of the vector being accumulated arrive. The Wait-Match Unit is not used by this actor.

This actor is not purely functional. It is implemented using 2 arrays VECTOR.TABLE and VECTOR.STORE (see figure 23).

VECTOR.TABLE is used to store the information about the vectors being accumulated.
OCCAM declaration: [table.size][5]INT VECTOR.TABLE:
VECTOR.TABLE[i] = an array storing information about one vector.
VECTOR.TABLE[i][0] = VECTORIZE actor number.
VECTOR.TABLE[i][1] = iteration number of the vector being accumulated.
VECTOR.TABLE[i][2] = total number of elements of the vector.
VECTOR.TABLE[i][3] = number of elements already arrived.
VECTOR.TABLE[i][4] = FP of the function invocation.

VECTOR.STORE is used to store the elements of all vectors being accumulated.
OCCAM declaration: [table.size][100]REAL32 VECTOR.STORE:
VECTOR.STORE[i] = array storing the vector whose entry (index) in VECTOR.TABLE is i.

8.1.1 Remark

Since we have a *distributed* memory multiprocessor (no shared memory), whatever is stored in the local memory of a Transputer cannot be seen by the other Transputers. Hence, all executions of this actor have to be done in a designated PE so that all elements of a vector being accumulated can be stored in the same Transputer. If the elements of a vector are distributed across the multiprocessor, then each time a PE executes this actor, it has to broadcast that news to all other PEs so that they update the entry for that vector in VECTOR.TABLE. Furthermore, when all elements arrive the vector has to be collected in one PE to be output. This obviously creates too much communication overhead.

9 Optimization Issues

9.1 Generating multiple tokens with different iteration tags

To implement loops using data-flow graphs we need to generate a set of input tokens with the same iteration tags for each iteration of the loop. In the traditional dynamic data-flow model designed at MIT, this is done by using the D and L actors inside a feedback path. This however introduces too much overhead into the data-flow graph. If the iterations of the loop are independent, we can use special actors that will generate all the input tokens to the loop with different iteration tags as a stream. This eliminates much of the overhead caused by the D and L actors. For this purpose 2 special actors GST and GSTVEC were created. GST generates a stream of scalar tokens and GSTVEC generates a stream of vector tokens with different iteration tags if desired.

method 1 is desired then the mapping function of actor execution to PEs is such that all GCXT actor executions are mapped to the Root Transputer (PE_0). If method 2 is desired then the mapping function maps the GCXT actor executions randomly using a hashing scheme.

9.3 Accumulating Scalers into Vectors

One way to do this is by creating 2 actors, an INI actor that initializes an empty vector and an ACC actor that accumulates the elements of the vector. The INI actor is monadic and receives the length of the vector to be accumulated and outputs an empty vector. The elements of this vector are preceeded by the length of the vector to be accumulated. This output vector goes to input 0 of the ACC actor. The ACC actor is diadic. Its input port 1 receives the elements to be accumulated. If all the elements of the vector being accumulated have arrived then the accumulated vector is output. Otherwise, the partially accumulated vector is fed back to input 0 of the ACC actor. The advantage of this method is that the actor is purely functional and unlike the VECTORIZE actor, it does not have to remember anything. Therefore any execution of this actor can be done on any PE. However the partially accumulated vector creates too much traffic and communication overhead when it is fed back. This will deteriorate the performance of the program. Hence, the VECTORIZE actor was chosen instead.

9.4 Mapping of execution of VECTORIZE actor

We can improve the execution of this actor by making it distributed across the multiprocessor rather than centralized.
The idea is to map all tokens that are destined to the VECTORIZE actor and that contribute to the accumulation of a vector (vector length and all elements of the vector) to the same PE and execute the initiation of the actor on that PE.

9.5 Choice of the designated PE

The designated PE is the Transputer were we execute the actors that have to execute on one PE only or those that we choose to execute on one PE only. For instance, the CREATE actor has to execute on a designated PE.

is the array descriptor of the newly created array. The vector includes the base, type and the sizes of the dimensions of the array. The output vector is needed for the SELECT and APPEND actors each time we want to select or append an element. If some array size in a program is 1000, then we have at least 2000 append and select operations in the program just for this array. Therefore, 2000 vectors are needed. This obviously will create too much traffic and communication overhead. Also, too much memory space will be needed to store the vectors.

A solution to this problem is to store the information about arrays (array descriptors) in a table in memory and keep a pointer to it. This pointer will be the output of the CREATE actor. However, since we don't have a shared memory system, we will have to keep this table in each Transputer because the APPEND and SELECT actors could execute on any PE. In order to have this, either the CREATE actor has to execute on each Transputer or each time the CREATE actor executes on a PE it has to broadcast the array descriptor to all other PEs. Obviously none of these methods is efficient.

Another solution for this problem is to perform the creation of arrays at compile time and eliminate the CREATE actor from the graph. Hence, the compiler from the high level language (e.g. SISAL) to the data-flow graph will be responsible for the task of creating arrays and allocating storage for them. A table is constructed which will store the information (array descriptors) about all the arrays created. The table will have one entry per array and the index of this entry will be used by the SELECT and APPEND actors to reference the array descriptors. Since these actors could execute in any PE, we need to load this table in all Transputers. If we have on the average 5 arrays per program, then on the average the table will have 5 entries. Therefore the storage needed for this table is very small.
A further optimization could be done by including the pointer to the entry in the table in the SELECT and APPEND instructions since it is a constant. This will eliminate the need to circulate this constant from iteration to iteration and the SELECT and APPEND actors will have fewer input arcs.

Another advantage of this scheme is the reduction in the number of actors in the data-flow graph, since there is no more need for the CREATE and

Number of PEs	Execution Time (ticks)
1	172,335
2	91,366
4	66,249
8	56,327
16	52,976

Table 1: Livermore Loop 3 (Using old array handling scheme)

Number of PEs	Execution Time (ticks)
1	97,769
2	56,822
4	43,885
8	38,345
16	36,471

Table 2: Livermore Loop 3 (Using new array handling scheme)

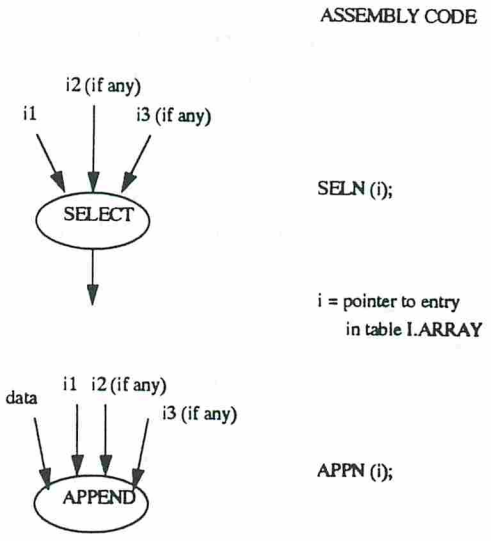


Figure 25: New APPEND and SELECT actors

9.8 Mapping Function

The complexity of the mapping function from actor executions to PEs affects greatly the execution time of the program and the performance.
As an example, we used 2 different mapping functions for the APPEND and SELECT actors. The first one is a simple random mapping (iteration number MOD N, where N is the total number of PEs). The second one maps the execution to the PE where the element to be selected or appended resides. One would expect that the second mapping would perform better, since it usually creates less communication overhead. However, experiments showed that the first mapping function performs better. This can be explained by the fact that the second mapping function is too complex and takes too much time to execute.
From this, we conclude that the mapping function should be chosen to be a simple one.

the program. In other words only the final result packet will be printed out, along with the total execution time of the program. Note that this will not affect the execution time of the program. It will simply make it faster to get the performance statistics of the program. Also, we can only do this after making sure that the program is working correcly, since by running the program this way we won't be able to look at all the results of the program.

There are two ways that could be used to solve the communication overhead caused by the TPR actor.
The first method is to send only one result packet to the TPR actor as soon as all the results of the program are computed. This will allow us to get the execution time of the program but will not enable us to check whether the program is correct or not, since we won't be able to see all the results of the program. Another problem with this method is that it is not always possible to determine when the program terminates without introducing too much overhead actors in the data-flow graph. For instance if the result of the program is an array, then we have to count all the elements of the array that have been appended in order to determine when the program terminates. This will introduce much overhead in the data-flow graph.
Another method is to always execute the TPR actor in the Root Transputer or in Transputer number 14. These 2 PEs are directly connected to the HOST Transputer. Therefore the result packets that are constructed on these 2 PEs need only one hop to reach their destination (the HOST). This obviously greatly reduces the communication overhead. Although this method forces the sequentialization of the executions of the TPR actor, experiments showed that the net result is an improvement over the distributed executions of the actor.

9.10 Routing data between PEs

It might be better to collect larger packets before sending them to the network to reduce the communication overhead. Instead of sending individual tokens to remote PEs, we could collect several tokens destined to the same PE and send them all together in the same packet.

Number of PEs	Execution Time (ticks)	Speedup
1	1,094,086	1
2	509,818	2.15
4	380,395	2.88
8	347,986	3.14
16	329,952	3.32

Table 3: Livermore Loop 3 (Arrays have 1000 elements)

Number of PEs	Execution Time (ticks)	Speedup
1	4,118,301	1
2	1,784,179	2.31
4	1,289,441	3.19
8	1,131,949	3.64
16	1,056,266	3.90

Table 4: Livermore Loop 3 (Arrays have 2000 elements)

Number of PEs	Execution Time (ticks)	Speedup
1	2,561,218	1
2	1,121,560	2.28
4	542,026	4.73
8	351,335	7.3
16	217,121	11.8

Table 7: Livermore Loop 3 (Arrays have 1800 elements)

Number of PEs	Execution Time (ticks)	Speedup
1	3,142,578	1
2	1,414,888	2.22
4	642,847	4.9
8	390,382	8.05
16	235,075	13.4

Table 8: Livermore Loop 3 (Arrays have 2000 elements)

Number of PEs	Execution Time (ticks)	Speedup
1	1,456,369	1
2	606,290	2.4
4	323,051	4.51
8	221,228	6.6
16	171,305	8.5

Table 9: Livermore Loop 1 (Arrays have 1000 elements)

Number of PEs	Execution Time (ticks)	Speedup
1	5,329,509	1
2	2,038,499	2.61
4	916,090	5.82
8	548,003	9.73
16	367,733	14.5

Table 10: Livermore Loop 1 (Arrays have 2000 elements)

TPR actor executions were mapped to the Root. The GST actor executions were mapped to the designated PE. Finally, the executions of all remaining actors in the graph were mapped randomly (iteration-number MOD number-of-PEs).
The Execution time and the speedup for this program are shown as a function of the number of PEs in tables 9 and 10.

10.3 Livermore Loop 12

This is a first difference program. The SISAL program follows.

```
function Loop12(n:integer; Y:OneDim; returns OneDim)
  for i in 1,n returns
  array of Y[i+1] - Y[i]
  end for
end function  %Loop 12
```

Number of PEs	Execution Time (ticks)	Speedup
1	1,102,437	1
2	539,836	2.04
4	284,447	3.88
8	197,670	5.58
16	157,549	7.0

Table 13: Matrix Addition (Arrays have 1000 elements)

Number of PEs	Execution Time (ticks)	Speedup
1	4,133,499	1
2	1,847,578	2.24
4	836,624	4.94
8	493,083	8.38
16	338,334	12.22

Table 14: Matrix Addition (Arrays have 2000 elements)

```
  for i in 1,n returns
  array of X[i] + Z[i]
  end for
end function  % matadd
```

The best performance results were obtained using the following mapping of actor executions to PEs. The APPEND and SELECT actor executions were mapped randomly (iteration-number MOD number-of-PEs). The designated PE was chosen to be Transputer number $(number - of - PEs)/2$. The TPR actor executions were mapped to the Root. The GST actor executions were mapped to the designated PE. Finally, the executions of the add actor that adds the elements of arrays X and Z were mapped randomly (iteration-number MOD number-of-PEs).
The Execution time and the speedup for this program are shown as a function of the number of PEs in tables 13 and 14.

The execution time of a program run using the emulator is much larger than if we execute the program on the Transputers directly using OCCAM (it was about 100 times larger for a Matrix Addition program). The reason for this is that all the hardware units of the machine being emulated are implemented in software (emulator engine). Since the model that we are emulating is quite complex, the overhead introduced is quite large. Hence whenever we execute a program using the emulator, we have to run a very large software program (emulator engine) in addition to the program that we are trying to execute.
The hardware units that exist in a data-flow machine are very specialized. They are designed in such a way as to cope with the overhead introduced by the data-flow execution model. The goal of the data-flow research community is to design data-flow computers with comparable performance with the existing Von Neuman Multiprocessors. The advantage of the data-flow computer would be an implicit and easier detection of parallelism and synchronization and therefore better programmability. If it is hard to make a data-flow machine give comparable performance with conventional multiprocessors due to the overhead that exists with the data-flow model, then it is to be expected that emulating a data-flow machine would give very slow absolute execution time. Since our transputer Multiprocessor only interprets data-flow programs, the absolute execution time is expected to be slow. However, the relative execution speeds would be more meaningfull.
Note that if the program that we are running is quite large and comparable in size to the emulator engine then the absolute execution time of the program using the emulator or directly using OCCAM will also be comparable. Due to the limitation in the memory of the transputers and the fact that the representation of the memories of the data-flow machine being emulated is done using software (data-structures), when we try to run very large programs the memory of the transputers is overritten. That is why we are enable to do experiments with very large size programs.

12.1 Comparaison between Occamflow and Emulator

Consider a program that we want to execute.

1. Using Occamflow:
   SISAL program $\rightarrow$ IF1 graph $\rightarrow$ P = program written in OCCAM.

13 Usefulness of the Emulator

If the goal is to have an efficient way to program the transputers, then the emulator is not the right approach. This is because no matter how simplified the emulator is, the emulator program is very large and therefore the execution time of a data-flow program using the emulator is quite big.
The merit of this emulator is to show the performance of the macro data-flow computer that we emulated. Mainly, we showed the advantages of using large grain (macro) actors, fat tokens (vectors) and vectorization code. Also we showed how important the mapping of actor execution to PEs is and we gave some useful mapping techniques for some data-flow programs. Furthermore, parallel loops are implemented differently from the way they are implemented in a tagged token data-flow machine. We don't use any D and L actors and instead, we use a special purpose actor that generates the stream of tokens needed for the different iterations of the loop. Some simple optimization issues are also shown to improve the performance of the machine. Finally, we demonstrated the need to have an efficient data-flow graph and how this can improve the performance of the algorithm greatly. This means that the compiler that generates the data-flow graph has to be very powerful so that the output generated can be exploited efficiently under the data-flow model of execution.
From this project, we can conclude that OCCAM is a very convenient language to emulate data-flow graphs. Indeed, OCCAM processes are executed upon arrival of data on their input channels. This is very similar to the data-flow model of execution. In addition, OCCAM processes are very convenient to represent hardware units such as the Matching unit and the ALU unit. The communication between these units is very nicely done using OCCAM channels.
It is probably safe to say that using OCCAM and the transputers is a very efficent way to emulate any distributed memory multiprocessor system.

14 Statistics gathered by the Emulator

In OCCAM, two or more processes combined by the PAR construct cannot share a data-structure if at least one of them writes to it. Because of this restriction, we could not represent the components of the data-flow proces-

average. This result may have to be duplicated if it has to be sent to multiple actors or we could have more than 1 result if the actor being processed has more than 1 output. However 1 actor is processed each time the last stage in the pipeline fires. Note that there are some actors (SELECT and APPEND) where this does not apply because of the split phase characteristic of their operation.

If we have a total of I instructions in the data-flow graph, then the total execution time can be estimated to be

$t = [(a+3) + (I-1) * a] * T$

where T is the clock period of the pipeline.

The $(a+3) * T$ term is the time needed for the first instruction to complete (or to fill up the pipeline).

The $(I-1) * a * T$ term is the time needed to complete the remaining $(I-1)$ instructions.

$a = \frac{\sum^{number\ of\ input\ arcs\ of\ all\ actors}}{total\ number\ of\ actors}$

$T = T_m + T_l$

T_m = the maximum time delay of the circuitry of any of the 4 pipeline stages.

T_l =time delay of each interface latch introduced between each 2 stages of the pipeline.

Note that I is not the total number of actors in the graph. It is rather the sum of the number of executions of all actors in the program. Since this number is usually very large for a reasonable size program, we can approximate the total execution time of the program to

$t = I * a * T$.

A single transputer:

The wait-match process produces a packet every a tokens it receives. Also, no more than one stage of the pipeline executes at a time. The four processes representing the pipeline are combined as follows:

```
PAR
  WHILE TRUE
    wait.match()
  WHILE TRUE
    instruction.fetch()
  WHILE TRUE
    ALU()
```

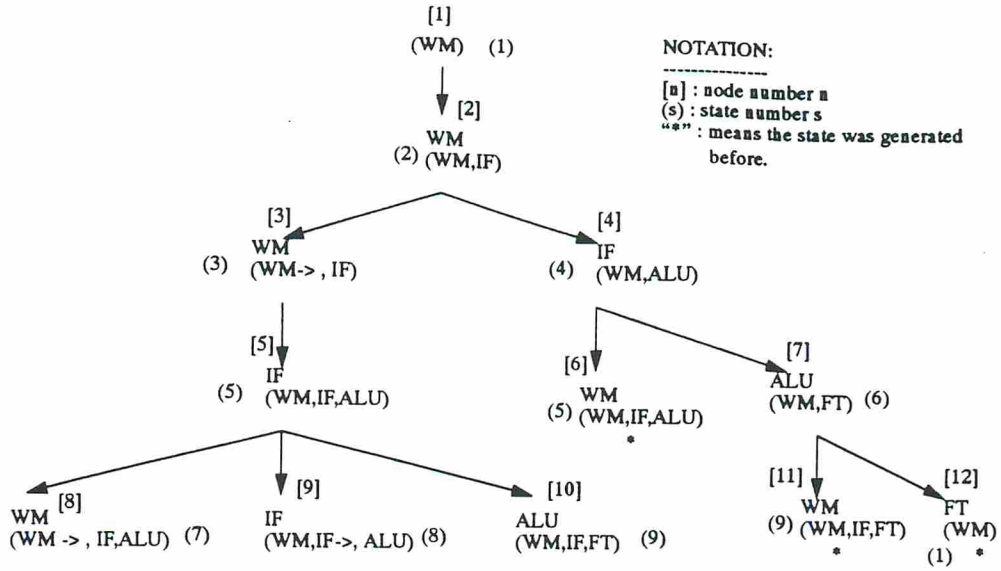


Figure 26: Execution Tree

Since each of these processes is inside an infinite WHILE loop, at any time, it is either waiting for an input, waiting for an output or active (either being executed or in the pool of active processes).
Let the state of our machine be the set of processes which are active or waiting for an output. A process P which is active is represented by P and a process Q which is waiting for an output is represented by Q→.
In general, when a process starts executing, it will not run until completion. Indeed, when a process has been running for 2 timeslices (1 timeslice = 1024 ticks of the high priority clock) continuously, the processor will attempt to deschedule it (this is the case when the process is running at low priority, which is the default), and another process is taken from the active pool and scheduled. Since the execution time of each of our 4 processes is much larger than 2 timeslices, when more than one process is active, the execution will alternate between these active processes.
One way to estimate the execution time of a program executed by the emulator is to construct the execution tree shown in figures 26 through 30.

Each node of this tree consists of a process P and some processes in parentheses which represent a state S of the machine. The interpretation of a node

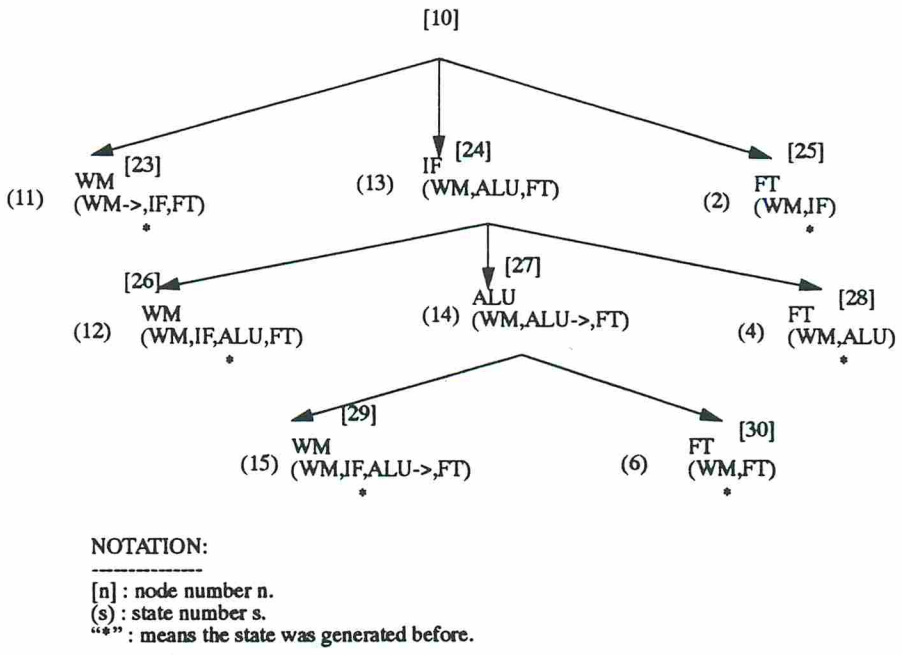
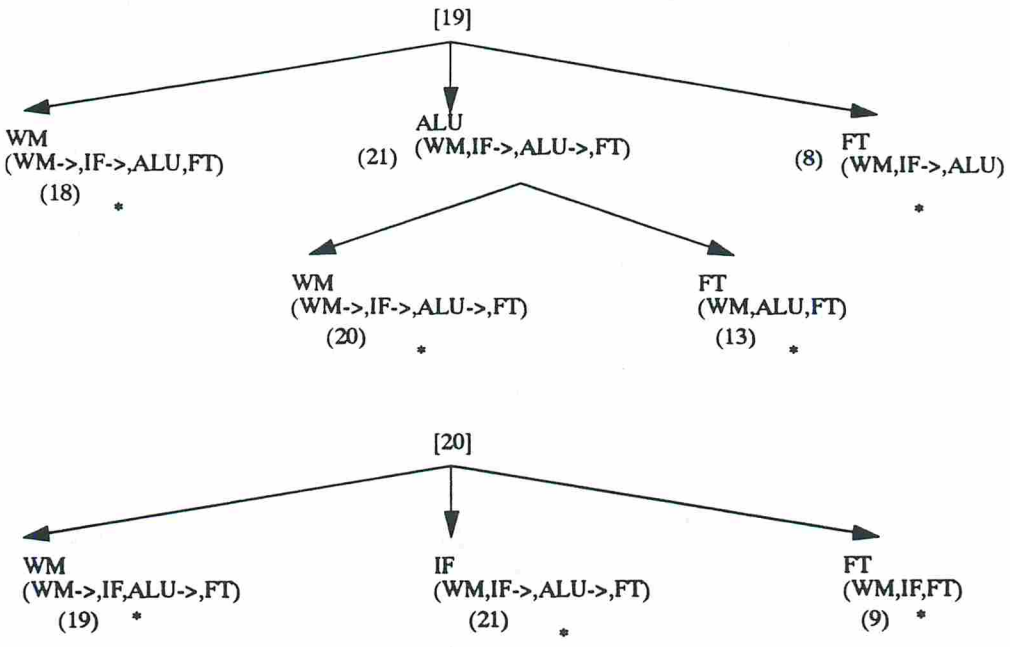


Figure 28: Execution Tree



NOTATION:

(s) : state number s.
"*" : means the state was generated before.

Figure 30: Execution Tree

with process FT (i.e. the first instruction has just finished execution) and travel along that path until we find another node labeled with process FT (the second instruction has just finished execution) and so on. The time taken to go from one node to the other is the sum of the execution times of all processes that label all nodes along the path connecting the 2 nodes (obviously this excludes the processes inside the states in the nodes). If we know the number of paths in the tree, we can take the average rate over all paths. However, we have an infinite number of paths in the tree. Therefore we decided to consider the worst case execution time.
Let's assume that the maximum length of any path connecting 2 consecutive nodes labeled by process FT in the tree is d. Then $d * t_{WM}$ is the worst case rate of execution of the instructions after the first instruction has executed, assuming that t_{WM} is larger than the execution time of any of the 3 other processes. Thus we can approximate the total execution time of a program having I instructions to (ignoring channel communication time)
$t_e = T_0 + (I-1) * d * t_{WM}$,
where T_0 is the worst time taken for the first instruction to finish execution. Since the WM process outputs a packet every a tokens it receives
$t_{WM} = a * t(WM)$,
where t(WM) is the execution time of the WM process. Thus
$t_e = T_0 + (I-1) * d * a * t(WM)$,
Since $T_0 = c_1 * t_{WM}$, where c_1 is some integer constant, then $t_e = (c_1 + (I-1) * d) * a * t(WM)$
For a reasonable size program I will be quite large compared to the small constant c_1 and therefore we can omit the c_1 term. We get
$t_e = (I-1) * d * a * t(WM)$, which can be approximated to
$t_e = I * d * a * t(WM)$.

Result: $\frac{t_s}{t} = c$, where c is a constant.

If we use more than 1 processor, the number of instructions executed in a data-flow processor or the corresponding transputer emulating it is the same. This is because the mapping of actor executions to PEs used in the data-flow computer being emulated and in the emulator is the same. Thus *the ratio of the execution time of a program using the emulator to the one using the data-flow computer being emulated is approximately equal to a constant for any number of PEs used*: