an individual basis with each of its PRM's. The process of individual communication always terminates with a synchronization message to all PRM's of a task. This message is sufficient to cause the PRM's to exit the micro-code loop and re-synchronize with the other PRM's of the task.

4.4.2 Process Destruction

Another cause of communication failure is that of process destruction, where a process is destroyed before or during the attempted communication [GEN 81].

It can also be seen that this type of communication failure cannot occur. The system provides facilities for intra-task and inter-task communication. Intra-task communication obviously present no problem since tasks are basic units of execution and are treated as one entity. Thus they are created or destroyed as a whole.

The inter-task communication failure due to process or task destruction cannot occur because tasks are created and destroyed only by the Job Monitor. Thus is done at a job level. That is, all the tasks of a job are

created at the same time and later destroyed at the same time at the end of job. A task will logically and physically exist throughout the lifetime of a job. There is never a need for inter-job communication since, if it arises, both jobs can me merged into one job and communication will now be at the inter-task level.

4.5 Conclusions

We have described in this chapter the basic Process Communication System for a Reconfigurable Network Operating System. The varistructure and reconfiguring properties of RNA's make it necessary not only to support Inter-task Communication but also Intra-task Communication, and among processes or tasks whose configuration is only known to the PRM at execution time. By having one of the processes named the TH-PRM, process communication becomes independent of task configuration. Also we have explained the nature of possible communication failure and how the system is capable of resolving this situation

Chapter 5

Reconfiguration

5.1 Introduction

RNA's create logical computer systems from primitive processors and memory units by binding communication paths or sets of processor/memory components. This binding may be changed during the execution of a process. Programs executing in a physical processor must execute correctly independent of configuration. This implies that the PRM must be able to map any communications between logical processors correctly independent of physical configuration.

The concept of reconfiguring a logical computer system permits the task to have a configuration which reflects the natural computation structure of the problem [BRO 82]. No longer will a problem be mapped on a rigorous fixed architecture, but now the architecture can be configured to conform as much as possible with the structural demands of the problem.

The PRM will be responsible for performing reconfiguration on a temporary basis. That is, the task will eventually reconfigure itself to its original structure. Permanent reconfiguration cannot be accomplished without the intervention of the upper-levels of the Operating System since data global to the entire system must be updated. In this respect the PRM will only be able to detect the instances when this type of reconfiguration is required.

This chapter will focus on the different types of reconfiguration, namely Local and Global Reconfiguration, as well as on the detection of situations where reconfiguration of a task is needed.

5.2 Need for Reconfiguration

During the lifetime of a task there are three situations in which reconfiguration is required. These arise upon:

- occurrence of a page-fault.
- explicit request at JCL level.
- hardware failure.

A page-fault will require that the task be

reconfigured so that page transfers can occur between the backup devices and the corresponding memory module. The reconfiguration is needed since backup devices are not directly accessible by all processors. Several processors will share a backup device for page storing. The task has to be reconfigured to group together the processors which share such a backup device so paging operations can be performed. Operations on different backup devices can be carried out in parallel by different tasks. Once the paging situation has been handled the task will be reconfigured to its original structure for normal processing. This situation in which the task is broken up into smaller tasks has duration only while the paging situation is cleared. It is therefore a temporary situation which can be handled locally by the TH-PRM. This will be called Local Reconfiguration.

The other two situations, explicit requests at JCL level and hardware failures, will require that the task be reconfigured and the new structure will acquire a permanent status, at least until either of these two events occur again. The fact that the new structure acquires a permanent status will require the intervention of the upper-levels of the Operating System since global data and new resources are involved in the operation. For

this Global Reconfiguration situation the TH-PRM will limit its action to detect the events and communicate the situation to the JM.

Figure 5-1 shows the situations in which reconfiguration is required. In summary, Local Reconfiguration is characterized by:

- reconfiguration can be done locally by the PRM
- upper-levels of the Operating System do not need to know the task is being reconfigured
- there is no change in the physical structure of the task (no resources are either added or deleted from the task)
- the reconfiguration is temporary

Global Reconfiguration situations are characterized by:

- the PRM can only detect the need for such requirements
- the upper-levels of the Operating System have to perform the reconfiguring operation
- there will be a physical change in the structure of the task (i.e. processor/memory modules will either be added and/or deleted from the task).
- the new structure of the task is permanent

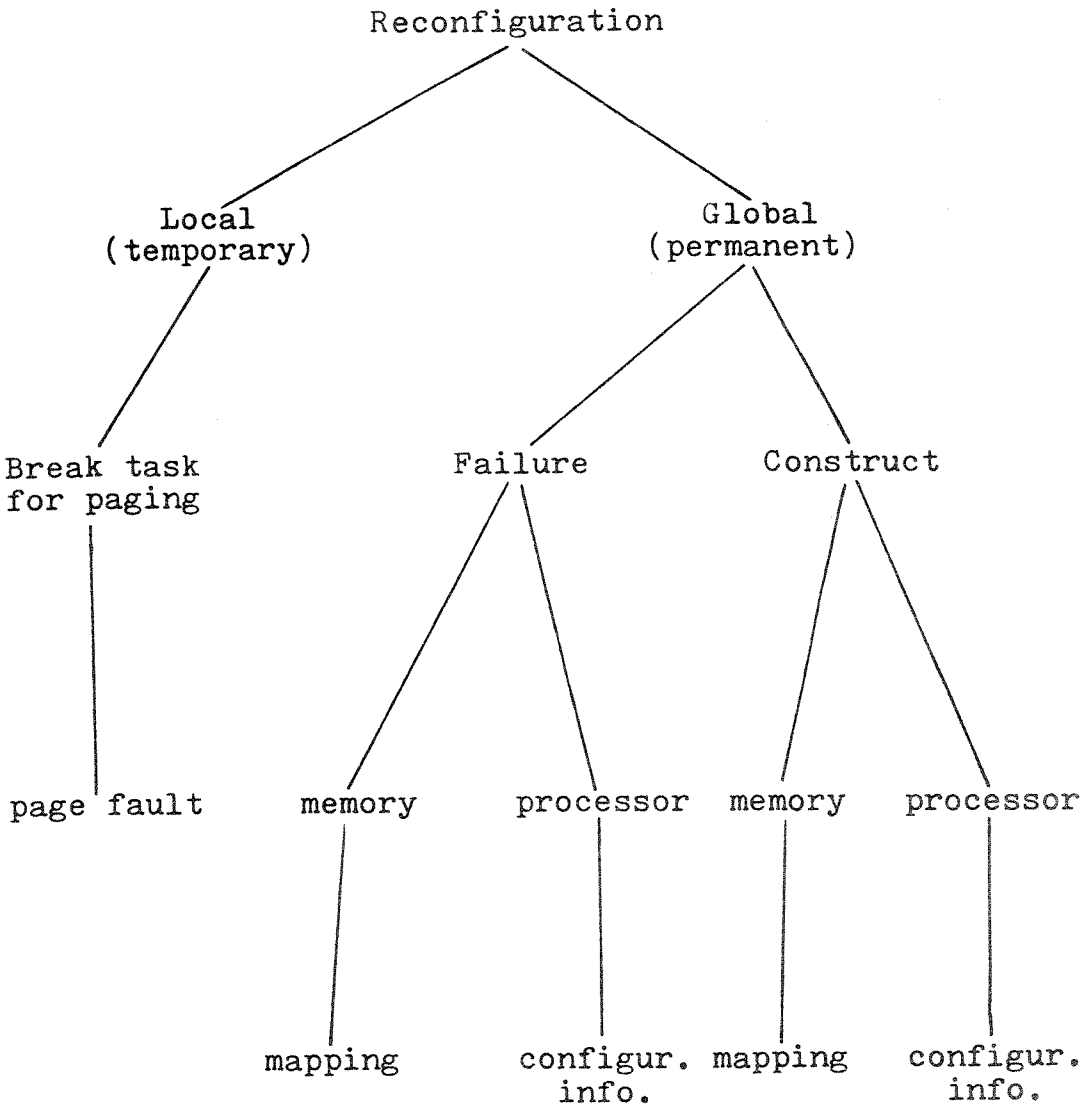

Figure 5-1: Reconfiguration situations.

5.3 Local Reconfiguration

As we mentioned in Section 5.2, Local Reconfiguration is required when a page fault occurs. The task will be broken into smaller tasks for the page-in, page-out situation to be cleared. The principal characteristic which distinguishes a Local Reconfiguration is the non-intervention of the upper-levels of the Operating System, leaving the situation in total control of the PRM.

Before discussing the actual reconfiguration procedures let's examine how the TH-PRM recognizes and detects the presence of a page-fault.

5.3.1 Page-fault Detection

Within a task all processors will operate in SIMD mode. Memory allocation becomes more sophisticated in parallel reconfigurable varistructured systems executing in this mode [DeG 80].

In multi-width SIMD tasks, memory can be viewed as a two dimensional matrix where each column has the same bit width as the processors [DeG 81]. Each processor in the task is responsible for managing one column of the

matrix. Whenever a page must be swapped-out, that page must be swapped-out of each column. Similarly data swapped-in is reflected in changes taking place in all of the columns. This is true for data memory, but not for instruction memory which is linear even though it may be distributed along several processors of the task.

For this reason it must be possible to distinguish when a page fault occurred due to instruction or data fetch. Even if the mechanism for paging in/out is the same, we must differentiate between page faults occurring in two-dimensional memories from page faults occurring in one-dimensional memories, because the procedure for determining if a page fault did occur is different for each case.

Independent of the type of memory, all processors generate their access requests in lockstep mode. The memory module which has the requested address responds to the processor while the other memory modules would respond with a fault. All processors will broadcast their memory module statuses to all other processors through the broadcast bus along with any data if the fetch occurred in the instruction fetch cycle. A non-fault signal received through the broadcast bus will indicate that a page fault

did not occur since one of the processors did not fault. If no non-fault signal is received, a page-fault occurred. At this point it becomes necessary to distinguish if the access was to a two-dimensional or to a one-dimensional memory, for which hardware support is required.

For the one-dimensional case, it is sufficient that at least one processor did not fault. In this case the processors can continue their execution in lockstep mode. If a fault occurred, the TH-PRM must make a decision as to which page, if any, must be swapped-out. It will also search its tables to determine which processor owns the backup device where the requested page resides. The TH-PRM will try to reconfigure so that the processors involved in the transfer (page-in and page-out operation) will share a memory module. If the reconfiguration is possible, the TH-PRM will send the necessary information to both PRM's involved and the transfer is initiated. If reconfiguration is not possible, the transfer will have to be made through NIGM. Once the transfer is made, the TH-PRM will update its tables and normal execution resumes.

If a fault occurs in a two-dimensional memory all processors must have faulted and a page must be swapped-in

for every processor. All processors will stop upon receiving the page-fault signal and the TH-PRM will take control. The TH-PRM will again decide upon page-outs, reconfiguration and page transfers. Normal execution will resume once all the transfers have been carried out and the TH-PRM has updated its tables. The actual paging mechanism will be discussed in Chapter 6. The next section will discuss how reconfiguration of the task is accomplished.

5.3.2 Task Reconfiguration

After a page-fault occurs and the type of access which caused it (i.e. one-dimensional or two-dimensional memory fetch) has been detected as explained in Section 5.3.1, the TH-PRM will proceed to decide which page must be swapped out. The actual page replacement is not of interest here, but it will be assumed that there will be hardware support to easily implement an LRU page replacement algorithm [KAP 80].

It is not known a priori the availability of backup devices. It is not known until load time how many backup devices there are and with which processors they will be associated. The number of backup devices assigned

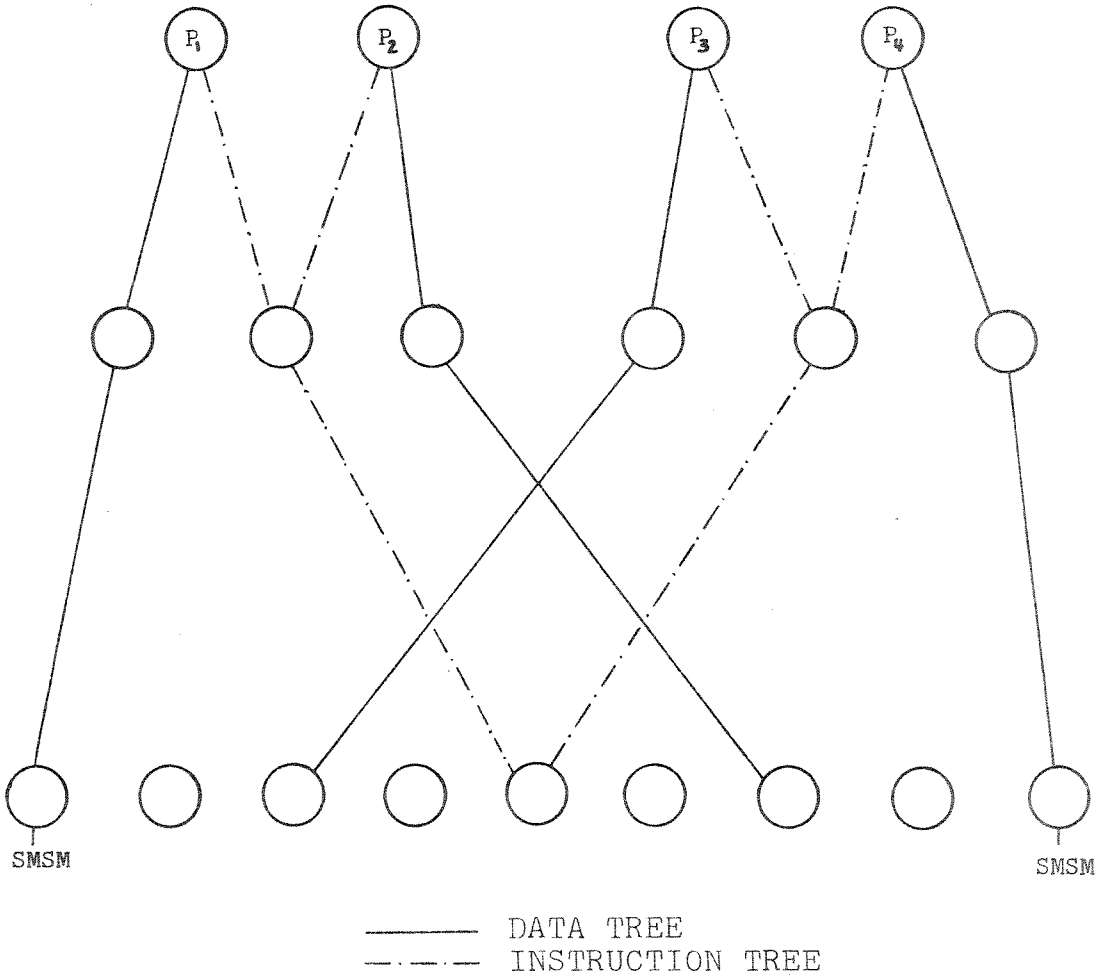

Figure 5-2: A four physical processor task

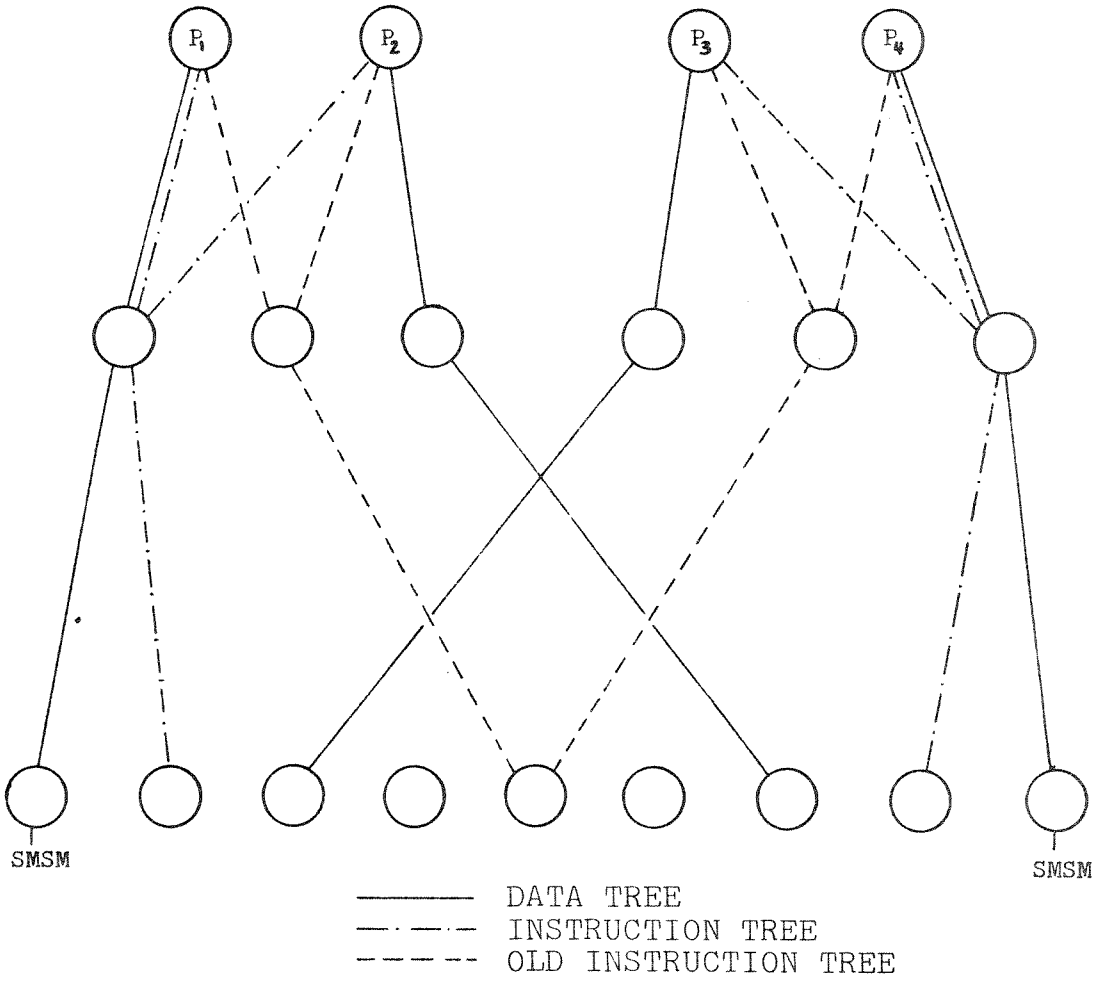

Figure 5-3: Two paging tasks

they will update, in the shared memory module, the number of active paging-tasks and will de-activate the local instruction tree. The last paging task to complete will set this counter to zero, meaning that all paging-tasks are through, and thus the original task must be re-constructed. To do so, the last paging-task will re-activate the original instruction tree and the original task is ready to proceed as an SIMD task.

5.4 Global Reconfiguration

Global reconfiguration is required under two circumstances --first, when the PRM detects a hardware failure and, second, when there is an explicit request at JCL level. In either case the PRM will stop the execution of the current task and the Job Monitor will be notified of the situation. It is up to the upper levels of the Operating System to take the proper action.

5.4.1 Hardware failure

The PRM is capable of detecting a hardware failure when either a page fault occurs on page zero or when an invalid message is received by the TH-PRM.

Each processor has a private memory module where

the PRM resides in page zero. The PRM resides entirely in this module and therefore a page fault should never occur while performing a fetch on the PRM's address space. When the PRM is in the process of determining the type of page fault, the presence of a page fault on page zero is an indication of a hardware error which the PRM is not capable of resolving. The situation is communicated to the JM which will take the proper actions.

The other instance of hardware failure detection arises when an inconsistent message is received by a PRM. In Chapter 3 the Communication System with its Upward and Downward Consistency Checking mechanisms was introduced. This system guaranteed that only valid messages would ever leave a PRM, and therefore only valid messages should be received. The arrival of an invalid message would imply that somewhere during the transmission of the message an error occurred in the network. This situation will indicate the presence of a hardware failure in the System. Again, the PRM is not able to resolve the situation and must report the failure to the upper levels of the Operating System for proper action to be taken.

5.4.2 Explicit request at JCL

During the lifetime of a job, a task may execute to completion several times, and each invocation of the task could require a different configuration. The hardware requirements are explicitly declared at the Job Control level [BRO 80b, FED 80]. Therefore reconfiguration of the task is required. At the time reconfiguration takes place, the task is inactive, so there is no possible intervention of the PRM. This situation is mentioned here for completeness of the discussion on Reconfiguration but its details are out of the scope of this research.

5.5 Self-testing mechanisms

The ability of a task to reconfigure itself, and the characteristics of RNA's of having the modules of the Operating System replicated and distributed among several processors, present a wide range of self-testing capabilities. In our context the self-testing mechanisms fall into the category of those Operating System functions capable of detecting hardware failures. By no means are we claiming that these procedures are fail-safe mechanisms, since, at the PRM level, action is limited to detecting these anomalies and no attempt is made to solve the problem and continue with the execution of the task.

One of the mechanisms will take advantage of the intra-task communication capabilities of the system and one will take advantage of the ability of a task to break itself up into paging tasks (see Section 5.3.2). The last mechanism discussed will take into consideration the replication and distribution characteristics of the PRM.

5.5.1 Time-out protocol

The first mechanism will rely on the Communication System to test for the correct performance of the hardware. Section 4.3 discussed intra-task communication where processors of the same task were able to communicate among themselves. Of special interest is the NIGM communication among processors. To check for hardware failures the TH-PRM will issue a SEND message to all processors of the task. Each PRM will respond by sending the contents of the object which was specified in the SEND message. The TH-PRM should allow for a reasonable amount of time to receive replies from all processors. If after this timeout occurs there are some replies pending, the TH-PRM can conclude that there is a hardware malfunction. The execution of the task must then be stopped and appropriate notice must be sent to the JM.

It should be noted that this self-testing mechanism does not have to be executed as such but can form part of the Communication System since it is expected, due to the nature of RNA's, that intra-task communication will occur very frequently.

5.5.2 Task break up

To be able to handle a page fault the task has to be broken into paging tasks. This implies the conversion of instruction trees into shared memory trees and the creation of new instruction trees for each paging task.

The TH-PRM can check for the creation and conversion hardware by breaking the task into paging tasks. Once the paging tasks have been created, the original task will be restored. No actual paging will take place, but only local task reconfiguration will occur. Hardware failures are detected if the task cannot be reconfigured to its original state. The TH-PRM can perform this check at predefined intervals if actual paging has not occurred.

5.5.3 TH-PRM rotation

The two self-testing mechanisms introduced in the previous sections are valid if there is no failure in the hardware where the TH-PRM resides. This is true because the TH-PRM must initiate the test and must validate its results. Any failure in the processor/memory where the TH-PRM executes/resides can produce unpredictable results. To avoid this problem the TH-PRM can be assigned to another processor/memory pair of the task before any of the tests is run or it can be done at a predefined interval of time.

The decision as to which PRM of the task will have the functions of the TH-PRM is software controllable. An arbitrary processor is selected at load time. Besides additional functions the TH-PRM is characterized by its ability to process IGM. This is so because at any time a PRM can recognize itself as being a TH-PRM. PRM's who are not TH-PRM's will mask their execution while processing interrupts.

Since the assignment of the TH-PRM is software definable, it is a trivial task to re-assign the TH-PRM to another processor by simply modifying a specific memory

location used to designate the TH-PRM. Besides this the JM must also be informed of the change so that new IGM can be routed to the new processor.

For the re-assignment of the TH-PRM to another processor to perform correctly the code for all PRM's must be identical. This being true, only execution masking during interrupt processing distinguishes between actual execution of a PRM and the TH-PRM. Since code replication throughout the processors is essential, so must be the data structures which support the execution of the TH-PRM. This motivates the use of a Message Table approach as opposed to a Capability List for the message system. The Message Table is independent of the type of software which manipulates it, being up to the software to recognize itself as a PRM or TH-PRM. The use of a Capability List is dependent on it being accessed by a PRM or a TH-PRM so rotating the TH-PRM among different processors could not be accomplished without reflecting this change on the Capability List.

5.6 Conclusions

In this chapter we have outlined the situations when Reconfiguration is required on RNA's. Also, the two types of possible reconfiguration have been introduced, namely Local and Global Reconfiguration. Local Reconfiguration is handled by the PRM's of the task while Global Reconfiguration must be resolved by the upper-levels of the Operating System since permanent changes in the architecture are required.

The capability of RNA's to dynamically change the topology upon which a task is executing, combined with the intra-task communication facility, present a wide range of possible self-testing mechanisms which can be carried out by the TH-PRM. The three self testing mechanisms discussed in this Chapter have the advantage that they use existing Operating System functions to perform the diagnostics. The time-out protocol is based on the Communication System, the task breakup is based on the Paging System, and the TH-PRM fully uses the execution masking of the syste

Chapter 6

Paging

6.1 Introduction

The paging problem in traditional architectures is bounded by the page selection and page replacement algorithms. A page is selected for replacement, is swapped out and a new page is brought in. In RNA systems the paging problem is augmented by the existence of two-dimensional memories and by the availability of backup devices.

A task does not know the topology of its configuration until load time. Different execution requests for the same task may execute on different hardware configurations. Depending on the topology of the hardware upon which a task is executing, processors may or may not have access to backup devices where its pages are stored. For this reason paging tasks must be created.

This Chapter will concentrate on the paging

problem for RNA systems. The paging mechanism and the effects that different topologies have on it will be discussed.

6.2 The Paging Problem

The scope of the paging problem in RNA systems is amplified by the presence of two-dimensional memories and by the availability of backup devices. The existence of two-dimensional memories implies that more than one page must be replaced. That is, more than one page must be swapped out and more than one page must be swapped in.

The availability of backup devices implies that not all processors of a task will have access to such devices. This means that some processors are not able to replace pages in their memory modules. To be able to transfer pages in these cases paging tasks are created. A paging task was defined as the smallest execution unit capable of performing page in and page out operations. In practice a paging task is a one processor task with access to a backup device. Paging tasks will execute in parallel among them unless they share the same backup device, in which case they must compete for the use of the device.

A paging task must perform the paging operations for all processors which have their pages stored in the backup device and who do not have access to the device. We will define a paging group as a group of processors whose pages are stored in the same backup device, but only one of whose processors has access to the device. Therefore, a paging task must preform all paging operations for the paging group. A paging group could consist of only one processor, being the paging task itself, or it may consist of one paging task and several other processors.

The creation of a paging group is topology dependent and therefore known until load time. The actual determination of paging groups is done dynamically once the task has been broken up as explained in Section 5.3.

To determine which processors constitute a group, first the paging tasks must be established. Each processor will consult its SMSMLOC byte. A one in this location will indicate that the processor has access to a backup device, and therefore will constitute itself as a paging task. Figure 6-1 depicts the situation where processors 1,3 and 4 will constitute the paging tasks since they have access to a backup device.

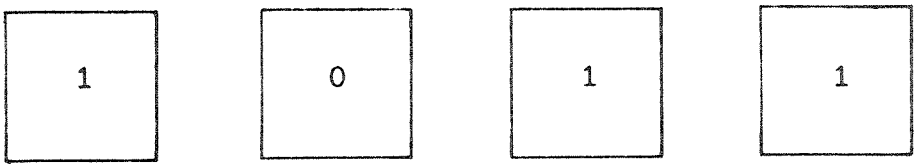

Figure 6-1: SMSMLOC of a 4 processor task

Processors with no access to a backup device (processor 2 in Figure 6-1) must constitute a group with a paging task. Each paging task will scan the PTASKGR table to find if a group must be established with other processors. The PTASKGR table will consist of two entries for each processor in the task:

- <proc #><group>

where <group> will indicate the group to which <proc #> belongs. Figure 6-2 shows the case in which processor 1 and 3 each form a group of only one paging task each, while processors 2 and 4 constitute another

group. From the topology shown in Figure 6-3, groups 3 and 4 will share a backup device while group 1 will have exclusive access to its backup device.

PROCESSOR NO.	PAGING TASK
1	1
2	4
3	3
4	4

Figure 6-2: PTASKGR for a 4 processor task

Once the groups are formed each paging task is responsible for the page transfer for all processors of the group.

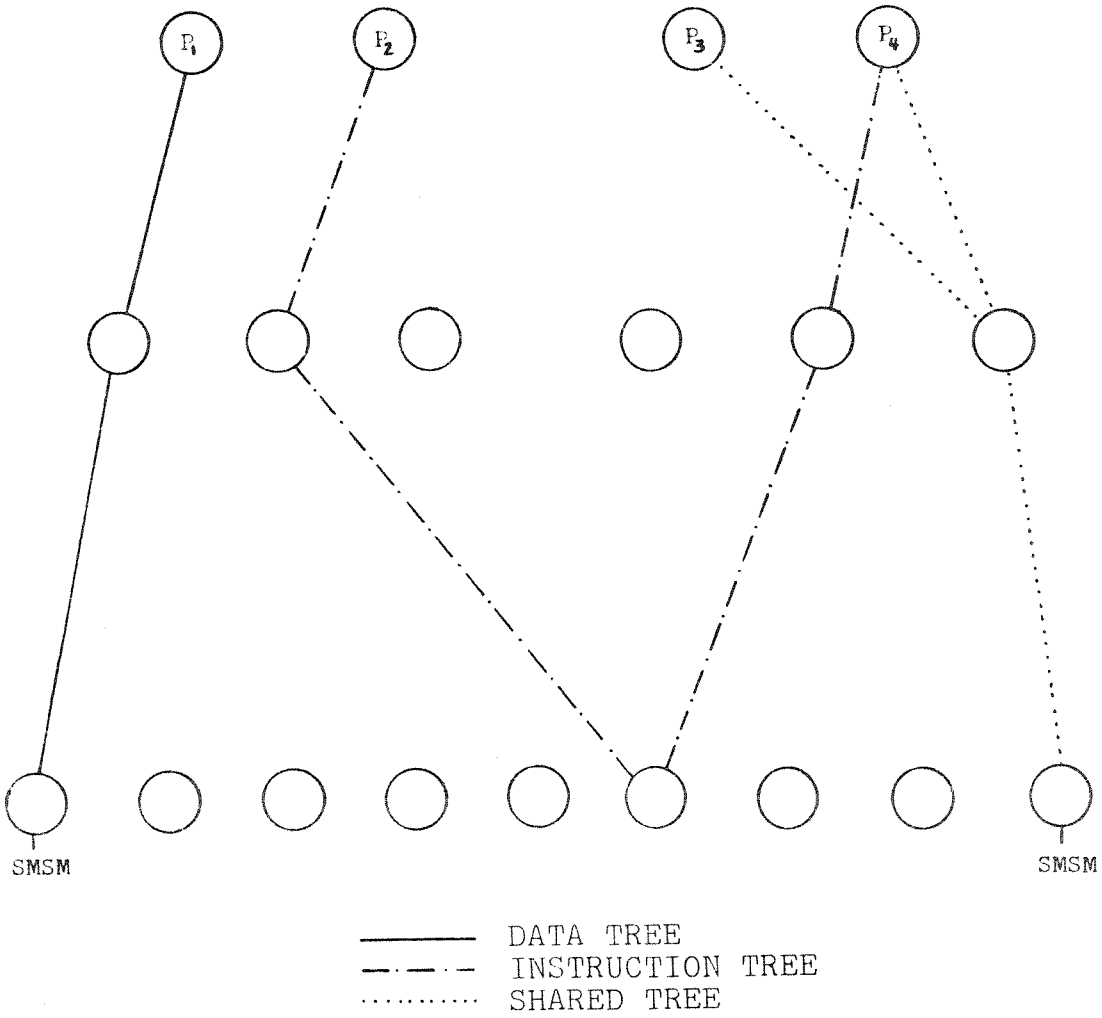

Figure 6-3: Paging tasks

The three basic steps involved in the paging procedure are:

- break up task
- transfer pages
- restore task

The task break up and restore have already been discussed in Chapter 5

For the page transfer mechanism to perform correctly, a page buffer must be accessible to the paging task. The location of this page buffer is not relevant to our discussion. The status information (<page in><page out><type of fault>) was made available to the paging task before the task was broken up into paging tasks. For each processor in the paging group the paging task will perform the following steps:

a. Page out:

- copy <page out> into page buffer.
- transfer <page out> from page buffer to backup device.

b. Page in:

- transfer <page in> from backup device to page buffer.
- copy <page in> to required memory module.

Upon completion of all page operations the original task must be restored as explained in Section 5.3.

6.3 Paging Group Topology

A paging task is responsible for the transfer of pages to and from the backup devices. The topology of the network which conforms the paging task plays an important role during paging. During a page-in operation (the same holds for a page-out operation but in reverse order), a page is brought from the backup device and placed in the page buffer of the paging task. Once the page is in the buffer it must be transferred to the appropriate memory module. If the page belongs to the paging task a local transfer takes place (i.e. a copy operation within the address space of the paging task). If the page belongs to another processor of the group a non-local transfer must be carried out (i.e. a copy operation involving an address space in a memory module outside the domain of the paging task). Since by definition of a paging group, only one processor of the group has access to the page buffer and

backup device, the page must either be copied first to a shared memory, or an instruction tree must have been created among the two processors involved. The actual procedure is architecture dependent and we will delay this discussion for later.

Having defined local and non-local transfers we will now see the three different topologies which might arise in the forming of the paging groups. The three different topologies are:

- Non shared backup devices.
- Shared backup devices with shared trees.
- Shared backup devices with no shared trees.

The different topologies arise due to the availability of backup devices. The ideal case is encountered when there are as many backup devices as there are processors in the task, and performance starts decreasing as sharing of backup devices and non-local transfers increase. Let's now discuss each of the different topologies.

6.3.1 Non-shared backup devices

This situation arises when there are as many paging groups as there are processors in the task and each paging group has exclusive use of a backup device. In this case a paging group contains only a paging task and therefore only local transfers are required.

Since there is no device sharing, all paging tasks can execute in parallel, hence throughput is maximized. This is the ideal case. Figure 6-4 shows this situation.

6.3.2 Shared backup devices/shared trees

The case of shared backup devices with shared trees is present when there are as many paging groups as there are processors in the task and two or more paging groups share a backup device. As in the previous case, only local transfers are required since each paging group is constituted by only one processor, namely the paging task.

Since the backup device is shared, its use must be coordinated by the different users by acquiring and releasing the device. Even though at this time there are no actual measurements as to what the impact of sharing

backup devices with shared trees has on paging, it is apparent that performance will be degraded as the number of paging-tasks which share the device increases. Figure 6-5 shows this situation.

6.3.3 Shared backup devices/no shared trees

This topology depicts the situation when there are more processors in the task than there are paging groups. In other words, at least one paging group is formed by two or more processors. By definition of a paging group, the backup devices are shared by the processors of the group and hence there will be a need for non-local transfer of pages. This topology is shown in Figure 6-6.

Again, there are no performance measurements at this time, but it is also apparent that performance will be degraded as group size increases.

6.4 One- and Two-dimensional Page Faults

So far we have discussed the paging problem and the influence different topologies may have on it. Paging tasks and paging groups are necessary because of availability of backup devices. When a page fault occurs on a two-dimensional memory, several pages must be

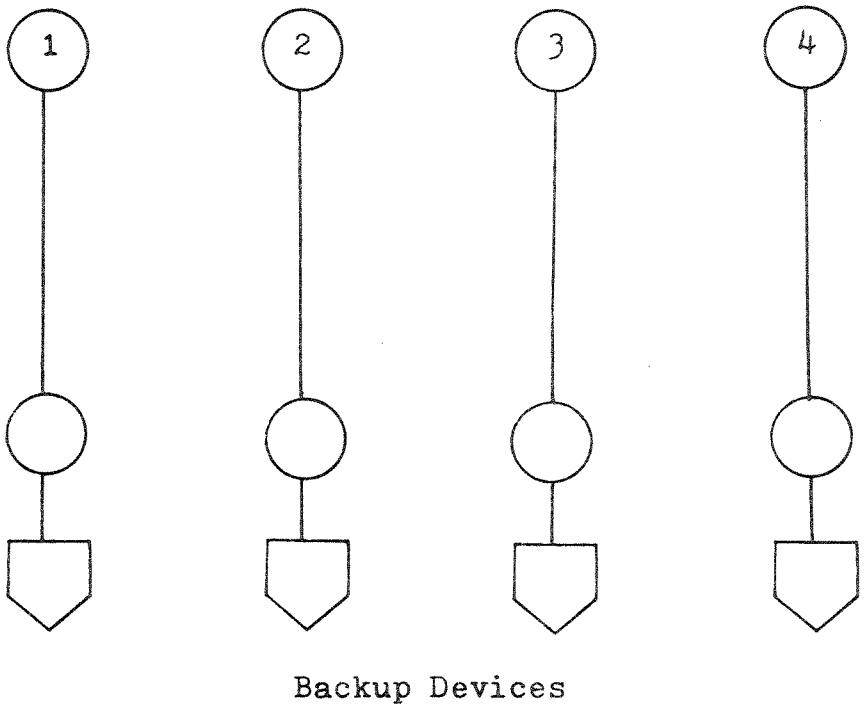

Figure 6-4: Non shared backup devices

replaced. The number of pages is equal to the number of processors of the task.

Page faults on a one-dimensional address space may be simpler to handle since only one page must be replaced. It is desirable, but not strictly necessary, that at load time one-dimensional spaces be clustered on memory modules which belong to a processor with access to a backup device. In this way only local transfers are required. If this is not possible, a paging group consisting only of

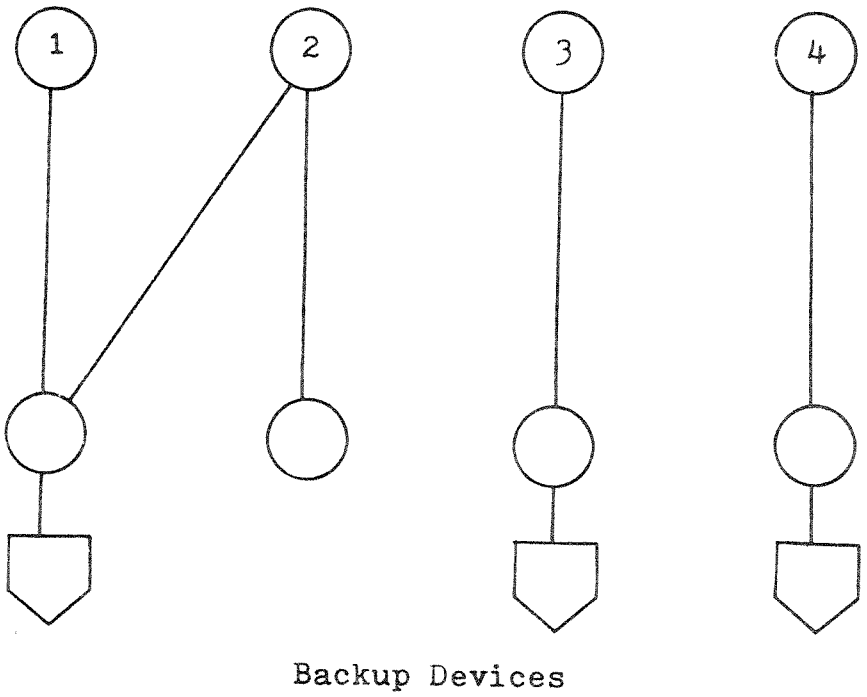

Figure 6-5: Shared backup devices/shared trees

the two processors involved in the transfer must be created and a non-local transfer must be accomplished. Again, only one page must be replaced.

6.5 Input output

We have delayed the discussion of I/O until now because it can be expressed as a special case of paging. The same problematic situations which appeared on paging are present in I/O, namely two-dimensional memories and availability of backup devices, or I/O devices in this case.

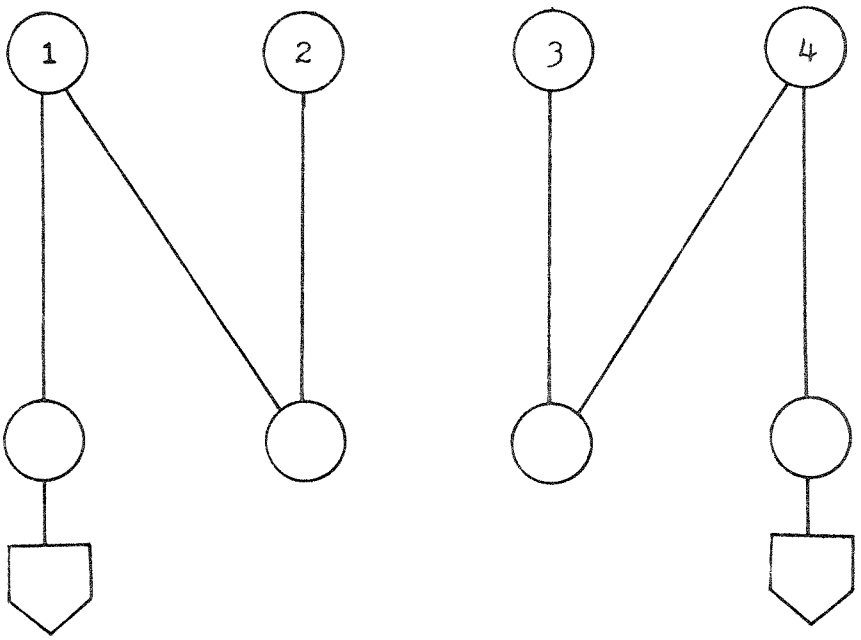

Figure 6-6: Shared backup devices/no shared trees

The I/O problem can be viewed as the collection of information from a two-dimensional memory and its mapping into a one-dimensional form through an I/O device. To achieve this result the same mechanism which is used for paging will suffice, with the restriction that now there is only one I/O device, and the information to be transferred is one byte (as opposed to one page frame) long. Only one paging task (and therefore one paging group) is required, the processor selected being the one with access to the required I/O device.

6.6 Conclusion

Paging in RNA's is complicated by the presence of two-dimensional memories and by the availability of backup devices. These two conditions require, first, that several pages be swapped in/out and, second, reconfiguration of the task is required since not all processors may have access to backup devices.

The distribution of backup devices is responsible for the different topologies that may arise. These topologies determine the creation of paging tasks and paging groups which must be handled dynamically by the Operating System.

Finally, the I/O problem can be viewed as a special case of the paging problem. Simply by having only one backup device (the I/O device) with data being one byte long as opposed to one page long.

Chapter 7

Conclusions

7.1 Summary

Several approaches have been proposed for the design of Operating Systems for variable structure multiprocessor systems [OUS 80, JON 79a, QUA 78, KAR 77]. These designs fall into one of the following categories:

1. One Operating System for each processor.
2. Operating System functions distributed among available processors.
3. Centralized nucleus with repeated distributed functions along computer modules.
4. Hierarchical-multiprocessor level structured hardware.

The span of control of these Operating Systems are specified by boundaries of physical processors. The Operating System described herein is decomposed by jobs as shown in Figure 1-1. Each job has a local Operating System. The total Operating System is a collection of job Operating Systems which control resources of a partition.

The span of control is the set of tasks which comprise the job. Each logical processor has a local Operating System which implements:

- those elements of the logical architecture not realized by the hardware
- communication between tasks
- communication between its tasks and the job level component of the job Operating System

The task level component of the Operating System is created by distributing functions across the PRM's of the physical processors which compose the logical processor. This structure which partitions by job and then partitions by task within jobs has the properties of localized communication and localized control. These properties yield an Operating System structure where the cost of system management is proportional to the number of jobs executing on the system rather than to the number of processors. Thus this structure seems practical even for systems with very large number of processors. The cost of virtualizing sets of physical processors into logical processors can be seen to be constant with respect to the number of physical processors composing a logical processor

We thus conclude that the strategies of the Operating System for network distributed multiprocessors described herein is practical not only for moderate number of processors but could be scaled to large number of processors. An implementation of such an Operating System in terms of the Texas Reconfigurable Array Computer is presented in the Appendices.

7.2 Future Research

Reconfigurable Network Architecture Systems present an entire new field for Operating System research.

The concept of two dimensional address space is beginning to emerge with the design of RNA's. The effects that this address space has on paging requires further work. Paging techniques and distribution of page frames throughout the backup devices also present a challenge for future research.

Intra-task communication and synchronization is another area where research is just in its beginnings. Dynamic task break-up and re-synchronization is a new concept and therefore further study should be pursuit in this direction. Simulation and modelling studies should

give a better understanding on this subject as well as a means of evaluating new mechanisms.

Thus, almost all aspects of software design for reconfigurable computers are fertile areas for further research.

Appendix A.

Synchronization in TRAC

A.1 Introduction

Paging, intra-task and inter-task communication require that the TH-PRM carry out specific functions. To do this the lockstep mode of execution must be abandoned. When the task is ready to resume execution the processors must be synchronized to proceed in lockstep.

This Appendix will discuss three mechanisms for synchronization. The first will use the capability of the Control Port to dynamically create and delete instruction trees. The other two will use the Carry Look-Ahead (CLA) logic as a means of sending synchronization signals.

A.2 Control Port Algorithm

This mechanism is based on the ability of the Control Port to create and delete instruction trees and to send signals to the processors involved.

To delete an instruction tree, the TH-PRM will send a message to the JM requesting the deletion of a particular instruction tree. The instruction tree is identified by the processors which make up the task connected by that tree.

In a similar manner the TH-PRM will request the creation of an instruction tree.

The following instructions are required:

- DINST(<param>): this instruction is actually converted to an IGM to the JM requesting the deletion of an instruction tree among the processors bit-wise selected by <param>. Once the tree has been deleted a TSP signal is sent to the processors involved.
- CINST(<param>): this will be the DINST counter instruction. It will request the creation of an instruction tree among processors bit-wise selected in <param>. A TSP signal is sent to the processors involved except when <param> has all processors of the task set. In this case an SSP signal is sent to the processors.

To complement these instructions two more synchronization primitives are required. The first is for processors waiting on a TSP signal and the second for processors waiting on an SSP signal. Actually both instructions have the same function but they must be distinct in the signal which awakes the processor because

an SSP signal will vector execution to a predefined location. The two instructions are:

- WAIT(TSP): this instruction will cause a processor to enter a microcode wait loop until it receives a TSP signal. Execution will resume at the next executable instruction after the wait.
- WAIT(SSP): this instruction will cause a processor to enter a microcode wait loop until an SSP signal is received. Execution will vector to a predefined location.

With these four instructions deletion and creation of instruction trees can be accomplished when desired. Therefore, synchronization of processors can be achieved. An example of the use of this mechanism is discussed in Appendix B.

A.3 CLA Logic Algorithm

The CLA logic over a shared tree can be used to implement a synchronization mechanism to put the processors at the leaf nodes in lockstep. This scheme would be useful for any situation in which a single processor controls the starting of the processors in the ensemble. A shared tree is set up over a memory module and the module is not made shared (i.e the shared flip flop in the module status register is reset). If the shared flip

flop is reset in a memory module, the module always feeds back the end around carry during phases 3 and 4 at the bottom level switch. If a shared tree is set up with this memory module as the root, a wire-ored ring of processors is formed as shown in Figure A-1. Even if a single processor sets its Generate to be true, each processor on the ring will receive a true Carry. The Generate signal will be called the Acquire and the Carry signal will be called the Grant [DES 83].

To break lockstep execution the processors will hold the Acquire signal to false. The processors may now execute on their own. Any processor which is ready to be synchronized will enter a microcode loop waiting for the Grant signal to be true. When all processors are ready for synchronization a processor, usually the TH-PRM processor will set its Acquire to true. As a result all the processors see a true Grant simultaneously and start in lockstep.

The following instructions are required:

- WAIT(GRANT): will cause the processors to enter a microcode loop waiting for the Grant signal to be true.
- SET(ACQUIRE): will set the Acquire signal to true.

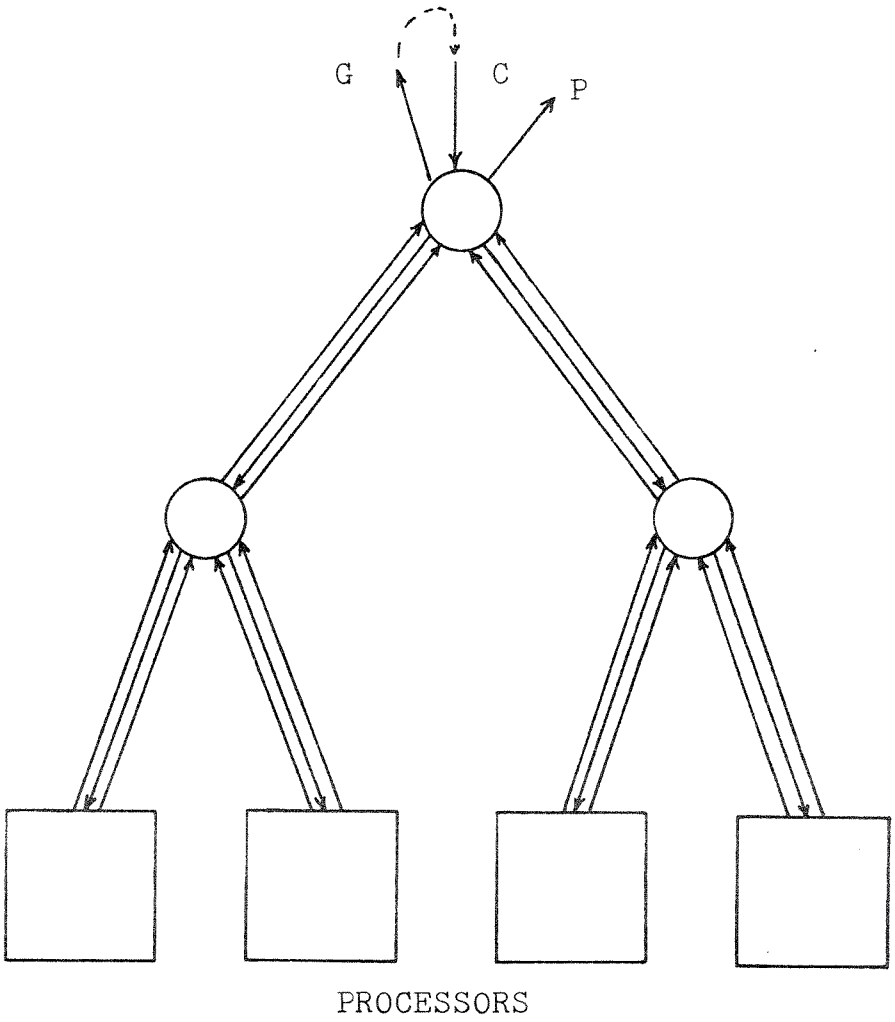

Figure A-1: CLA logic over a shared tree

- RESET(ACQUIRE): will set the Acquire signal to false.

A second synchronization mechanism is possible when there is a shared memory with its shared flip flop set. In such case the CLA link is open at the top and the memory behaves like a true shared memory as shown in Figure A-2. The number of processors to be synchronized is maintained in the synchronization register in the memory and forms a shared variable.

After finishing asynchronous processing each processor acquires the shared memory module and decrements the count by 1. If the count is not zero the processor simply creates its part of the instruction tree and sets an arithmetic operation so that it asserts a true propagate but does not produce a generate. The processor then waits for the incoming carry to become true. The last processor to acquire the shared memory module sees a zero in the synchronization register. It will create the last branch of the common instruction tree and executes an arithmetic operation to assert its generate signal. It will then vector to a predetermined location where every other processor is expected to branch when it sees a true Carry. Thus all processors start in lockstep.

The following instructions are required:

- SET(PROPAGATE): this instruction will generate a true propagate signal.
- WAIT(CARRY): this instruction will cause the processors to enter a microcode loop until a true Carry signal arrives.
- SET(GENERATE): this instruction will generate a true Generate signal.
- SETSEM(<param>): this instruction will set the synchronization to the value specified by <param>.
- GETSEM(<param>): this instruction will store the value of the synchronization register in <param>.
- DELBRNCH: deletes local branch of instruction tree
- CREBRNCH(<color>): creates a local branch of an instruction tree of color <color>.

A.4 Conclusions

Three synchronization mechanism were discussed. One makes use of the ability to dynamically create and delete instruction trees. The other two use the CLA logic as a synchronization mechanism One when a single processor is in control while the other the processors were independent.

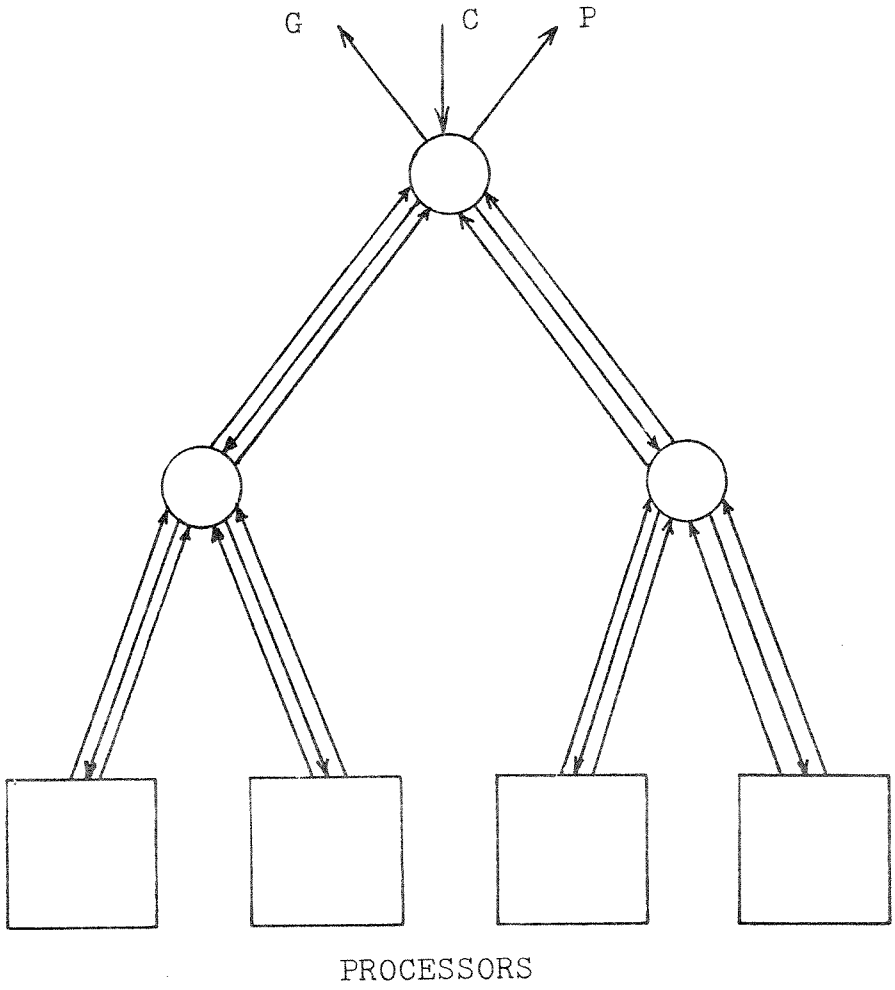

Figure A-2: CLA logic in a true Shared Memory