MULTIMICROPROCESSOR RESOURCE ALLOCATION

=======================================

[illegible]

DAVID WYNNE

[illegible]

Department of Information Science

University of Tasmania

1983

to be conferred 1985

PAGE INDEX.

Acknowledgements

I wish to acknowledge the assistance to Professor A. Sale, University of Tasmania, for his help at the start of my research, Professor J. Keedy and Dr J. Rosenberg, Monash University, for their comments and suggestions on my work, Dr E. Gehringer, visitor to Monash University from Carnegie-Mellon University, for suggesting the research topic, the staff at the University Computing Centre, and finally Dr C. Keen, University of Tasmania, for his invaluable supervision of my research and his assistance in producing the final thesis.

ABSTRACT

This thesis investigates the problems of allocating the data and code address spaces of a concurrent program onto the stores of a given multiprocessor computer architecture, and the allocation of the processes of the program to the processors of the architecture.

The minimum required of this resource allocation is to produce a legal mapping of the resources onto the multiprocessor computer. It will also attempt to give the most efficient mapping, and allow the user to guide this activity. This thesis describes the methods developed to implement this, which includes the specification of the structures of both the program and the computer architecture in a machine understandable form, and the design of algorithms to perform the allocation.

With the resulting techniques the emphasis is upon small scale multiprocessor computer architectures running dedicated concurrent programs. The resource allocation scheme results in a fixed allocation of the parts of a single program to a possibily nonstandard and specially tailored multiprocessor architecture. This would find little application with large regular mainframe multiprocessor computers executing time shared operating system programs, where the allocation of resources is highly dynamic and unknown at compile time.

CHAPTER (1)

(1.1) INTRODUCTION

This thesis investigates the problems of allocating the data and code address spaces of a concurrent program onto the stores of a given multiprocessor computer architecture, and the allocation of the processes of the program to the processors of the architecture. For the remainder of the thesis this activity is called resource allocation.

(1.2) RESOURCE ALLOCATION APPLICATIONS

Such a resource allocator will be useful in many applications. At present there are numerous inexpensive microprocessor chips available, some of which are described in [1,7,71,86,95,97], and it is economically feasible to construct from them multimicroprocessor systems. Such systems would be useful for dedicated and special purpose applications. In the past these applications may have been either too expensive to implement, or else the only choice available would have been to use custom designed discrete hardware logic or a general purpose minicomputer. The possibility of using microprocessor systems is attractive in these areas since such systems will be easier to design than dedicated hardware logic and less expensive than a minicomputer. Using a multiple microprocessor machine also gives the considerable advantage of allowing many operations to be performed in parallel, thus offering the potential of much faster solutions. There is also the possibility of constructing fault tolerant computer systems. A recent overview of these applications appears in [20].

The multiprocessor computer systems being considered for the purposes of this research would be constructed from off the shelf microprocessor and memory chips, and be connected together by straight forward bus technology. Special purpose networks such as delta networks [70] and dynamically reconfigurable or partitionable networks [80,84] are not explicitly included. Such systems have special purpose hardware that is difficult or impossible to construct in the above way.

Having decided to use such a computer system, the user is now confronted with the problem of getting software to run on these architectures. Some of these difficulties are described in [35,37,94]. Generally the use of high level languages allow the coding to be done relatively easily, and there are a number of concurrent programming languages becoming available that may be used [32,36,56,76,90,92]. However, given a concurrent application program written to use several processes, there is also now the problem of deciding where on the computer system the processes and address spaces of the program are to go. The requirements are to produce both a legal mapping and an efficient mapping. They can be achieved by introducing as little overhead as possible in the way of memory conflicts, avoiding the overcrowding of processors and by the reduction of excessive scheduling overheads. Only with these will the maximum work be obtained from the multiprocessor system.

One approach to achieve this is the static allocation of the program to the architecture. Thus memory contention can be avoided where possible by allocating the logical address spaces into physically separate memory modules. Processes can be distributed uniformly across the available processors, and the scheduling requirements will generally be confined to the processes on each single processor. Such an arrangement is particularly suitable in small computer systems where there may be only a minimal operating system resident to support the program. Alternatively, even if the architecture and operating system form a moderately sized system, the minimization of memory contention and process scheduling overheads can still be important, as it is in StarOS system [26].

To aid the discussion in the main part of this thesis some terms are now defined or explained explicitly in the following.

(1.3) CONCURRENT PROGRAMS

A concurrent program consists of a number of sequential processes that have the capability of being executed simultaneously. As these processes execute they will access their code and they will also access their variables and perhaps procedure invocation stacks and dynamic heaps. This data information is collectively known as the logical address space of the program. Processes and address spaces together are known as the elements of the program, thus a concurrent program consists of process elements and address space elements.

In such a concurrent program the processes will not have equal access to all of the available address space elements. Instead this access pattern will be highly irregular, with some address space elements being accessed much more than others, and some processes will perform many more such accesses than other processes. This is referred to as the access pattern of the program, and information about this is conveyed by the number of cycles performed between each process and address space.

(1.4) COMPUTER ARCHITECTURE

The program will execute upon some computer architecture. The architectures considered in this research are all multiprocessor architectures having more than one hardware processor. A processor provides the physical capability of executing one process at a time, while the address space elements of the program reside upon the physical memory stores of the system. The processors and stores need not be all be identical; both homogeneous and heterogeneous architectures are allowed. In a heterogeneous architecture the processors may be of different kinds or the stores provided may be of different sizes and access speeds. Collectively the processors and stores of the computer architecture are known as its resource elements, and thus an architecture consists of processor elements and store elements (or physical memory elements).

As does the program, the computer architecture has an interconnection structure. Processors are connected to the physical stores in a manner that may or may not be uniform. This interconnection structure is represented by access paths between processors and stores, by the cycle speeds of the stores themselves and the access times of the interconnection hardware. Processors can only communicate to other processors via the use of these common stores. Many kinds of interconnection structures are possible, as are discussed in [4,22,79].

The interconnection structure can result in memory contention. This results from two processors simultaneously attempting to access the same store or to use the same connection hardware. Such interference is discussed in [8,9,44] and it is very important in determining how efficiently the resources of the computer architecture are used in supporting the program application. This efficiency is measured by the throughput of the program executing upon the architecture. In this context the throughput is the number of times the program can execute a given program workload. Thus if a process is specified to make 445 references to a particular address space in some time period, and if in the implementation it accesses this address space at the rate of 44 references per second, then the throughput measure is 0.1. An allocation mapping that is twice as efficient as this will have a throughput of 0.2.

(1.5) RESOURCE ALLOCATION

Finally there is the action of bringing the program and the architecture together. This resource allocation applies to all of the program elements, which must be assigned to some subset of the resource elements of the computer. The assignment or allocation of an individual program element to a resource entails-

Specifying upon which hardware processor a process is to execute, and

Specifying upon which physical memory an address space is to be placed.

This specification or resource mapping must satisfy legality conditions, and preferably it is also to be efficient. The legality constraints under which the resource allocator must work are

Each process must be assigned somewhere, and must be assigned so that it is executed by only one processor.

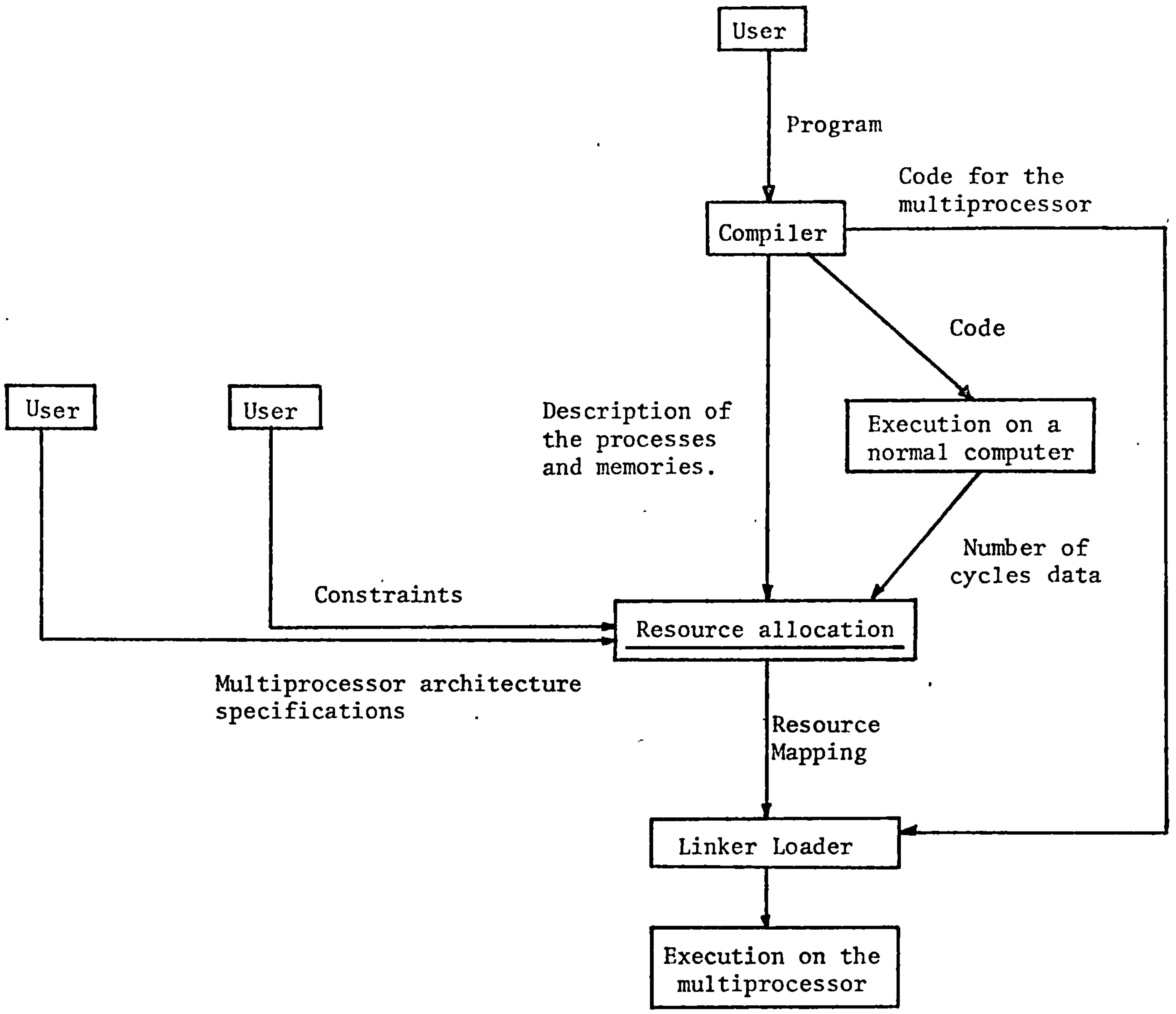

User
Program
Code for the multiprocessor
Compiler
Code
User
User
Description of the processes and memories.
Execution on a normal computer
Number of cycles data
Constraints
Resource allocation
Multiprocessor architecture specifications
Resource Mapping
Linker Loader
Execution on the multiprocessor

Figure 1.1

Each address space must be assigned to some single physical memory space. There can be no overlap with any other simultaneously present address space.

Each process must execute from a processor which can access all of the stores to which the address spaces accessed by the process have been allocated.

This concludes the definition of the basic terminology. The concept of resource allocation and is application areas have been introduced. Given such a utility and starting with a suitable application there are a number of stages involved in using it in order to implement a problem onto a multiple microprocessor. Figure(1.1) represents this information flow digrammatically.

(1.6) HIGH LEVEL LANGUAGES

Firstly the problem needs to be implemented as a concurrent program. The advantages of using a high level language for any programming is well documented [15,18,35,93,94]. In view of this, and the fact that the resource allocator would itself be a complex program utility designed to aid program production, it is reasonable to assume that the development of the user program will always utilize a high level language. Thus the resource allocator will always be preceded by a compiler.

A suitable high level language will contain all of the standard features associated with such languages, as is found in languages like Pascal, Algol, Fortran, Cobol and the like. Furthermore, since the target architecture is a multiprocessor architecture, the language must have the capability for specifying concurrent processes and for controlling their execution. Some examples of this kind of language are Pascal Plus [10,92], Concurrent Pascal [34,36,39,76,82], Path Pascal [31,32,63], Concurrent Euclid [56], Modula 2 [29,90] and ADA [2,96]. There is no restriction implied upon the number of different compilers or languages that may be used, provided some means is available to link together at some stage the codes and data spaces produced by the different compilers.

All of the languages in this last list allow the specification of concurrent processes. The languages also provide some mechanism for communicating between two processes and for the sharing of data. Most of these languages that include processes will also have modules. The definition of a module is different for each author and language, with some examples being presented in [16,41,49,64,67,68,69]. However in most cases the compiler can implement the module as a collection of procedures and variable spaces. So generally the use of modules has no effect upon the application of a resource allocator, which deals with variables and procedures and the access paths between these. However if the computer architecture supports modules directly, as in the Monads architecture [50,51] or the StarOS system [25,26], this poses no essential problems. In this case the resource allocator would deal with modules that have access paths between modules, as well as variables and processes. Nevertheless, to simplify the research, modules are not considered further.

(1.7) ARCHITECTURE SPECIFICATION

When used the allocator requires the specification of the structure of both the program and the architecture. The program structure is best described by the compiler in terms of its process and address space elements and the access paths between these. The number of cycles information for the throughput calculations will be obtained by running the program on a normal uniprocessor computer. The code would be argumented with statements to gather statistics about the number of accesses made. This step is important as without the number of cycles information there is no feasible method for the resource allocator to obtain relative efficiencies of differing resource allocations.

The user is required to give a description of the computer architecture to the allocator. The information that needs to be conveyed concerns such things as-

> The kinds of processors, including their cycle speeds and microprocessor type.

> The sizes and access times of the physical memories.

The locations of memory mapped I/O, and the port addresses of nonmemory mapped I/O, as well as the processors which have access to these.

The addresses of interrupts, and the processors to which these interrupts occur.

The interconnection pattern between the processors and their stores. This will cover hardware buses and also the locations in the addressing range of a processor of its attached memories.

Only this level of information is required. Greater detail about the hardware, as is given in many computer hardware design languages (a survey of these is given in [59,87]) is not required by the allocator and so is not supplied in this specification.

(1.8) PROGRAM SPECIFICATION

The specifications of the computer architecture need only be produced once per architecture, and used for the allocation of all programs to this architecture. Extra information is however required for each program. The user can interact with the resource allocator to guide it in its allocation strategy. The initial starting point for this is the description in [26, section 11] of the StarOS resource directives. These constraints may be to ensure that some conditions external to the allocator are achieved, or to guide the allocator in its global strategy to achieve the most efficient mapping. The interaction takes place by the means of constraints placed upon the allocation. These constraints may be to

Ensure that processes execute upon processors that have hardware access to the appropriate I/O ports,

Ensure that variables of a program which are used to access memory mapped I/O ports are placed at the correct address in the appropriate physical memory module.

Allocate selected processes and address spaces onto the same processor or store, or upon separate processors or stores. This ability is useful when using a multiprocessor to provide greater degree of computing reliability, one example of such a multiprocessor design being described in [6]. If different parts of a program are allocated upon separate physical resources, then a failure of one resource will only bring down one part of the program.

Allocate processes with special requirements to processors that possess special execution capabilities, such as a floating point accelerator.

Finally the resource allocator will operate upon this information and produce a resource mapping, or perhaps indicate that no mapping is possible. If the allocator succeeds then it will generate an allocation mapping. This would be used for a subsequent linker stage to actually load the program onto the machine.

(1.9) THE TOPICS RESEARCHED

The research area and its application have been defined. The aim of this thesis is to investigate this problem, concentrating on the following topics

A) The computer specification language.

The design of the input computer architecture specification language to support the specification of the computer and to allow the user interactions is outlined. These specifications need to deal with a wide variety of architectures, since the actual hardware may be connected in many ways. However at the same time it is recognized that most architectures will be regular and involve repetitive constructions. Thus the specification language allows for the natural expression of such structures. They also allow for the easy extraction of information from the specification for use by the user in writing the user constraints.

B) The throughput of the allocation mapping.

For its allocation activity the allocator will need to derive the throughput of an allocation, to decide if the allocation is efficient or not. Thus a general purpose throughput calculation algorithm is derived, which takes into account the effects of memory contention. Two different versions of this are implemented and examined. The original starting point for this work is from [44] which describes a general throughput calculation model that takes into account memory interference produced by a number of independent nonconcurrent programs executing on a multiprocessor. The thesis work extends this to include the effect of differing store cycle speeds, the effect of bus contention and bus cycle speeds, and to provide the throughput for a single concurrent program.

C) The allocation algorithms.

Finally the allocation algorithms themselves have been designed and an implementation produced to demonstrate them. This research borrowed ideas from search techniques developed in other areas, such as parallel searches in game trees [62]. It builds on the need for resource usage directives as described by [26] for the StarOS project.

A list of the original research performed follows..

A) The design of the computer specification language is the authors own.

B) The original memory interference model is taken from [44]. The authors own original research is to modify this to suit the requirements of a resource allocator.

C) The starting point for the resource allocator research is [26]. The design of the constraint specifications and the design and implementation of the allocation algorithms are all original research by the author.

(1.10) CHAPTER SURVEY

The remainder of the thesis is concerned with an expanded description of this work. Chapter 2 introduces the resource allocation activity in more detail and describes some of the problems encountered in performing this.

Chapter 3 is concerned with the design of the specification language. This language is based upon a graph structure description of the computer architecture and allows the specification of the multiprocessor at the level of its processors, stores and bus interconnections. Chapter 4 discusses how this language is used to describe to the allocator the various kinds of computer architectures that are likely to be encountered.

Chapter 5 then describes how the computer program that is to be mapped onto the architecture is specified to the resource allocator. The extra information required of the user to guide the allocator is also introduced. No implementation of the specification language was attempted. While the ideas presented are important for the use of a resource allocator, there are essentially no new difficulties in implementing such a language once it has been designed.

Chapter 6 describes how, given a particular resource allocation mapping, its throughput may be calculated. Two alternative ways of computing this is presented, one by a simulation model and one by a probabilistic model. Programs implementing both were developed to demonstrate their validity.

Chapter 7 is concerned with the search method used to find solutions. This is basically a tree search with a heuristically ordered search pattern designed to increase the probability of obtaining satisfactory solutions.

Finally chapter 8 presents the conclusions.

CHAPTER (2)

(2.1) AN OVERVIEW OF RESOURCE ALLOCATION

Simple applications of the resource allocation problem addressed by this thesis are described in the following.

The simplest example of resource allocation is the implementation of a program to execute on a uniprocessor system possessing a uniform memory structure. Even for concurrent programs this is readily achieved. The processes of the program execute on the same processor and can be managed by an appropriately written scheduler. Memory allocation schemes for a linear memory are well understood.

The addition of more processors, thus creating a multiprocessor computer architecture addressing a common memory, can also be handled relatively easily. One method is to construct a scheduler which allocates time slices on different hardware processors to the processes of the program as they become ready to execute. In this approach, the rest of the computer programming system need not even be aware of the change to a multiple processor architecture. Unfortunately, as the number of processors attached to a common physical memory increases, the amount of memory contention also increases. Eventually there comes a point of diminishing returns where the addition of an extra processor to the hardware will add only a marginal improvement to the throughput.

Many techniques may be used to alleviate this problem. Interleaved memories, separate memory modules, cache memories or memories that are faster than the processors are some possibilities. Many of these memory designs are more applicable to large computers because of the cost of the associated hardware required to implement them. As well these solutions have the common characteristic of ignoring the specific structure of the programs being executed.

For illustration of this last point, consider a program consisting of two processes that access separate variables. The logical address spaces for these variables can be placed in a common physical memory module and the two processes can execute on separate processors. In this

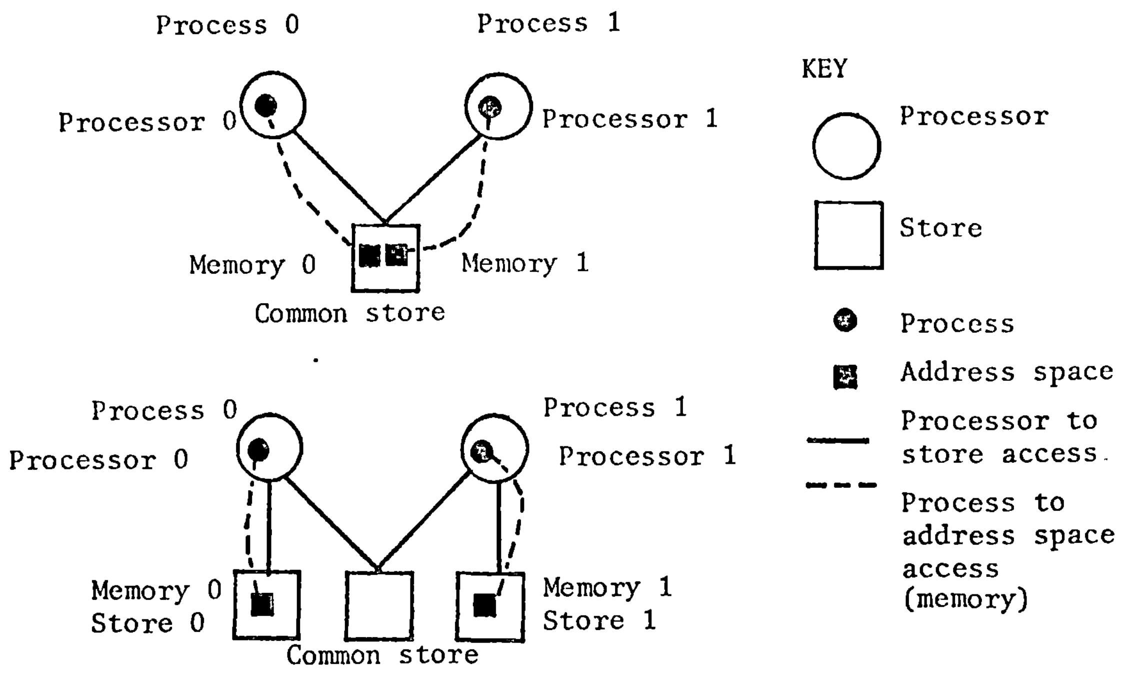


Figure 2.1

case there will be memory access conflicts when the two processors attempt to access the same store simultaneously in order to refer to their own memories. This situation is seen in the figure(2.1,top).

The memory interference may be reduced by the hardware techniques discussed above. Alternatively, if the structure of the program can be taken into account, on a suitable computer architecture the variable spaces could be placed into separate memory. blocks, as in figure(2.1,bottom). Now the interference due to accessing these memories will be nonexistent.

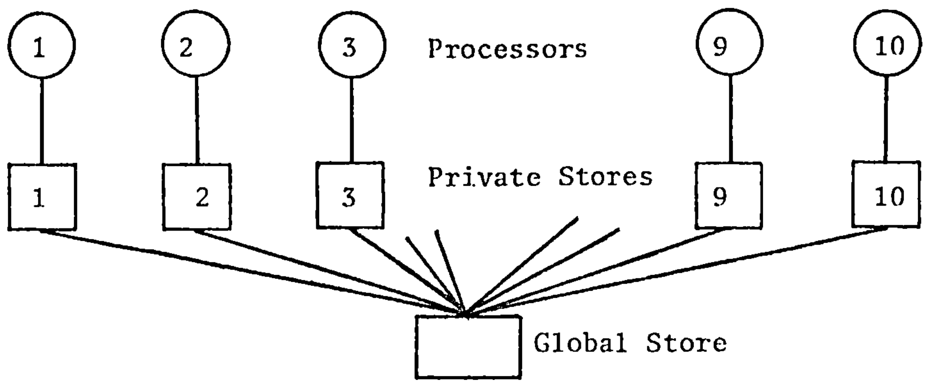
This example illustrates how a knowledge of the program may be used to optimize its execution without the use of sophisticated hardware techniques. The information utilized here was that the accesses of the processes of the program were to independent address spaces and this allows the derivation of the more efficient allocation solution. However for a large computer system such information about address access patterns is awkard to obtain since there will be many different programs executing, and these will be changed often. To attempt the optimal allocation of every program based upon its individual address accessing patterns will be impractical. The research emphasis on medium sized statically allocated programs is a consequence of this.

(2.1.1) EXAMPLES OF RESOURCE ALLOCATION

As an example of resource allocation a simple instrument monitoring computer system is used. This system is to monitor a number of instruments, and record their values in such a way that they can be retrieved upon command and displayed on a terminal. One way to structure a program to perform this action is to have an individual process obtain the results from each instrument and put these into a common table. Another process would be used to maintain the terminal display based upon the information in the table and according to user entered commands.

If it is assumed that the program work required to monitor a single instrument requires a significant part of the execution time of one individual processor, then a possible hardware implementation will have one processor for each of the instrument monitoring processes, and one more for the command process. This will give the best execution time performance for the complete program. Each processor can be supplied with its own private memory and also some global memory in common with all the other processors. For such an architecture as much as possible of the local address space of each process of the program would be assigned to the local physical memory of the processor. This will reduce the memory contention to the obligatory minimum, reducing it down to conflicting accesses by the processes to the address space that is shared with other processes. This hypothetical structure is depicted in the figure(2.2).

> NOTE. In this figure, and in others, a computer architecture is depicted by using circles to represent processors and squares (or rectangles) to represent physical store modules. An access path between a processor and a store is represented by a line drawn between the circle representing the processor and the square representing the store. Thus figure(2.3,left) represents a computer architecture of two processors and three stores. PROCESSOR_1 accesses STORE_1 and PROCESSOR_2 accesses STORE_2.



Instrument Monitoring Multiprocessor

Figure 2.2

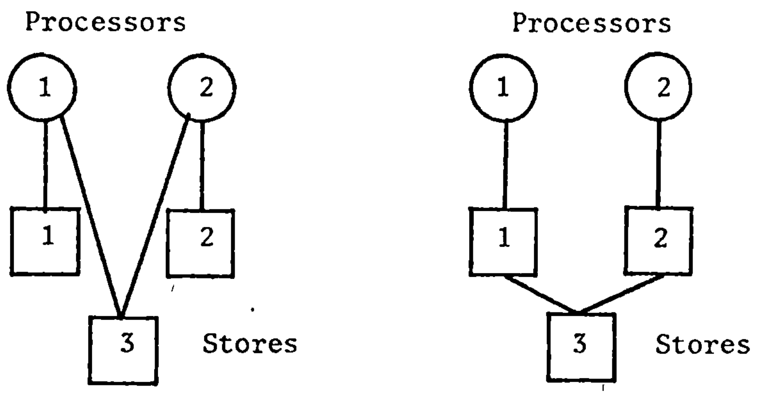


Figure 2.3

Both processors access STORE_3.

In order to avoid visual clutter in diagrams containing a large number of processors and stores, the following convention is adopted. If a line is drawn from a square representing a store to a second store square, then the first store is considered to be accessed by all of the processors that access the second store. Thus in figure(2.3,right), a line is drawn between STORE_1 and STORE_2. STORE_1 is accessed by PROCESSOR_1 and so the line between the two stores shows that PROCESSOR_1 can also access STORE_2. Thus the architecture of figure(2.3,right) is identical to that of figure(2.3,left).

In figure(2.2) a homogeneous architecture has been proposed. It could be possible to use different sized stores for each of the processors, and even to use different kinds of processors, thus creating a heterogeneous architecture. However it will generally be preferred to design and use homogeneous architectures, both because of an easier design stage, and also because such designs will more readily transfer to other projects.

To this structure the instrumentation input and output ports will be connected, with the ports for each individual instrument being connected to a separate processor.

Given nine instruments, a possible skeleton program for this application is

```
PROGRAM MONITOR ;
   COMMON DEFINITIONS ;
   COMMON VARIABLES ;

   PROCESS COMMAND ;
   PROCESS INSTRUMENT_1 ;
   PROCESS INSTRUMENT_2 ;
     . . .
   PROCESS INSTRUMENT_9 ;
END ;
```

Each process will have a number of private variables and procedures, and the instrument processes communicate to the command process via a common table and common table access procedures.

If this program were to be implemented upon a normal computer architecture then either the compiler or a subsequent linker would be able to allocate the program onto the computers memory store, using standard techniques. When using the architecture of figure(2.2), one process can be assigned to each of the processors. This has the advantage of incurring no scheduler overheads. As well, the private address space of each process can be allocated to the private stores of the corresponding processors. This gives the advantages of conflict free access to these address spaces. In these circumstances it is not

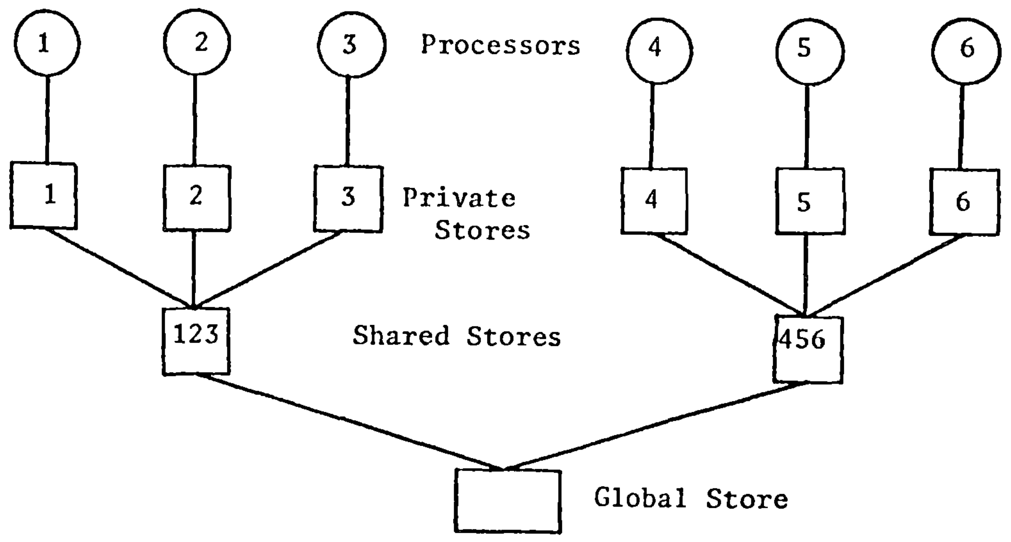


Figure 2.4

appropriate to use a general purpose scheduler which allocates a ready process to a free processor as one becomes available.

If all of the I/O ports are not available from every processor then the user will be required to indicate to which processors the instrument monitoring processes are to be assigned. This is to ensure that each process is capable of accessing its correct instrument I/O ports. If this specification is imposed, then the resource allocator would then allocate the remaining control process to the best processor for it, which in this case will be the only unused processor available. Otherwise, if there are no such specifications, the allocation program allocate the program so as to obtain the best throughput, which in this computer architecture will imply one process per processor. At the conclusion of this activity the resource allocator will insert linking information into the compiler generated code to allow the code of the processes to access correctly their memory address spaces.

For this example the process to processor allocation can be simple, particularly if the user specifies the process allocations. The memory allocation is also straight forward. The allocator needs to allocate the private variables and code blocks that are referred to the most into the private store of each processor, and allocating all common address spaces and the left over private address spaces (if any) into the common store. Thus memory contention, a product of the number of accesses by a process to a physical memory and the number of different processes accessing this memory, can be reduced to an unavoidable minimum.

The resource allocator problem may easily become more complicated with only a few changes to the target architecture. For example a computer system with only six processors, each of which has access to all the required input ports, may be available to implement this program. Furthermore the memory may be arranged with a equal sized private memory attached to. each processor. Then each group of three processors would share a common memory block, and all processors would share a common global memory block. Such a design is given in figure(2.4).

The intent of constructing a computer system with these different levels of shared memory is twofold.

1. To increase the total amount of physical memory without exceeding the memory addressing range of any individual processor.

2. To allow the possibility of greater memory sharing between processors and yet still reduce memory contention.

In demonstration of this last point, processors 1, 2 and 3 can communicate between each other via the shared store 123 without interfering with processors 4, 5 and 6 in their accessing of their own shared store 456.

In this situation all that the resource allocator needs to know from the programmer is the addresses of the input ports that are to be used by each individual process. These addresses would be inserted into the appropriate I/O routines of the process codes. This information could not now have any affect upon the allocation of processes to processors, since each processor now accesses all of the input ports. From this information the resource allocator will be able to go ahead and allocate the program.

(2.2) RESOURCE ALLOCATION ASPECTS

Now some of the factors that may affect the resource allocation placements will be considered.

(2.2.1) LOW LEVEL DETAILS

Firstly the resource allocation may be influenced by some machine level details, such as the programmer inserting simple assembler language routines to control input/output ports. Such information is not directly accessible to the resource allocator, but instead the user programmer will need to impose constraints upon the permissible mappings to guide the allocation activity in this area.

(2.2.2) PROCESS TO PROCESSOR ALLOCATION

In the instrument monitoring example, where the architecture of figure(2.4) is used, there are ten processors to be statically assigned to the six processors. The allocator will tend to allocate the longest running processes to separate processors, with the other less time consuming processes placed where ever they fit. The length of the run time of the processes is obtained by the execution of the program upon a normal computer and gathering statistics. However an allocation made in this way may not be optimal, depending on the combination of the particular program and computer architecture being used. So it will not always be the arrangement selected. This will be influenced by the effects of memory interference, different memory cycle times of each physical memory block and of each shared memory bus, the size of the logical address space accessed by each process and the size of the physical memory shared by each processor.

(2.2.3) DEGRADATION DUE TO MEMORY INTERFERENCE

The inappropriate allocation of processes may lead to serious execution time inefficiencies by the action of memory contention. In the example architecture of figure (2.4) this can be demonstrated by considering two

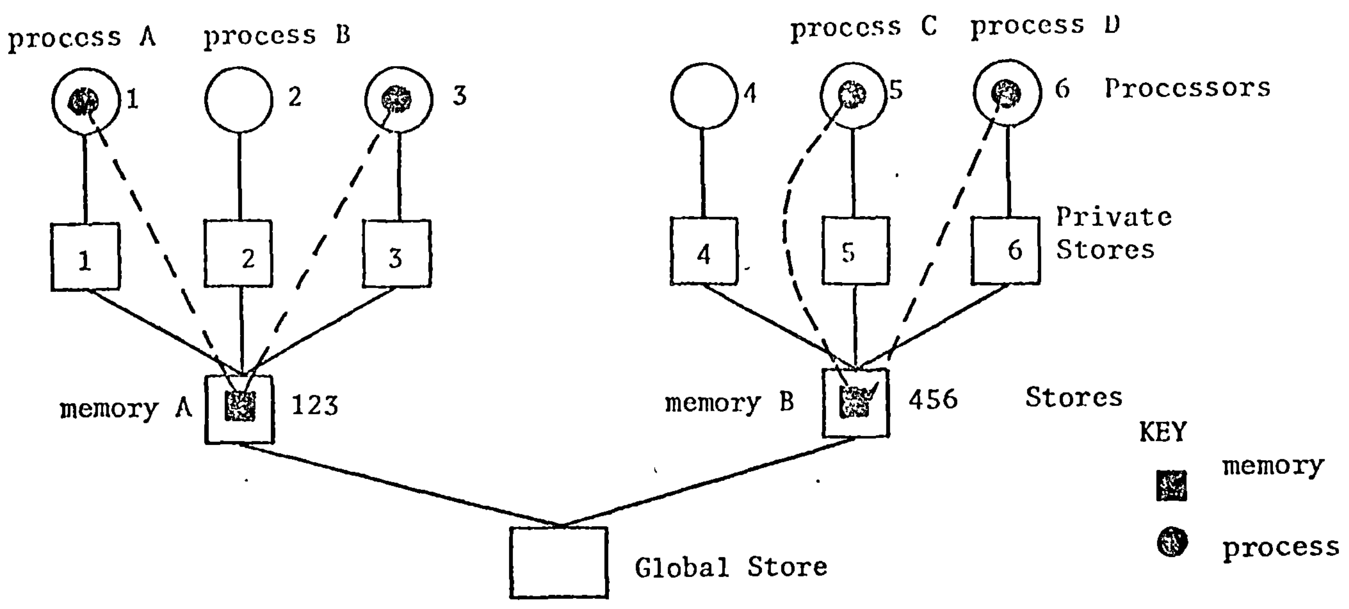


Figure 2.5

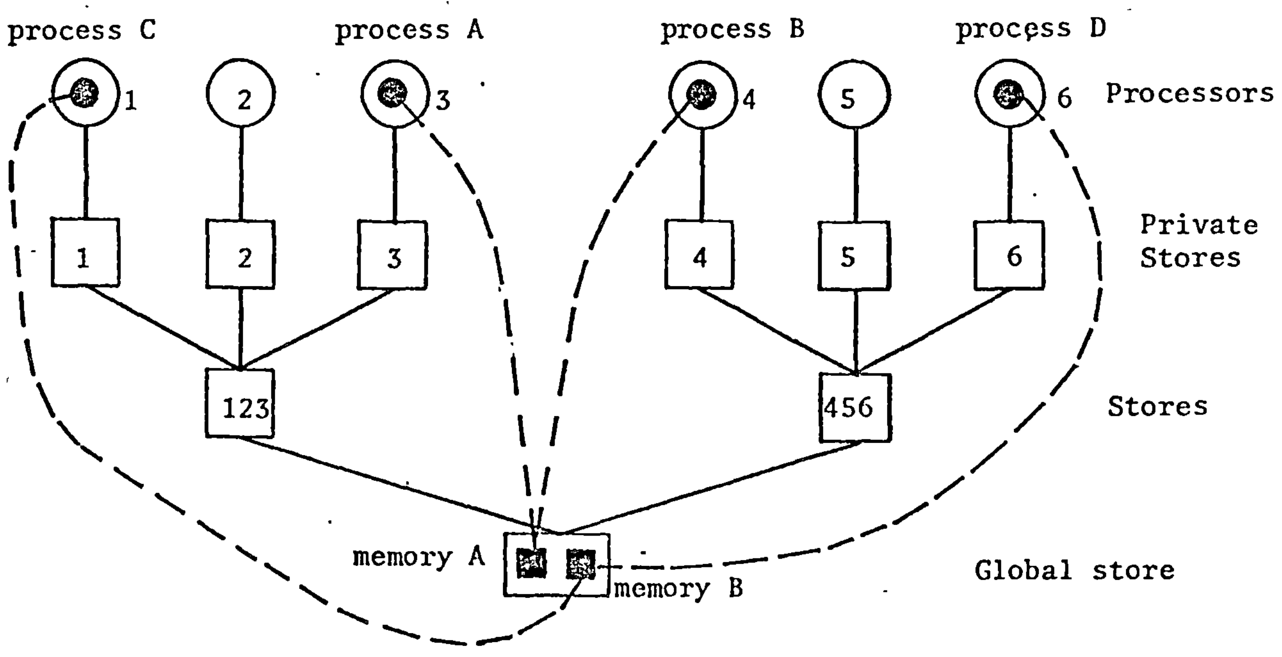


Figure 2.6

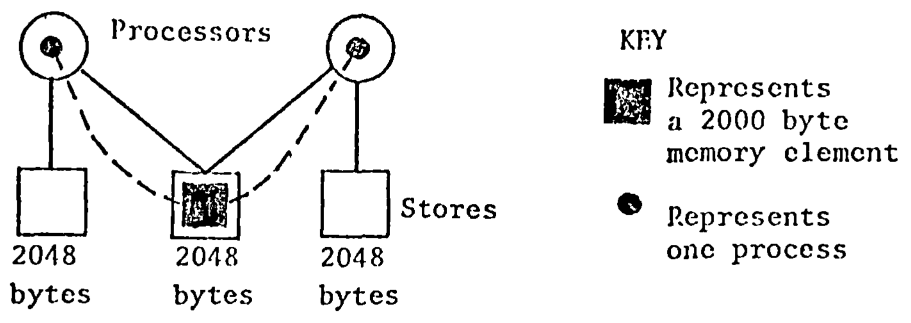


Figure 2.7

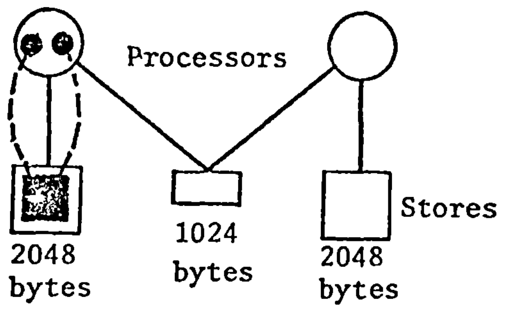


Figure 2.8

pairs of processes, each pair their own heavily used data section. The pair A and B could be assigned to any of the processors 1, 2 or 3, and their shared data space A placed upon the shared store of these processors. The other pair can be similarly assigned to the processors 4, 5 or 6. With such an allocation the pairs of processes can access their own shared address spaces without interference. This situation is represented in figure(2.5).

However, if each process had been assigned so that the first process of the pair is in the processor group 1 to 3, and the second process of the pair is in the other processor group, as in figure(2.6), then the common shared data will have to be assigned to the global memory store. This assignment will inevitably result in greater memory conflict.

Thus after the preference of processor execution speed and process execution times, the possibility of execution degradation arising from memory interference has to be considered.

(2.2.4) ALLOCATION INTERACTIONS

A final difficulty in the allocation process is the interactions that occur between individual allocations of program elements to resources. These interactions frequently prevent any straight forward allocation strategy, and will often prevent the most efficient usage of the computer architecture. As an example, for a two process program the best allocation onto a two processor architecture is to have a process assigned to each processor, shown in figure(2.7).

However the common address space element may not be allowed onto the common store. This will happen if the size of the common address space is larger than the size of the common store. Therefore the common address space now has to go into one of the private physical memories. In order to access this, both processes will then end up on the same processor, with the other processor idling. This is depicted in figure(2.8), where the common store has a reduced size of 1024 bytes.

A similar situation can occur easily with the allocation of address spaces to stores. The difficulties also increase when memory and bus contention is to be taken into account. These interactions may be caused by other factors, and can affect the allocation strategy of the whole program.

Because of these interactions the allocation problem is nonlinear, it is not possible to work out the allocation for individual parts of the given problem and then to combine these to give a complete allocation. In most cases it will unfortunately turn out that the allocations for one part will interact with the allocations in all of the other parts, so completely invalidating any such divide and conquer solution.

(2.2.5) RESOURCE ALLOCATION FAILURE

The resource allocator can fail to find a legal mapping for a particular problem if there exists the situation where

The total physical memory space of the computer exceeds the size of the program.

The physical memory addressing range of a processor exceeds the address space sizes of all the processes that are required to execute upon it.

A process is assigned to a processor so that it cannot access the stores to which its address spaces have been assigned.

A process or address space element is assigned to a resource where it cannot access its I/O ports (or memory mapped I/O ports).

(2.3) SOME RESOURCE ALLOCATION APPLICATIONS

The introductory examples given so far have given some of the basic requirements, and some of the problems confronting a resource allocator have been demonstrated. In the following more example applications are introduced.

(2.3.1) PICTURE PROCESSING

One feasible application of a multiprocessor architecture is in picture processing. Special purpose hardware designs exist for this [72,80]. However, for the purpose of this example, a design using standard microprocessors and memories is considered. For such an architecture the picture processing program could be structured as one or more main processes which deal with the overall control of the program. This would be the input and output of picture data, the initialization and the termination of the picture processing algorithms. Then there could be any number of small individual processes, each designed to operate independently upon one small area of the picture information. A decision to choose this structure can be made because it can be efficiently implemented as one or more main processors accessing a global store, and a series of smaller processors capable or performing picture type operations. A suitable computer architecture for this is

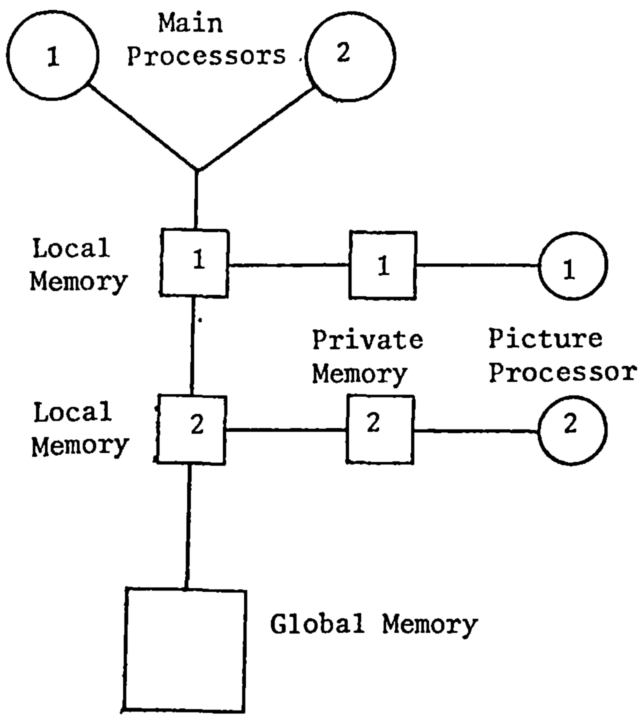


Figure 2.9

given in figure(2.9).

The user will need to provide constraints which will place the picture processes onto the picture processors, and supply the additional information that the code for the picture processors has to be compiled into a different instruction set from the code for the main processors. The user is also required to supply specifications of the computer architecture. Then using these user directives and the specifications, the resource allocator will be able to perform the rest of the allocation for a suitably constructed program automatically.

(2.3.2) CM* TYPE COMPUTER ARCHITECTURE

Another example where resource allocation is useful is when using a computer architecture similar to the Cm* computer system[26]. In such a computer there are a number of processors, each accessing its own local memory. In figure(2.10) the local memory of processor 1 is store 1, and so on for the other three processors.

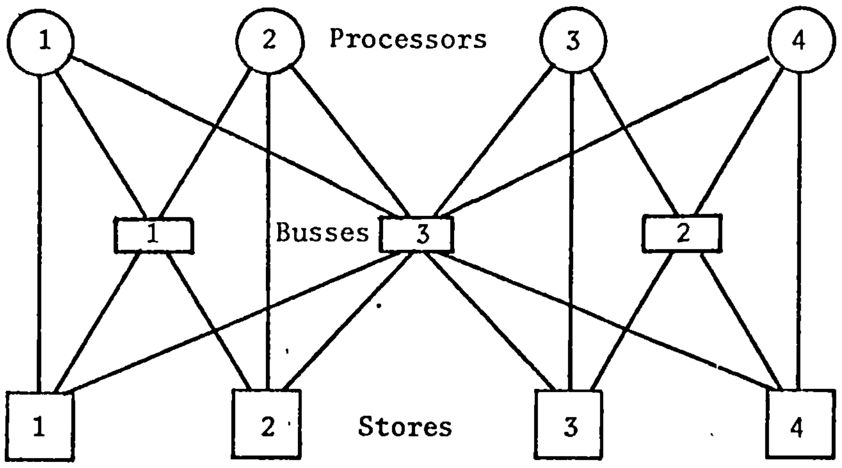


Figure 2.10

These processors are grouped together into clusters, and the processors of each cluster can access the local memories of all the other processors in the same cluster, but at a greater access time penalty compared to accessing the processors own local memory. In the diagram processors 1 and 2 form one cluster and processors 3 and 4 form the other. Processor 1 can access the local memory of processor 2 via the number 1 bus. Finally each processor in a cluster can access the memories of any processor in another cluster, but with a still higher access time penalty. Thus processors 1 or 2 can access the stores of the other cluster via the number 3 bus. However these accesses are now in possible conflict with three other processors.

Such a structure would be specified to the resource allocator by giving information about the processors, the memories and the bus interconnection network between these. From the point of view of the resource allocator, this computer system consists of a large number of processors each capable of accessing the entire memory. Some of these accesses will be direct and some by the means of intermediate buses. Therefore in this architecture there is no impediment to treating the memory as one common memory and allocating processes to processors as they become available. However the execution time will, naturally, be degraded by both memory interference and slow access times to nonlocal memory. So in allocating a program onto this architecture it will be

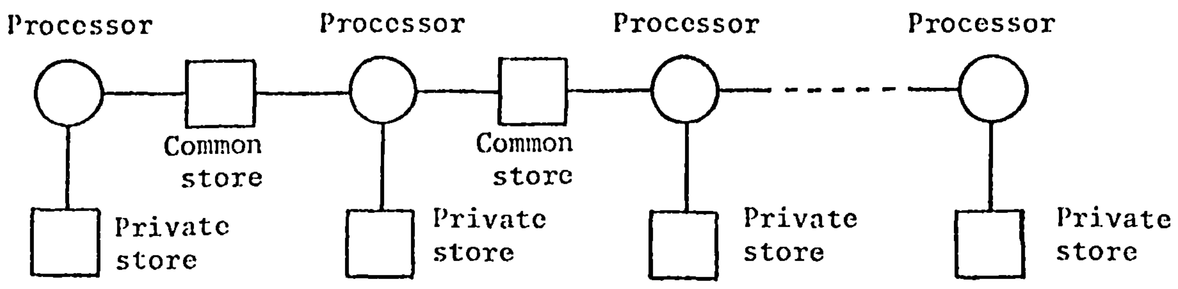


Figure 2.11

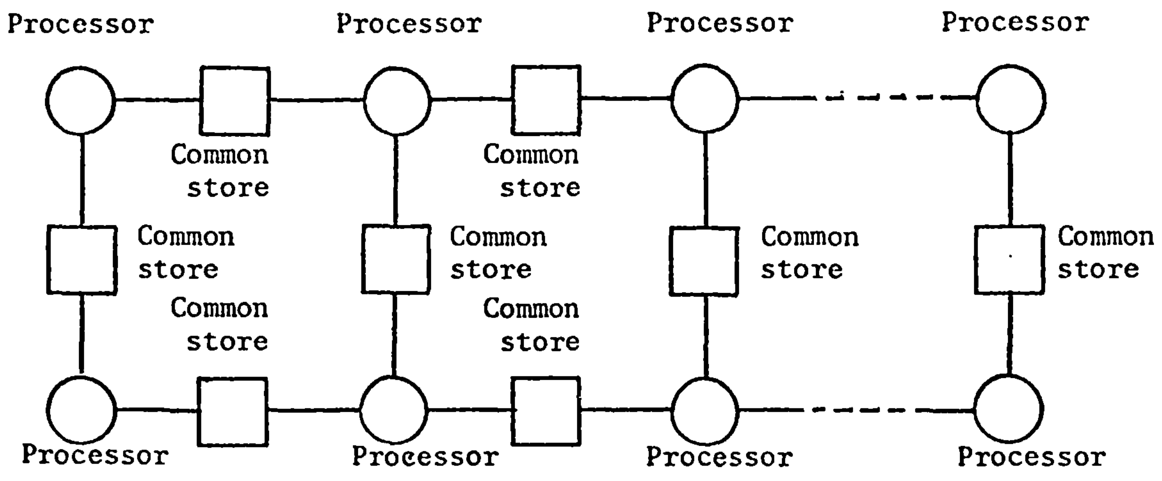


Figure 2.12

important to produce an efficient implementation. The resource allocation will be mainly concerned with reducing the possibility of memory conflicts.

(2.3.3) SYSTOLIC ARCHITECTURE

A systolic architecture, as described in [55], is one where data flows down a series of computer elements, each computer accepting information from its neighbour on one side, operating on it and sending it on to its neighbour on the other side. A design that fits this definition is given in figure(2.11), where the computing elements have their own private stores, and communicate with their right and left hand neighbours via the common physical memory elements. Such architectures are useful when the applications problem can be split into a number of stages of roughly equal computing load, and each stage can follow on from the one before it. One such application is in three dimensional computer graphics, where a program may be divided into processes to

Perform object ordering in depth first order.
Elimination of objects entirely out of view.
Removal of polygon faces facing the wrong way.
Three dimension to two dimensional coordination transformation.
Hidden line elimination.
Final drawing of the lines onto the screen.

If there are seven processors in the architecture, then the resource allocator can simply allocate a process to each processor. The resource allocator would be even more useful when there are less than this number of processors, since now some processes have to share a processor with a neighbouring process. These processes will be selected upon the basis of their workloads. An alternative systolic architecture could be constructed as is shown in figure(2.12) with two processors at each stage. This would make the resource allocation even more nontrivial.

CHAPTER (3)

(3.1) INFORMATION SPECIFICATION LANGUAGE

The information specification language (ISL) allows a machine understandable definition of a computer architecture to be constructed. It also provides the user with the facilities to guide the resource allocation activity.

This chapter will describe the basic underlying graph structure of this language, and introduce the parts of the language concerned with the definition of a computer architecture. The reference text used for the basic graph theory is [48].

(3.1.1) UNDERLYING INFORMATION STRUCTURE

Starting with a denumerable set X={X1,X2,...Xn} and a mapping M of X into X, a graph is the pair G=(X,M).

The ISL associates two functions with the set of elements of such a graph. One function is a mapping Fv from the set X to the set V, where V={null,V1,V2,...}. This is called the value function. The other function is a mapping Fn from the set X to the set N, where N={null,N1,N2,...}. This is called the name function.

A graph can be represented on paper by drawing vertices and arcs. A vertex is drawn as a point and corresponds to an element in X. A directed arc is drawn as an arrow from one vertex to another vertex. A directed arc exists from vertex Xi towards Xj if Xj is in the set M(Xi).

The value and name of each vertex may be represented also. If the name function Fn(Xi) of vertex Xi is nonnull, it is written alongside the vertex. If the value function Fv(Xi) of vertex Xi is nonnull, it is also written alongside the vertex. If both the name and value functions are nonnull, then the representation of the name is written first, followed by an = and then the written representation of the value.

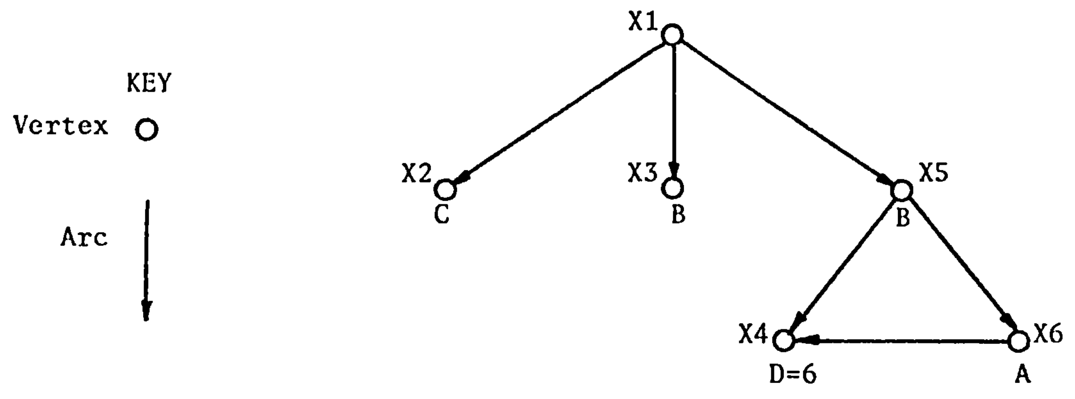


Figure 3.1

The graph of figure(3.1) provides an example representation of G=(X,M), where

X = { X1, X2, X3, X4, X5, X6 }

M(X1) = { X2, X3, X5 }	Fv(X1) = null	Fn(X1) = null
M(X2) = { }	Fv(X2) = null	Fn(X2) = C
M(X3) = { }	Fv(X3) = null	Fn(X3) = B
M(X4) = { }	Fv(X4) = 6	Fn(X4) = D
M(X5) = { X4, X6 }	Fv(X5) = null	Fn(X5) = B
M(X6) = { X4 }	Fv(X6) = null	Fn(X6) = A

Note that the function Fn does not necessarily give a unique name to each vertex. This graph has the name of the element from the set X written next to each vertex. In general this set identification is not needed in subsequent discussions about the ISL and so will rarely be mentioned after this section.

A directed arc U is represented by the pair (Xi,Xj). Xi is called the initial extremity of the arc and Xj is called the terminal extremity. An arc U is connected to a vertex Xi if U=(Xi,Xn) or if U=(Xn,Xi), Xi<>Xn. A directed path is a finite sequence of arcs (U1,U2, ... Ux) such that the final extremity of arc Un conincides with the initial extremity of arc Um, where m=n+1. A path is represented by the vertices which it contains, thus (X1,X5,X6) is a path in figure(3.1), and has the arcs (X1,X5) and (X5,X6).

A vertex Xj is attached to vertex Xi if Xj is a member of M(Xi). The attached vertices of Xi are all Xj such that this condition holds.

Given a vertex Xi, the connection set C of Xi is the set of all the vertices Xj, Xj<>Xi, such that there exists a directed path from Xi to Xj. In the figure some connection sets are

C(X1) = { X2, X3, X4, X5, X6 }
C(X2) = { }
C(X5) = { X4, X6 }
C(X6) = { X4 }

Any vertex Xi in the graph G, which is not in any set M(Xj), is called a root of the graph. That is there are no arcs whose terminal extremity coincide with a root vertex. In the example graph of figure(3.1), the vertex X1 is the root.

The graphs used by the ISL have some common properties. There is always one and only one root. If Xi is a vertex in the graph G, then there will always exist a directed path from the root vertex to Xi. Thus the connection set C(Xr)=X, where Xr is the root vertex. For the root vertex Xr, Fn(Xr)=null and Fv(Xr)=null. For all Xi where Xi<>Xr, Fn(Xi)<>null and Fv(Xr) can be null or nonnull.

In the following there is a brief overview of how the ISL may be used to construct a graph structure, and how to access such a graph once it exists. In appendix E a more detailed description appears. Chapter 4 continues with a discussion on how the ISL may be used to specify a computer architecture.

(3.2) OVERVIEW OF THE ISL GRAPH OPERATIONS

In the ISL there are operations that allow a graph to be constructed, and sets of vertices from this graph to be specified. There are also the more conventional high level language features which provide for arithmentical expressions, program flow control and the like.

A **graph** defined by the ISL always starts from a root vertex, which is denoted by a @ character. Other vertices, which may be directly or indirectly attached, can only be accessed via this root vertex. The simplest selection reference is

```
@
```

which will produce a reference set containing only the root vertex. The reference

```
@.N
```

will select all those vertices of name N that is attached to the root vertex.

Reference set variables may also be **used**, thus

```
V := @.N
```

will assign to the reference set V all the vertices named N that are attached to the root vertex. Now the reference expression

```
V.M
```

will generate the set of all the vertices of name M that are attached to any of the vertices in the reference set V. This is equivalent to the reference set expression

```
@.N.M
```

Instead of selecting all the vertices of a given name, a subset of these may be chosen, depending upon some additional criterion. For example

```
@.A.<NOT_EMPTY (@.B)>
```

will select only those vertices of name A which are attached to the root vertex, which themselves have one or more vertices of name B attached. Another example is

```
@.A.<NUMBER (@.B) = 2>
```

which will select only those A vertices which have exactly two B vertices attached.

Having selected a set of vertices, they may be used to create new edges in the graph, as in

```
@.A.B -> @.A.C
```

This will attach every C vertex defined in the second reference set expression to every B vertex defined in the first reference expression. Figure E.8 shows a diagram of this.

As well, new vertices may be created by using the NEW operation, as in

```
@ -> NEW ( A=3 )
```

which will create a new A vertex, give it a value of 3, and attach it to the root vertex. Another example is

```
@ -> (NEW(A), NEW(A), NEW(B))
```

which will create two new A vertices and one new B vertex, and attach them to the root vertex.

Program flow control constructs are provided to implement FOR loops and IF conditionals. As an example, the creation of three new D vertices might be achieved by

```
FOR I := 2 TO 4 DO
  @.C(I) -> NEW(D)
END ;
```

Each new D vertex will be attached to one of the C vertices numbered 2 to 4.

An example of a conditional statement is

```
IF 1 > 2 THEN
   @ -> NEW(A)
END ;
```

Finally the graph manipulation statements of the ISL may be grouped into procedure blocks and these procedures invoked by using call statements.

CHAPTER (4)

(4.1) USING THE INFORMATION SPECIFICATION LANGUAGE GRAPHS

Computer architecture specifications are used to specify the architecture of a (possibily multiprocessor) computer to the resource allocator. This information allows the allocator to deal with the allocation of code and data parts of a program onto the hardware processors and memory elements of the computer system.

For this purpose the resource allocation algorithms required a model of the computer system which contains information about the

Address ranges and sized of the physical memory elements,
The names of the processors,
The cycle speeds of the memories and processors,
The interconnections between processors and memories,
Information about the I/O system and interrupt addresses,

However there is no need for further knowledge of the system architecture in terms of registers, data and address buses or detailed knowledge of the input and output logic.

Consequently the user enters the information, by the means of the information specification language, in terms of the processor elements and memory blocks of the system, and their interconnections. All of this is standard information directly operated upon by the resource allocator algorithm. Extra user defined information may also be inserted and specifications written to operate on these. This is useful to aid the allocator in its global allocation strategy. It allows the programmer to specify information not easily accessible to the allocator.

(4.2) BASIC COMPUTER ARCHITECTURE SPECIFICATION

(4.2.1) A SIMPLE SYSTEM

The simplest computer system is one processor connected to a single memory unit. This can be described by

```
GRAPH
BEGIN
   @ -> NEW ( PROCESSOR ) -> NEW ( ADDRESS ) ;
END ;
```

This specifies to the resource allocator that a computer architecture has a processor and a memory module. The address range which the processor refers to the memory unit will be given by extra vertices attached to the address vertex. In subsequent specifications, to refer to the processor the reference used is

```
@.PROCESSOR
```

and to refer to the address range the reference used is

```
@.PROCESSOR.ADDRESS
```

The resource allocator will recognize the PROCESSOR identifier to be one of the standard identifiers which in this case refers to an actual hardware processor. Such processors can have properties that are directly understood by the allocator. This information includes the processors name and its cycle speed and this is represented by vertices attached to the PROCESSOR vertex. These have the standard names NAME and CYCLE. They may be defined for the example system as

```
GRAPH
BEGIN
   @ -> NEW ( PROCESSOR ) ->
      (  NEW ( ADDRESS ) ,
         NEW ( NAME = 'BRANDX' ) ,
            NEW ( CYCLE = 2.5 ) ) ;
      END ;
```

The PROCESSOR definition is as before. This vertex now has attached to it two new vertices, one called NAME and the other called CYCLE. They convey information about the name of this processor, BRANDX, and its cycle time, 2.5 microseconds. These values can be referenced by

```
      VALUE ( @.PROCESSOR.NAME )
      VALUE ( @.PROCESSOR.CYCLE )
```

(4.2.2) SPECIFYING MEMORY

Vertices named ADDRESS and PROCESSOR are directly understood by the allocator. It expects the ADDRESS vertex to have two further standard vertices attached. One vertex is called START and this has an integer value giving the start address at which the processor accesses the first memory byte of the memory module. The other vertex is called MEMORY and this vertex represents information about the physical memory module. This vertex has attached to it two further vertices, called ACCESS and SIZE. The ACCESS value gives the access time of the memory in microseconds, while the SIZE value gives the size of the memory in bytes. The ACCESS and SIZE vertices are not attached directly to the ADDRESS vertex, since different processors may have different address ranges in which they access this same memory.

As an example a memory unit of 4096 bytes for the computer system already defined can be specified by the addition of the statements

```
      @.PROCESSOR.ADDRESS ->
         ( NEW ( START = 0 ) , NEW ( MEMORY ) ) ;
```

This attachs two new vertices to the ADDRESS vertex. They are START and MEMORY, the START vertex has the value of 0. Information for the MEMORY vertex is further specified by

```
      @.PROCESSOR.ADDRESS.MEMORY ->
         ( NEW ( SIZE = 4096 ) , NEW ( ACCESS = 0.45 ) ) ;
```

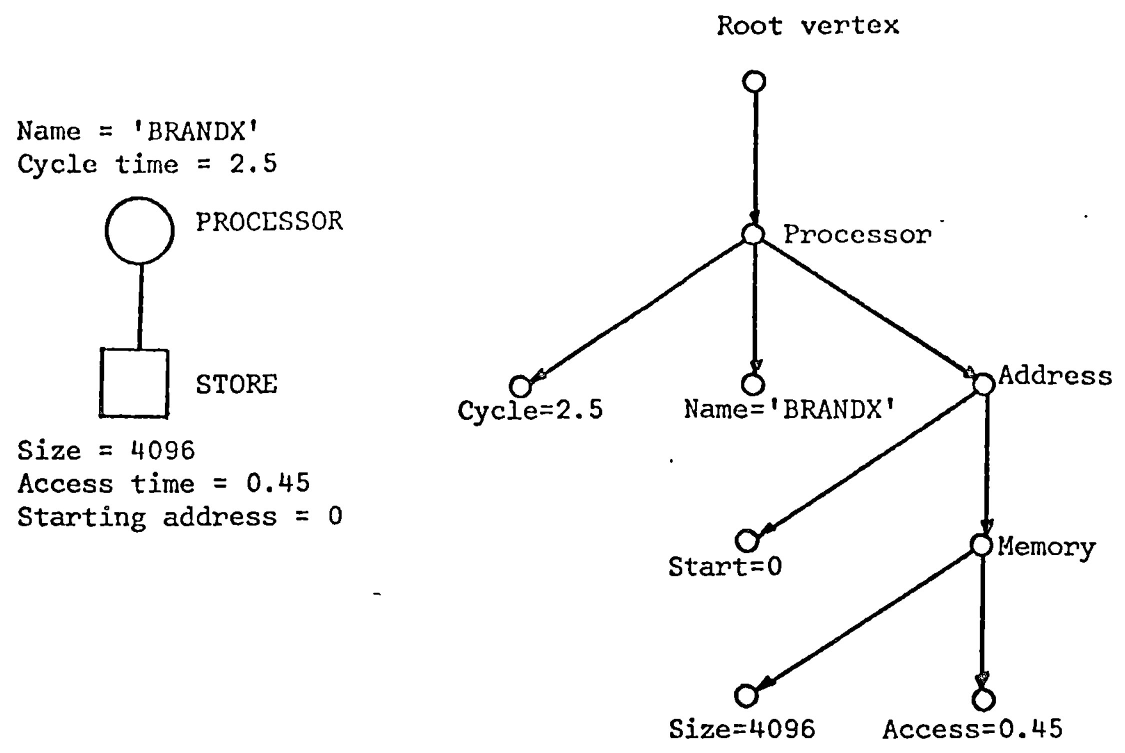


Figure 4.1

so specifying a memory with a size of 4096 bytes and a 450ns access time.

This, combined with the earlier specifications and set out in a slightly different way, results in the complete specification program like

```
GRAPH
   CONST NAME_VALUE = 'BRANDX' ,
         CYCLE_VALUE = 2.5 ;
BEGIN
   @ -> NEW ( PROCESSOR ) ->
   (  NEW ( CYCLE = CYCLE_VALUE ) ,
      NEW ( NAME = NAME_VALUE )   ,
      NEW ( ADDRESS ) ->
      (  NEW ( START = 0 ) ,
         NEW ( MEMORY ) ->
         (  NEW ( SIZE = 4096 ) ,
            NEW ( ACCESS = 0.45 )
```

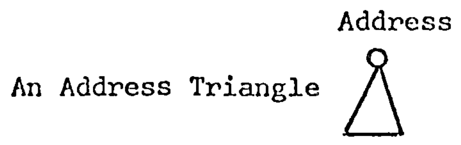


Figure 4.2

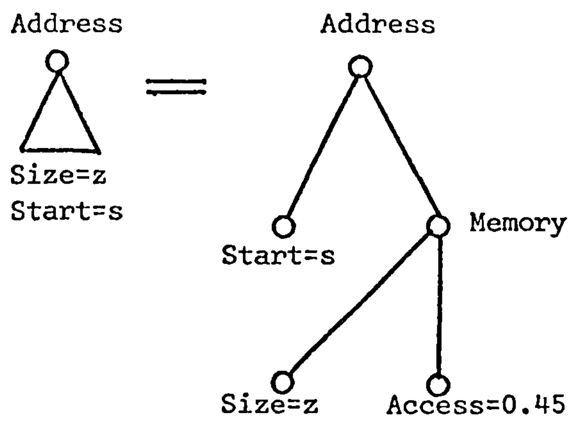


Figure 4.3

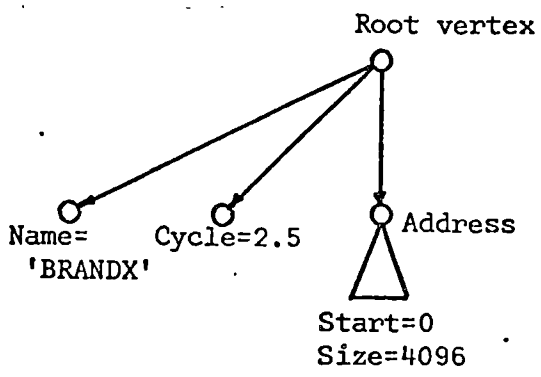


Figure 4.4

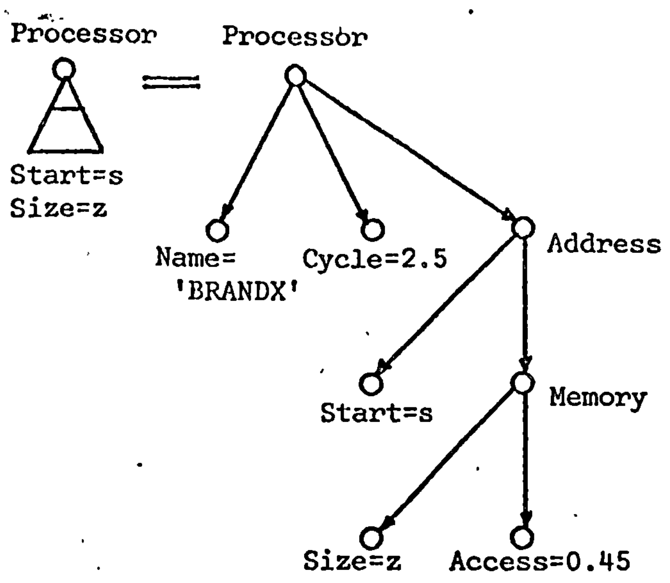


Figure 4.5

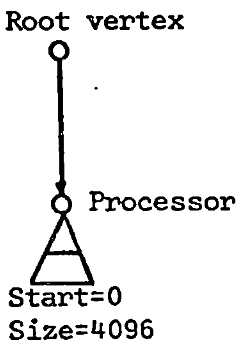


Figure 4.6

```
            )
         )
      )
   END ;
```

Thus this represents a computer architecture with a processor called BRANDX having a processor cycle time of 2.5 microseconds. This processor has access to 4096 bytes of 0.45 microsecond store attached, with the store occupying the first 4096 bytes of the processors addressing range. The graph representation of this is shown in figure(4.1).

To reduce the size of the graph diagrams in the following text, a visual shorthand representation is used. A triangle like that of figure(4.2) is called an address triangle. It is taken to represent an ADDRESS vertex and all of the vertices that are shown in figure(4.1) to be attached to this ADDRESS vertex. Its equivalent graph is given in figure(4.3), using this the graph of figure(4.1) can be redrawn as shown in figure(4.4). The SIZE and ADDRESS vertex values are given under the triangle. These are only specified in the following graphs if their values are important for the ISL example being demonstrated. Otherwise they are not explicitly mentioned.

An even more compact representation of the graph of figure(4.1) is provided by using a processor triangle defined as in figure(4.5). In figure(4.6) the graph of figure(4.1) has been redrawn this way. As with the memory triangle, the values of the vertices that have values attached are only explicitly provided if it is required for the example demonstration.

(4.2.3) MULTIPLE MEMORIES

In more complex computer systems a processor may access more than one memory module. This is represented in the specifications by attaching more than one ADDRESS vertex to the same PROCESSOR vertex. The address vertices of a particular processor must have nonoverlapping address ranges and will generally have access to different memory modules.

An extra memory may be added to the computer system defined above by adding the specification

```
@.PROCESSOR ->
( NEW ( ADDRESS ) ->
   ( NEW ( START = 4096 ) ,
     NEW ( MEMORY ) ->
     ( NEW ( SIZE = 4096 ) , NEW ( ACCESS = 0.45 ) )
   )
) ;
```

There are now two vertices attached to the PROCESSOR vertex, both

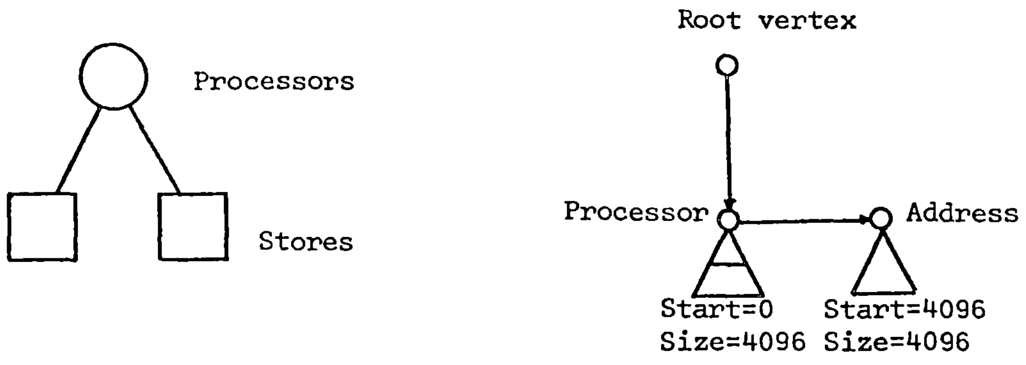


Figure 4.7

with name ADDRESS. This is depicted in figure(4.7).

Note that here the extra memory is represented by attaching the address triangle to the PROCESSOR vertex to which the processor Triangle is attached . Address vertices are always attached to the processor vertex if that processor accesses the memory, so this is possible.

The reference

    @.PROCESSOR.ADDRESS

will refer to both address vertices, and the reference

    @.PROCESSOR.ADDRESS.MEMORY

will refer to both of the memory modules.

To refer to only one of the address vertices indexing may be used. Thus to refer to the second memory module requires the reference

    @.PROCESSOR.ADDRESS(2).MEMORY

(4.2.4) USE OF PROCEDURE DEFINITIONS.

The length of the specifications will become long and thus their production tedious for any computer system having more than a few memory modules. Procedures may be advantageously used here. For an example a procedure defining a standard memory is given,

```
PROCEDURE STANDARD_MEMORY (
   C : SET ; START_VALUE , SIZE_VALUE : INTEGER ) ;
VAR MEM : SET ;
BEGIN
   MEM := NEW ( ADDRESS ) ->
   (  NEW ( START = START_VALUE ) ,
      NEW ( MEMORY ) ->
      (  NEW ( SIZE = SIZE_VALUE ) ,
         NEW ( ACCESS = 0.45
      )
   ) ;
   C -> MEM ;
END ;

PROCEDURE ONE_PROCESSOR ( C , PSR : SET ) ;
BEGIN
   PSR := NEW ( PROCESSOR ) ;
   C -> PSR ->
   (  NEW ( CYCLE = 2.5 ) , NEW ( NAME = 'BRANDX' ) ) ;
END ;
```

The first procedure creates a new ADDRESS vertex and attachs to this the vertices needed for a memory subgraph. This ADDRESS vertex is assigned to the MEM reference set variable. In the last statement of the procedure this vertex is attached to whatever vertices appear in the C formal parameter. If the memory subgraph had been directly attached to the C formal parameter, as in

```
C -> NEW ( ADDRESS ) ->
(  etc  ) ;
```

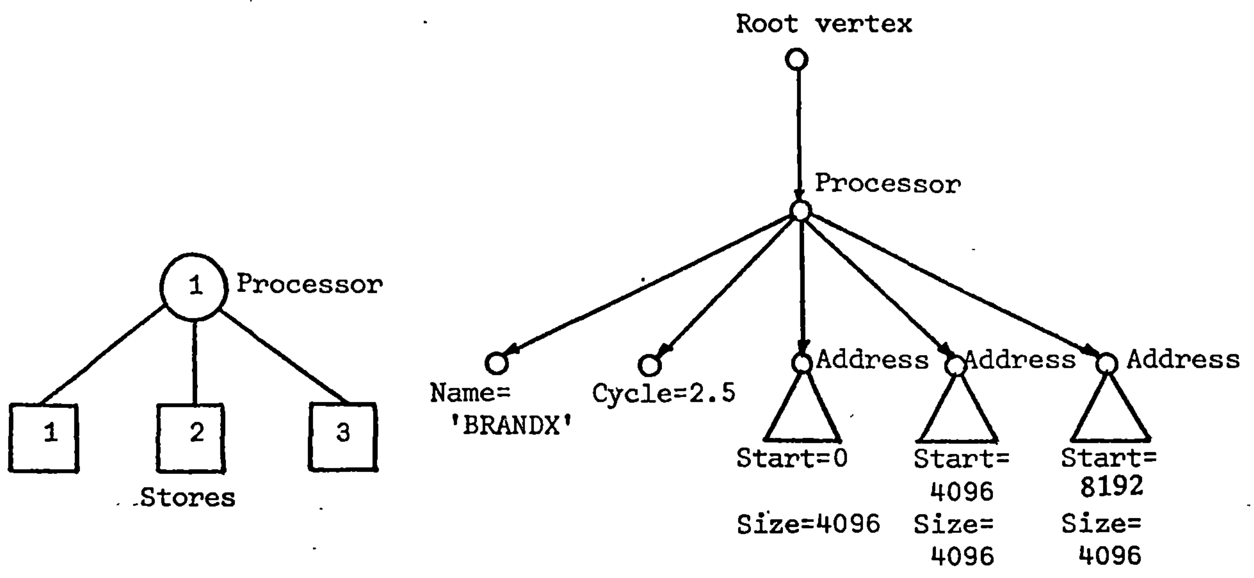


Figure 4.8

then there may be more than one subgraph created. If the procedure is called with three vertices in the actual parameter corresponding to the C parameter, there would be three such subgraphs created. The way that is choosen will result in only one subgraph being created and this will be attached to all of the vertices in the C parameter.

The second procedure creates a processor subgraph. This is attached to whatever vertices appear in the C parameter. The PSR formal parameter will contain the newly created PROCESSOR vertex upon the procedures return. This allows the memory information for the newly created processors to be attached to the correct PROCESSOR vertex.

These procedures contain all of the information needed to declare a processor and a memory. Thus the calls

```
ONE_PROCESSOR ( @ , PSR ) ;
STANDARD_MEMORY ( PSR , 0 , 4096 ) ;
```

will produce the graph of figure(4.1). Therefore the ISL equivalent of the processor triangle in figure(4.5) is these two statements above.

Alternatively, to declare a computer architecture with three standard memories attached requires

```
ONE_PROCESSOR ( @ , PSR ) ;
STANDARD_MEMORY ( PSR ,    0 , 4096 ) ;
STANDARD_MEMORY ( PSR , 4096 , 4096 ) ;
STANDARD_MEMORY ( PSR , 8192 , 4096 ) ;
```

and this will produce the graph of figure(4.8).

(4.3) VERTEX NAMES

In the specifications so far only predefined vertex names have been used. These are directly understood by the allocator, and do not need to be defined by the user. A list of the predefined names are

SIZE	NAME	START	CYCLE
MEMORY	ACCESS	ADDRESS	PROCESSOR
PORT	BANK	INTERRUPT	READ_PORT
WRITE_PORT	MEMORY_ACCESS	READ_WRITE_PORT	USER_ADDRESS

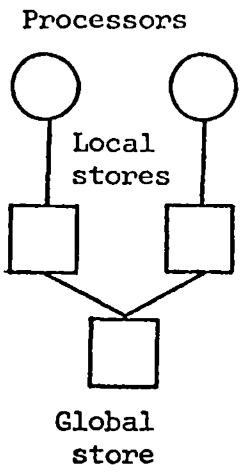
Those in the second part of the list have not yet been discussed. The user does not define these names, but does have to define any new names that may be used. For example, in the following the name SUB_SYSTEM is used. This is defined by

```
VERTEX SUB_SYSTEM ;
```

(4.4) SHARED MEMORY

So far the specification of a uniprocessor system has been described. The specifications may be expanded to deal with a computer architecture of two or more processors. The simplest way is to merely define two subsystems-

```
S := NEW ( SUB_SYSTEM ) ;
@ -> S ;
ONE_PROCESSOR ( S , PSR ) ;
STANDARD_MEMORY ( PSR , 0 , 8192 ) ;
```



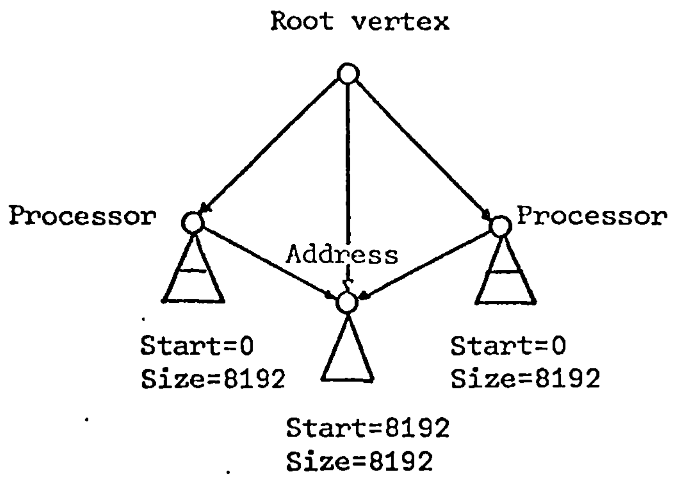


Figure 4.9

```
S := NEW ( SUB_SYSTEM ) ;
@ -> S ;
ONE_PROCESSOR ( S , PSR ) ;
STANDARD_MEMORY ( PSR , 0 , 8192 ) ;
```

These directives describe two independent processors, each with 8192 bytes of unshared memory. The use of shared memory is easily described by attaching the same memory vertex to the two separate address vertices of each processor that accesses this memory. The common memory for this is defined by

```
STANDARD_MEMORY ( @ , 8192 , 8192 ) ;
```

This is referenced be @.ADDRESS. The processors can then be defined as

```
FOR 2 DO
   S := NEW ( SUB_SYSTEM ) ;
   @ -> S ;
   ONE_PROCESSOR ( S , PSR ) ;
   STANDARD_MEMORY ( PSR , 0 , 8192 ) ;
END ;
@.SUB_SYSTEM.PROCESSOR -> @.ADDRESS ;
```

In this definition the two subsystems are created as before. The extra memory created by the first call to the standard memory procedure is attached to both of these new processors. The specifications now describe an architecture with two processors, each accessing their own local store, and both accessing a common global store. The graphs and computer structures produced by both of these examples are shown in figure(4.9). This pattern may be generalized to any number of processors accessing the same memory units-

```
PROCEDURE SUB_SYSTEM_PROCEDURE ( C , PSR : SET ) ;
VAR S : SET ;
BEGIN
   S := NEW ( SUB_SYSTEM ) ;
   C -> S ;
   ONE_PROCESSOR ( S , PSR ) ;
   STANDARD_MEMORY ( PSR , 0 , 8192 ) ;
END ;
```

The macro defines a new computer architecture portion which is given the name SUB_SYSTEM. The processor of this accesses its own local memory which is defined by the call to STANDARD_MEMORY. The SUB_SYSTEM vertex is attached to whatever vertices are in the reference set variable C. The PSR variable will contain the new PROCESSOR vertex, upon the return of the procedure.

From here the statements

```
STANDARD_MEMORY ( @ , 8192 , 8192 ) ;
FOR 10 DO
   SUB_SYSTEM_PROCEDURE ( C ,.PSR ) ;
   PSR -> @.ADDRESS ;
END ;
```

will create the common memory and 10 new subsystems. Each time through the loop a new SUB_SYSTEM will be created and the @.ADDRESS vertex will be attached to its new processor vertex.

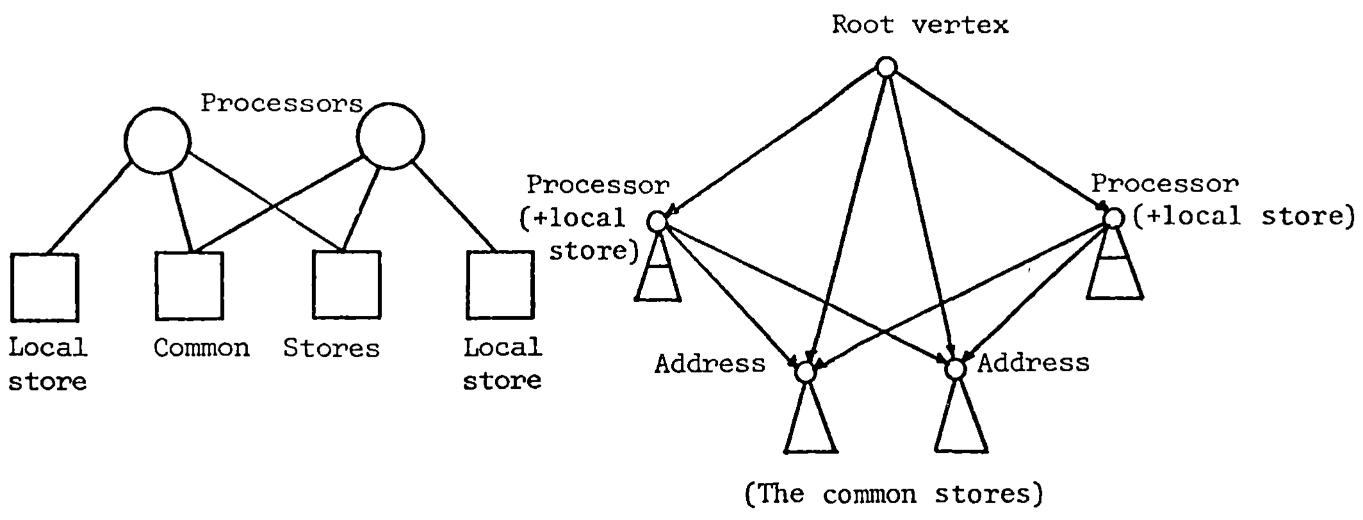


Figure 4.10

(4.4.1) MEMORY ACCESS INTERFERENCE

In the above system there are ten processors accessing the same common memory. Consequently there is the probability that two or more processors will attempt to access the memory at the same time. This requires memory arbitration logic whose function is to detect such clashes and to delay processor memory requests until the memory is free. How this is managed in the hardware is of no concern to the resource allocation problem. If two or more processors are specified to access the same memory then the resource allocator will assume that there is some kind of memory arbitration. This will result in memory contention, affecting the execution performance of the system. When performing the resource allocation the allocator will model this interference and take this information into account.

(4.4.2) DEPENDENT SHARED MEMORY

Consider the situation where there are two common memory blocks, perhaps defined as

```
      STANDARD_MEMORY ( @ , 8192 , 8192 ) ;
      STANDARD_MEMORY ( @ , 8192 , 16384 ) ;
      FOR 2 DO
         ONE_PROCESSOR ( @ , PSR ) ;
         STANDARD_MEMORY ( PSR , 0 , 8192 ) ;
         PSR -> @.ADDRESS ;
      END ;
```

and depicted in figure(4.10).

The common memories are defined by the first two calls to the STANDARD_MEMORY procedure. Each processor vertex created by the procedure has access to these, as well as access to a separate local memory vertex. This is defined for each processor by the call to the STANDARD_MEMORY procedure in the FOR loop. In this system each common memory is accessed independently by the processors, one busy common memory will not block the other common memory. The specifications described so far can readily describe this architecture.

However now consider the situation where a memory access to one of the common memories will block accesses by other processors to the other common memories. Such a situation could arise from a number of different kinds of architectures. Two possibilities are considered here. One is where a processor or group of processors access a number of memory blocks via bank switching. This is where each memory resides in the same memory addressing region of the processor and the appropriate memory bank is selected by a bank select instruction. The other possibility considered is a computer system built up with a number of processors, each having direct access to their own local memory by a dedicated bus, and each processor also having slower access to all the memories of the system via a common bus. These are considered in turn.

(4.4.2.1) MEMORY BANK SWITCHING

Memory bank switching is specified by attaching the memory vertices to a common vertex. This vertex is given the name BANK and is similar in use to an ADDRESS vertex in that it has attached to it a START vertex. This therefore implies that the different memory blocks are in the same memory range. An example is

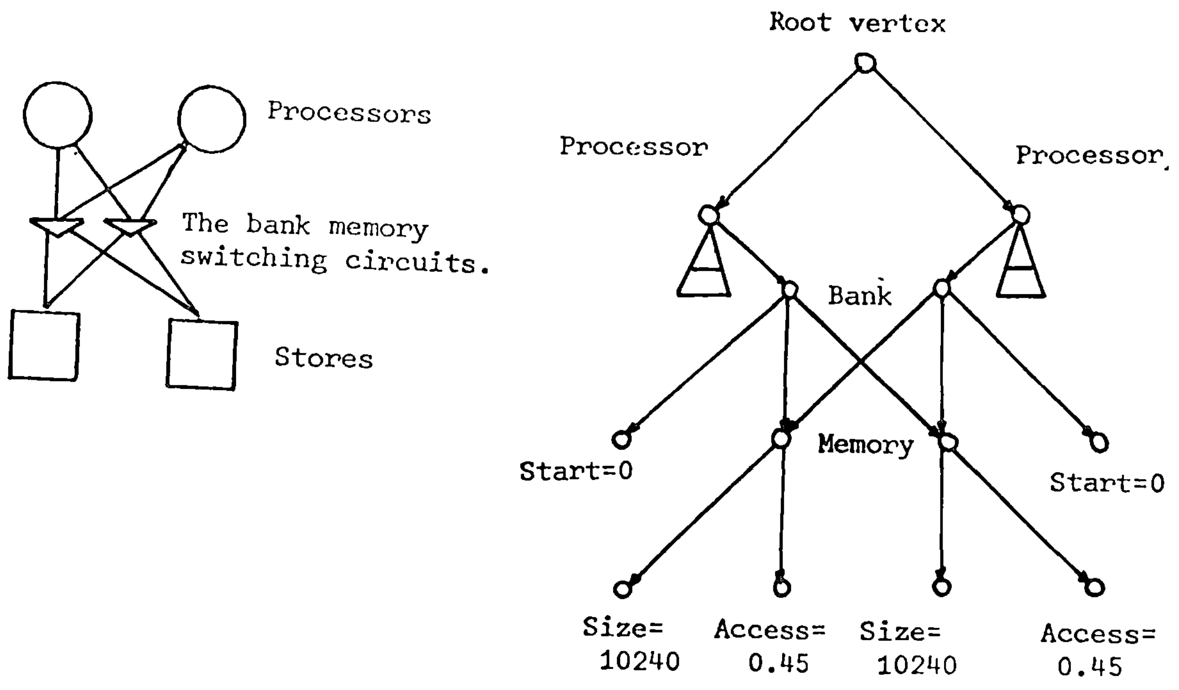


Figure 4.11

```
FOR 2 DO
  ONE_PSR ( @ , PSR )
  PSR -> NEW ( BANK ) -> NEW( START = 0 )
END
FOR 2 DO
  @.PROCESSOR(1).BANK -> NEW(MEMORY) ->
     ( NEW ( SIZE = 10240 ) , NEW ( ACCESS = 0.45 ))
END
@.PROCESSOR (2). BANK -> @.PROCESSOR(1).BANK.MEMORY
```

and this is depicted in figure(4.11).

Each processor now has access to two memory blocks, accessible in the addressing range of 0 to 10239. A memory bank select instruction has to be executed by the executing code to select which particular memory bank is to be used. The resource allocator inserts the appropriate bank selection instructions into the code in its linking stage.

(4.4.2.2) MEMORY MAP SYSTEM

The other possible memory structure requires the use of another predefined property name, MEMORY_ACCESS. This standard vertex name is used to represent the connection of several processors to a single memory system, where only one access at a time can be performed. Thus it indicates where memory arbitration is applied to a number of memory blocks, and not just to one memory block. To demonstrate the directives, a computer system with 4 processors and 4 memories is specified,

```
GRAPH
   INDEX I ;
   VAR PSR : SET ;

   PROCEDURE STANDARD_MEMORY (
      C : SET ; START_VALUE , SIZE_VALUE : INTEGER ) ;
   VAR MEM : SET ;
   BEGIN
      MEM := NEW ( ADDRESS ) ->
      (  NEW ( START = START_VALUE ) ,
         NEW ( MEMORY ) ->
         (  NEW ( SIZE = SIZE_VALUE ) ,
            NEW ( ACCESS = 0.45
         )
      ) ;
      C -> MEM ;
   END ;

   PROCEDURE ONE_PROCESSOR ( C , PSR : SET ) ;
   BEGIN
```

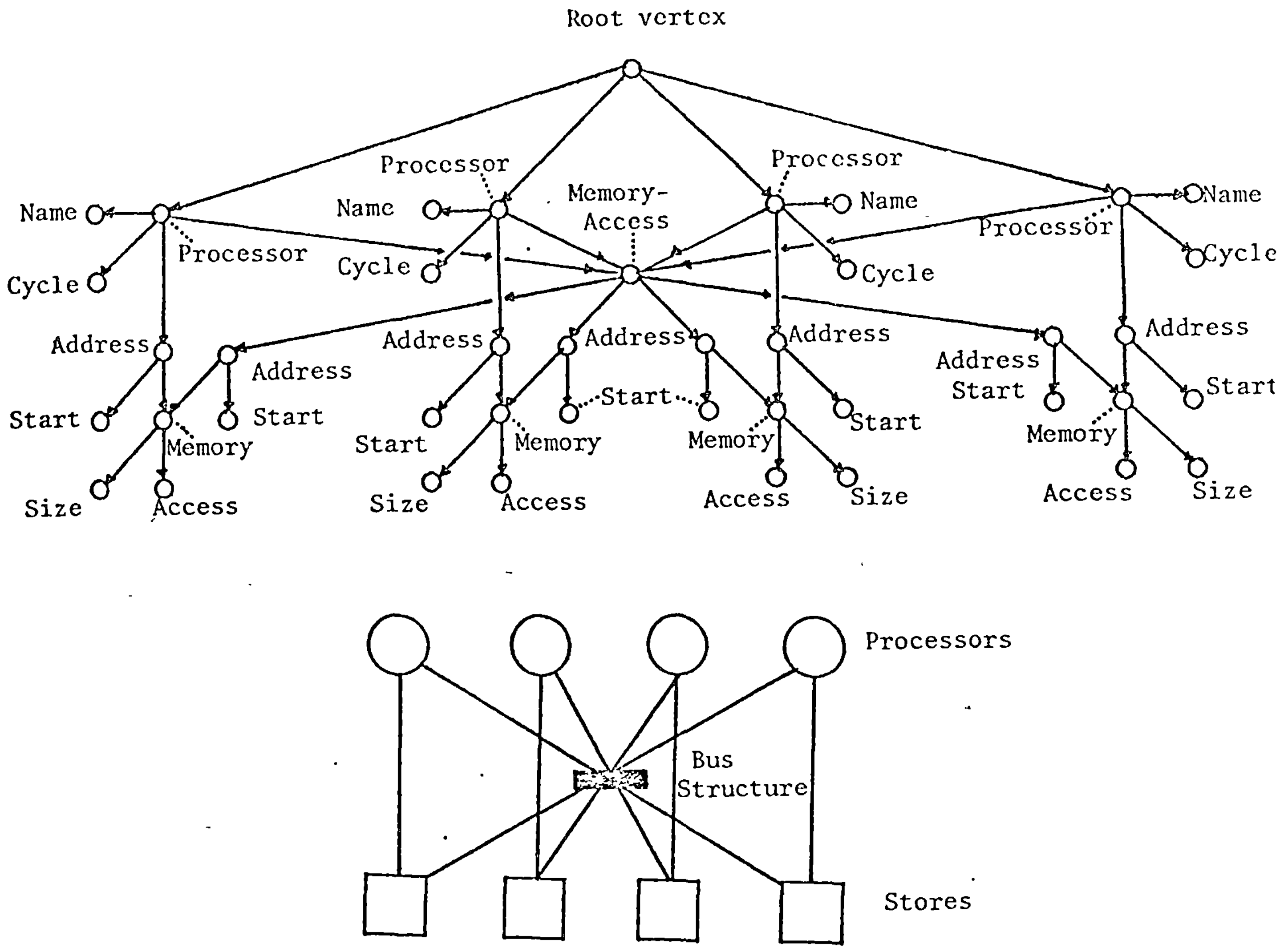


Figure 4.12

```
   PSR := NEW ( PROCESSOR ) ;
   C -> PSR ->
   (  NEW ( CYCLE = 2.5 ) , NEW ( NAME = 'BRANDX' ) ) ;
END ;

PROCEDURE MAP ( PSR : SET ) ;
VAR ME, P : SET ;
     I : INTEGER ;
BEGIN
   ME -> NEW ( MEMORY_ACCESS ) ;
   PSR -> ME ;
```

```
      I := 1 ;
      FOR P := EACH ( PSR ) DO
         ME -> NEW ( ADDRESS ) ->
         (  NEW ( START = 8192 * I ) , P.ADDRESS.MEMORY ) ;
         I := I + 1 ;
      END ;
   END ;

BEGIN
   FOR 4 DO
      ONE_PROCESSOR ( @ , PSR ) ;
      STANDARD_MEMORY ( PSR , 0 , 8192 ) ;
   END ;
   MAP ( @.PROCESSOR ) ;
END ;
```

The first procedure defines a standard memory module. The second procedure is a definition for one processor which accesses a standard memory. The third procedure defines the map structure, it creates a new MEMORY_ACCESS vertex and attaches it to the processor vertices. Then it attaches the MEMORY vertex of each processor to this MEMORY_ACCESS vertex via a new ADDRESS vertex. Thus, as is shown in figure(4.12), each processor ends up with direct access to its own local memory and indirect access to all the other memories of the computer architecture.

The address range of the local memory block is in the range of 0 to 8191. The addresses by which each processor accesses the nonlocal memories is in increments of 8192, starting at 8192 for the first nonlocal memory. Note that according to this description each processor has access to its own local memory twice, once through the local address range, and once through the nonlocal address range. In situations like this the resource allocator will assume that address accesses to the local memory are to be made in the most direct manner possible.

At any one time each memory may be servicing only one memory request. This request may come from the local processor, and at any one time all the processors may be accessing their own local memories. This request may also come from some other processor via the memory mapping logic. In this case only one such nonlocal request may be in progress, in the entire system, at one time.

(4.4.2.3) MULTILEVEL MEMORY MAPPING SCHEMES

The specification above can easily be extended to a computer system with a two level memory mapping hardware. Such a computer system will have a number of processors, each with their own local memory on a dedicated bus, and each processor will have access to all of the nonlocal memory units on a shared bus. This access is extended from that of a one level map by dividing the processor and local memory pairs into groups. A memory access request from a processor to a nonlocal memory which is within the same group can be made independently of other such accesses in other groups. One example of such a design is the Cm* computer architecture [26].

Thus there are three grades of accesses. The fastest are from a processor to its own local memory, and all processors in the computer system may make such requests simultaneously. The second in speed is from a processor to a nonlocal memory within the same group. There may be one such request within each group. The slowest is a request from a processor to a nonlocal memory not in its own group, and only one of these requests may be made at one time.

The degradation in speed in these requests may come about because of delays introduced by the memory arbitration logic used to connect numbers of memories and processors together. Most likely, however, the main degradation will come from memory contention, and this contention is what the resource allocation tries to minimize.

The following specification follows the same pattern used in the specification of a single level map, using MEMORY_ACCESS vertices to indicate dependent memory access paths.

Firstly a GROUP vertex name definition is added to the specifications and then the processors and stores of the computer architecture are defined by

```
FOR 4 DO
    G := NEW ( GROUP ) ;
    a -> G ;
    FOR 4 DO
```

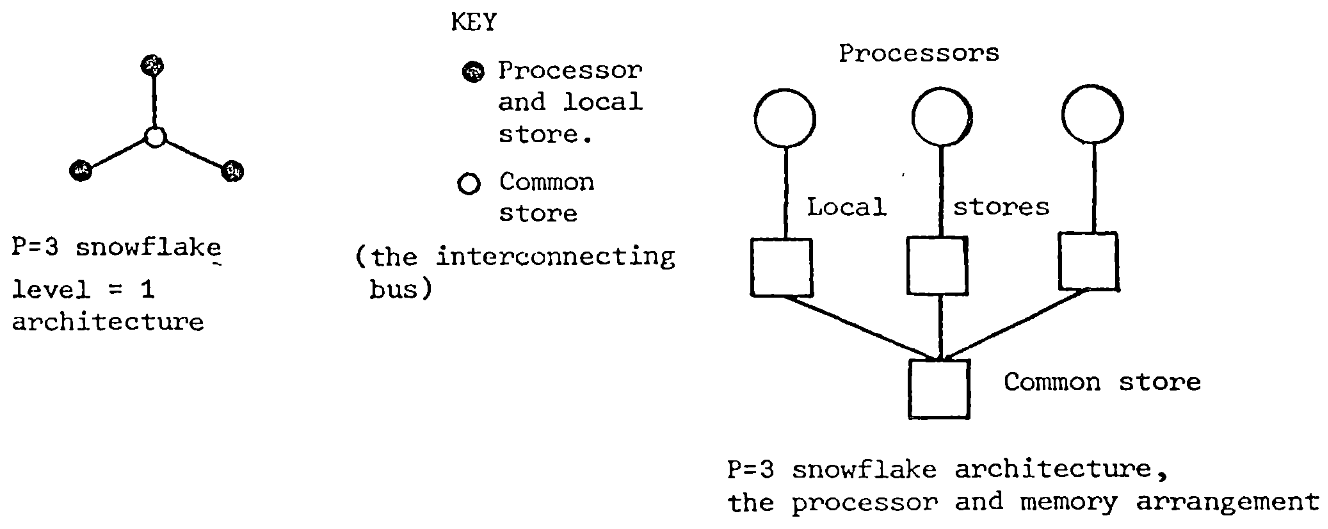

KEY
● Processor and local store.
○ Common store
(the interconnecting bus)
P=3 snowflake level = 1 architecture
Processors
Local
stores
Common store
P=3 snowflake architecture, the processor and memory arrangement

Figure 4.13

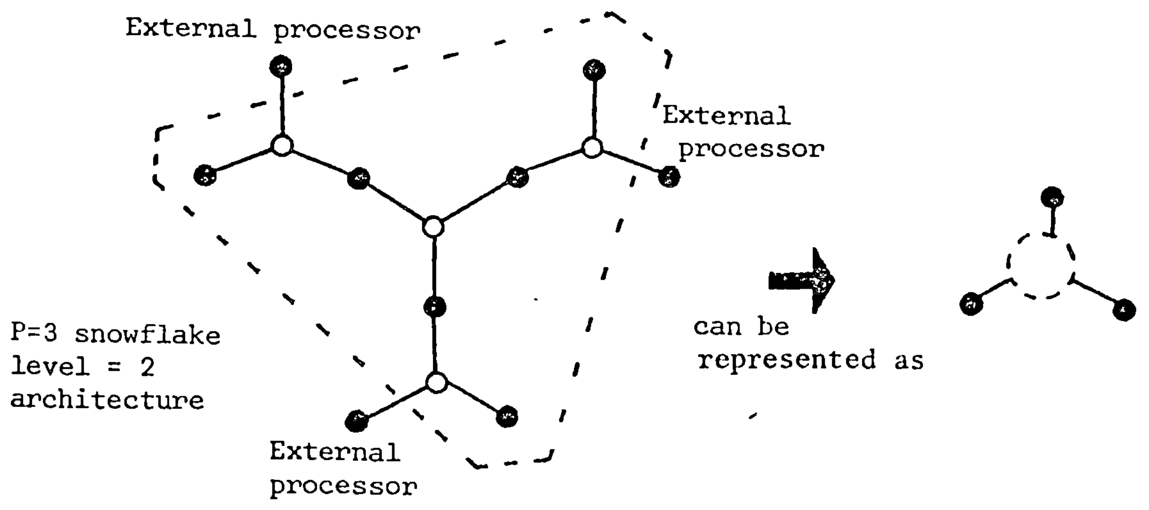


Figure 4.14

```
        ONE_PROCESSOR ( G , PSR ) ;
        STANDARD_MEMORY ( PSR , 0 , 8192 ) ;
     END ;
  END ;
```

This creates four GROUP vertices, each with four processors and their own local memory. Now to create the map structure requires

```
  FOR G := EACH ( @.GROUP ) DO
     MAP ( G.PROCESSOR ) ;
  END ;
  MAP ( @.GROUP.PROCESSOR ) ;
```

In the first statement the map procedure is applied separately to each GROUP vertex of the information structure and this will place the processors of each group into a single map. In the second statement the map procedure is applied to a reference which refers to all the processors of the architecture. This results in all of these processors being placed into a fifth map.

(4.5) SNOWFLAKE ARCHITECTURE

The snowflake architecture, as described in [21], is defined using the ISL in the following. This provides an example specification of a nontrivial computer architecture.

A first level snowflake has P processors and a single bus connecting these. Here this bus is provided by the P processors accessing a common memory. Figure(4.13) shows a first level snowflake for P equal 3.

A second level snowflake is constructed from P first level snowflakes and an extra bus. One processor from each first level snowflake is connected to this new bus. Another processor from each first level snowflake becomes the external processor. The remaining P-2 processors are internal processors. An external processor is used when a third level snowflake is constructed, with the bus for the third level being connected to these external processors. Thus a second level snowflake is shown in figure(4.14,left).

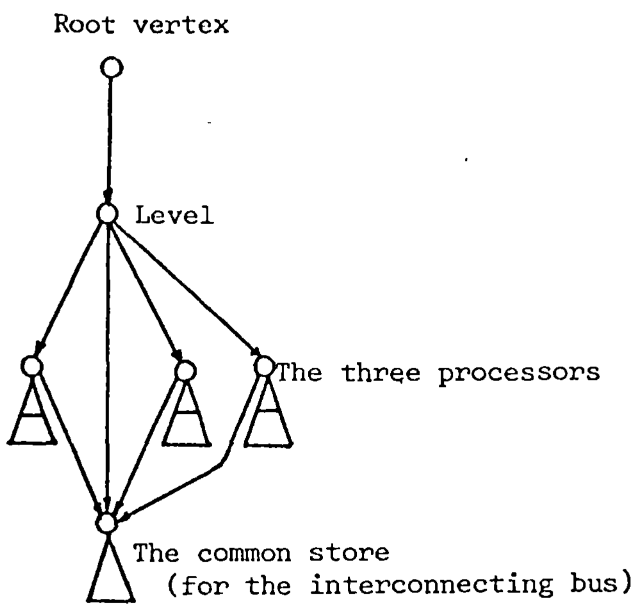


Figure 4.15

In figure(4.14,right), only the external processors of the snowflake have been drawn, the remainder of the snowflake is hidden in the dashed circle in the middle. This looks like the P=3 level 1 snowflake in figure(4.13). From this it can be seen how the construction of a snowflake for the next level up can be achieved.

The definitions required for a P=3 snowflake are now developed. Firstly the level one snowflake is just three processors, each with their own memory, and each with access to another common memory. This is specified by a procedure definition

```
PROCEDURE FIRST_LEVEL ( C : SET ) ;
VAR L , PSR : SET ;
BEGIN
   L := NEW ( LEVEL ) ;
   C -> L ;
   FOR 3 DO
      ONE_PROCESSOR ( L , PSR ) ;
      STANDARD_MEMORY ( PSR , 0 , 8192 ) ;
   END ;
   STANDARD_MEMORY ( L.PROCESSOR , 8192 , 1024 ) ;
END ;
```

The first statement creates a new LEVEL vertex. This is attached to the vertices in the C reference set. Then three processor subgraphs are constructed and attached to this LEVEL vertex. Each processor vertex has a memory subgraph attached to it. This indicates that the processors have 8192 bytes of local memory. The last statement will attach another memory subgraph to all three processor vertices. This common memory has 1024 bytes capacity and its starting address is 8192.

The architecture specified by this FIRST_LEVEL snowflake is as shown in figure(4.13). The specification graph for this is shown in figure(4.15). Note the the LEVEL vertex has the three processors attached.

To construct a level N snowflake, three level N-1 snowflakes are used. A new bus is created, and one processor from each of these N-1 snowflakes is attached to it. Another processor from each of the N-1 snowflakes is marked as being an external processor, by attaching it to the LEVEL vertex. Thus the procedure definition is

```
PROCEDURE SNOWFLAKE ( C : SET ; LEVEL_NO : INTEGER ) ;
VAR L : SET ;
BEGIN
   IF LEVEL_NO = 1 THEN
      FIRST_LEVEL ( C ) ;
   ELSE
      L := NEW ( LEVEL ) ;
      C -> L ;
      FOR 3 DO
         SNOWFLAKE ( L , LEVEL_NO-1 ) ;
      END ;
      FOR I := 1 TO 3 D
         L -> L.LEVEL(I).PROCESSOR(1) ;
      END ;
      BUS ( L ) ;
   END ;
END ;
```

Here the LEVEL_NO constant in the procedure parameter list indicates the level that is to be constructed. If this is a first level snowflake,

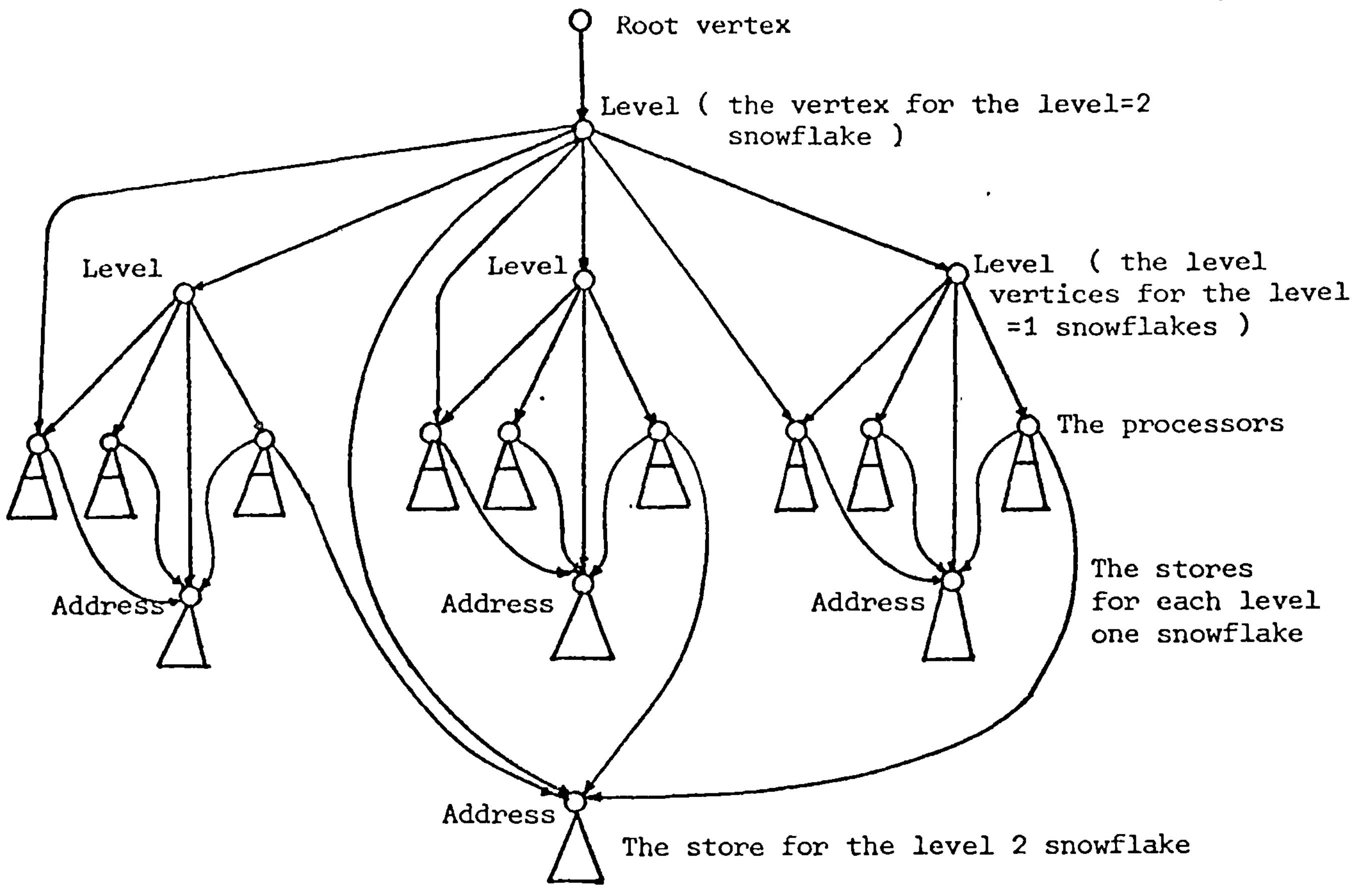

Root vertex
Level (the vertex for the level=2 snowflake)
Level
Level
Level (the level vertices for the level =1 snowflakes)
The processors
The stores for each level one snowflake
Address
Address
Address
Address
The store for the level 2 snowflake

Figure 4.16

then the IF statement will select the call to the FIRST_LEVEL procedure. Otherwise a new LEVEL vertex is created and three calls to the snowflake procedure are made. These construct the N-1 level snowflakes. After this the first processor of each N-1 level is attached directly to this level. These processors are the external processors that may be used to attach to buses to create higher level snowflakes. Finally a call is made to the BUS procedure. This will create the bus for this level and attach the correct processors to it. The definition of this bus is

```
PROCEDURE BUS ( L : SET ) ;
VAR ADR : SET ;
BEGIN
   STANDARD_MEMORY (
      L.LEVEL.PROCESSOR (2) , 9216 , 1024 ) ;
END ;
```

This procedure creates the common memory subgraph and attachs this to each of the three processors. These processors are choosen to be the second processor of each of the three attached N-1 level snowflakes.

Notice that at each level the LEVEL vertex has attached to it the three LEVEL vertices of the N-1 level, the ADDRESS vertex of the memory used for the bus at this level, and the three processors which may be used in a level N+1 snowflake. To show this a level 2 snowflake is represented in the graph of figure(4.16).

(4.6) INPUT, OUTPUT AND INTERRUPTS

In this section the specifications for hardware input, output and interrupts are described. The directives for this information will only describe the addresses and read/write status of input and output ports, addresses of interrupts and addresses of variables. All other hardware specific information is not modelled at this level.

(4.6.1) INPUT AND OUTPUT

The two kinds of input and output hardware structures modelled are memory mapped I/O and I/O ports that are accessed with a separate address space. In both cases the port may be read only, write only or both, and the port will have an address.

(4.6.1.1) MEMORY MAPPED INPUT AND OUTPUT

The information to specify a memory mapped input or output port is specified by vertices attached to the MEMORY vertex which represents the memory module within which the memory mapped port appears. These vertices may have one of the reserved names

READ_PORT   WRITE_PORT   READ_WRITE_PORT

and will have an attached value which gives the address. This address is not relative to the processors addressing ranges, but

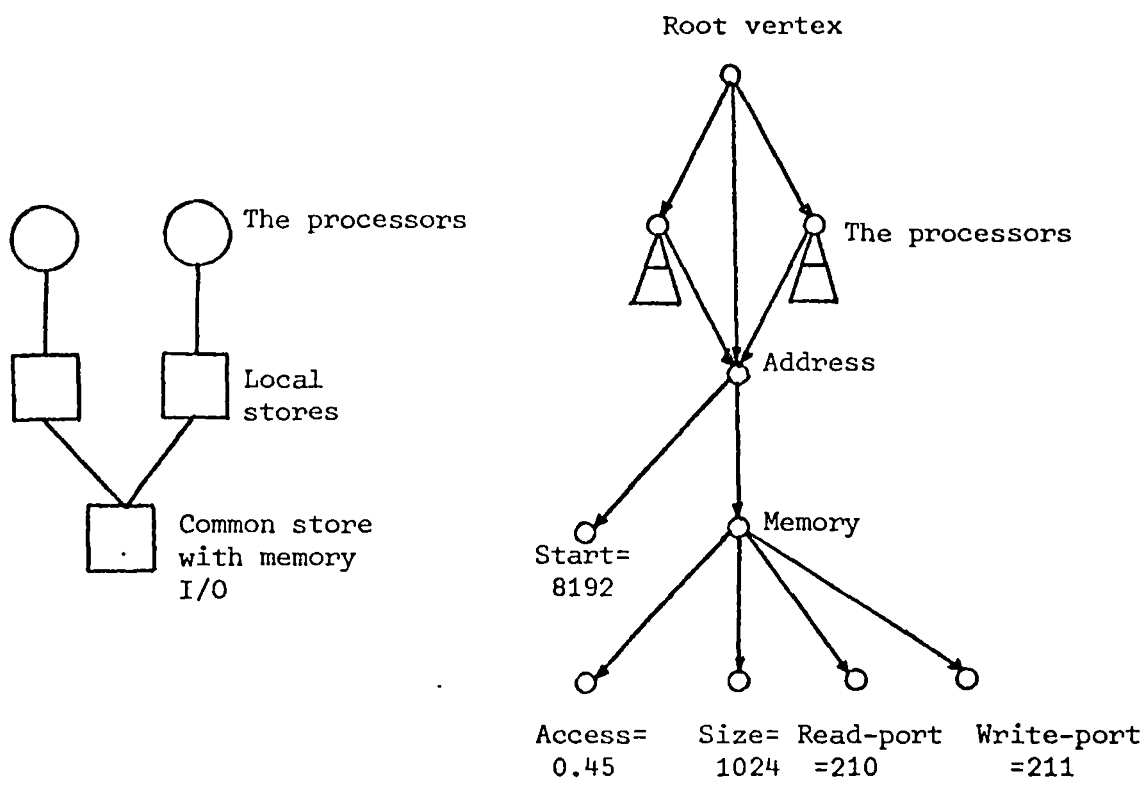


Figure 4.17

relative to the start of the memory module. Address 0 being the first location in the memory module. Thus to specify a memory mapped read port at location 210 in memory module STORE_1 requires

```
@.STORE_1.ADDRESS.MEMORY -> NEW ( READ_PORT = 210 ) ;
```

This directive attachs a new vertex of name READ_PORT and value 210 to the indicated memory vertex. Similarly to specify a write port requires

```
@.STORE_1.ADDRESS.MEMORY -> NEW ( WRITE_PORT = 211 ) ;
```

This information can be used by the allocator in the placement of variables from the user program which are to be used as input and output ports. The memory mapped input port is associated with the memory vertex information, and is equally accessible to any processor that can access the memory module.

An example of this structure is given for a computer architecture with two processors each accessing a common store. This store is part of a memory mapped I/O system and has a read port at 210 and a write port at 211. The specifications for this are

```
GRAPH
        VAR PSR  : SET ;
    PROCEDURE STANDARD_MEMORY ; ... etc as before ;
    PROCEDURE ONE_PROCESSOR ; ... etc as before ;
BEGIN
    FOR 2 DO
        ONE_PROCESSOR ( @ , PSR ) ;
        STANDARD_MEMORY ( PSR , 0 , 8192 ) ;
    END ;
    STANDARD_MEMORY ( @ , 8192 , 1025 ) ;
    @.PROCESSOR -> @.ADDRESS ;
    @.ADDRESS.MEMORY ->
    (  NEW ( READ_PORT = 210 ) , NEW ( WRITE_PORT = 211 ) ) ;
END ;
```

This specification is represented in figure(4.17).

(4.6.1.2) SEPARATE ADDRESS SPACE INPUT/OUTPUT.

In memory mapped input/output the information about the ports is attached to a memory. Separate address space refers to ports accessed directly by the processor, and in these specifications information is attached to a vertex given the reserved name PORT. Thus to specify a input port number 12 requires

```
@ -> NEW ( PORT ) -> NEW ( READ_PORT = 12 ) ;
```

The information about extra input/output ports may be attached to the same port vertex, or to different ones. Thus either of the following can be used

```
@ -> NEW ( PORT ) ->
(  NEW ( READ_PORT = 12 ) , NEW ( WRITE_PORT = 13 ) ) ;
```

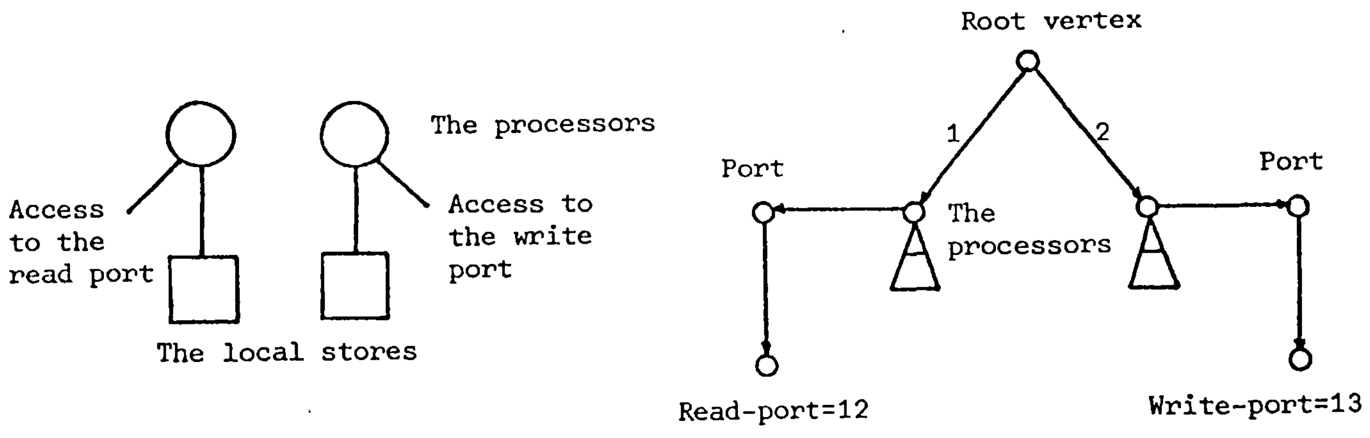


Figure 4.18

```
@ ->
( NEW ( PORT ) -> NEW ( READ_PORT = 12 ) ,
  NEW ( PORT ) -> NEW ( WRITE_PORT = 13 ) ) ;
```

to specify two ports, one input and the other output. The vertices are then attached to the processor vertex that represents the processor which accessed these input and output ports. Thus an example may be

```
GRAPH
   ... ;
BEGIN
   FOR 2 DO
      ONE_PROCESSOR ( @ , PSR ) ;
      STANDARD_MEMORY ( PSR , 0 , 1024 ) ;
   END ;
   @.PROCESSOR(1) -> NEW ( PORT ) -> NEW ( READ_PORT = 12 ) ;
   @.PROCESSOR(2) -> NEW ( PORT ) -> NEW ( WRITE_PORT = 13 ) ;
END ;
```

This gives a system of two separate processors, one accessing a read port and the other accessing a write port. The graph and computer architecture are represented in figure(4.18).

For the memory mapped I/O situation, two processors accessing the same memory containing a memory mapped port are regarded as both being capable of accessing this identical port. For separate address I/O ports, the meaning is somewhat different. Generally an I/O operation of this kind performed by one processor will access a different port from an I/O operation performed by another processor, even if the I/O port numbers are the same. To specify this, the usual occurrence, the same PORT vertices would not be attached to different processors. If, however, the ports of identical numbers on different processors are connected to the same hardware circuits so that they transmit and receive identical information, then this they can be regarded as one port shared by two processors. This is modelled in the graph structure by the same port vertex attached to more than one processor vertex. This is analogous to having a memory mapped information structure attached to more than one processor.

(4.6.2) INTERRUPTS

The resource allocator provides for the modelling of a user accessible interrupt structure. This takes the form of a hardware generated interrupt calling a user designated procedure. Providing the procedure code, and ensuring that the procedure returns before the next interrupt might be generated, is the users responsibility.

To specify the information about an interrupt requires a vertex with the reserved name INTERRUPT, and attached to it a constant value giving the address of the hardware interrupt. The binding between the interrupt and the high level language procedure it is to call is made by the linker stage.

Thus an example is

```
GRAPH
      ...
BEGIN
   FOR 2 DO
      ONE_PROCESSOR ( @ , PSR ) ;
      STANDARD_MEMORY ( PSR , 0 , 1024 ) ;
   END ;
```

```
        @.PROCESSOR(1) -> NEW ( INTERRUPT = 32 ) ;
    END ;
```

For an interrupt that jumps to address 32. If there is only one interrupt possible on a processor then the interrupt number may be left out. If the hardware that generates interrupts is also capable of producing an argument to go with the interrupt (as, for example, an interrupt made to a single address location which is given an interrupt number parameter), then this argument can be passed to the high level procedure via its parameter list.

These interrupt vertices are attached to the processor vertex representing the hardware processor that accesses this interrupt. A processor may have more than one interrupt. The converse is true also, the same interrupt may be attached to more than one processor. This is interpreted to mean that the hardware generating this interrupt sends the interrupt signals to the same interrupt addresses in all of these processors. Thus a program relying on this interrupt may reside upon any of these processors to work correctly. However the situation of identical interrupts being connected to differing interrupt addresses in different processors is not covered.

(4.6.3) SPECIFICATION OF VARIABLE ADDRESSES

Almost always the actual hardware addresses of variables will be assigned by the resource allocator. In the rare cases when the user requires to explicitly locate a variable this can be done by inserting into the information graph the required address and memory module. This information is then accessed by other constraint directives to bind the required variables to these addresses. As an example, to be able to place variables at address starting from 34 in memory A, requires

```
    S -> NEW ( USER_ADDRESS = 34 ) ;
```

where S is a reference set variable that contains the memory A vertex. Now to refer to this address the reference

```
    S.USER_ADDRESS
```

is used. More than one such vertex can be added, as in

```
S -> NEW ( USER_1 ) -> NEW ( USER_ADDRESS = 34 ) ;
S -> NEW ( USER_2 ) -> NEW ( USER_ADDRESS = 35 ) ;
```

Here the USER_1 and USER_2 are user defined vertex names, and the USER_ADDRESS vertex is the reserved name. Either address can now be referred to by

```
S.USER_1.USER_ADDRESS
S.USER_2.USER_ADDRESS
```

(4.7) CONDITIONALS

(4.7.1) USE OF CONDITIONAL SELECTION DIRECTIVES

The directives are used to specify the structure of the computer system. They are also used to examine this structure and select resource units which obey certain constraints. These constraints are supplied by the user when specifying how the elements of a program are to be allocated to the resources of the computer.

A user defined constraint indicates a program element or elements and lists the computer resources that the elements may be mapped to. An element is either a storage requirement of the program for code, data, stack or heap space, or it is a processing requirement for a process of the program. The resource this is mapped to can then be either a collection of physical memory modules onto which the program memory is allowed, or a collection of processors that are permitted to execute the process.

The selection of the elements and their specifications is described in chapter 5, as are the constraint directives themselves. Here the selection of the resource elements that are to be used for any one constraint directive is demonstrated. This involves selecting suitable processor or memory vertices from the entire system which satisfy the required conditions.

The selection directives are now described below in detail.

(4.7.2) SIMPLE SELECTION EXAMPLE

As an example the previous computer system defined with a two level map structure is used (given in section 4.4.2.3). This system is to be extended with the addition of disk I/O ports to some of the processors.

To specify the addition of a disk unit to one of the processors requires the definition of an additional vertex name, and a reference like

```
@.GROUP(4).PROCESSOR(2) -> NEW ( HAS_DISK = TRUE ) ;
```

This indicates that the 2nd processor of the 4th group has a disk attached. If there are a number of such directives, then to select any processors in this system that have a disk attached, without specifying the actual group and processor numbers, requires references like

```
@ -> NEW ( PROCESSOR_WITH_DISK ) ->
   @.GROUP.PROCESSOR.<VALUE( @.HAS_DISK)=TRUE> ;
```

which attachs all proccssor vertices which have a HAS_DISK vertex with a value true to the newly created PROCESSOR_WITH_DISK vertex. For the specification program to be legal this vertex name has to be defined in the vertex list of the specification block. In the examples following the need for this definition will not mentioned.

So now the processors that are in the system with a disk attached can be accessed directly with the reference

```
@.PROCESSORS_WITH_DISK.PROCESSOR
```

which will access all such processors. Any properties that these processors may have, for example their memory descriptions, can be referenced in the usual fashion starting from the above reference.

(4.7.3) MULTIPLE SELECTION CRITERION

More than one vertex can be specified in the selection criterion. For example some of the processors in the above system may be more reliable than others. This vertex can be specified by the reference

```
@.GROUP(1).PROCESSOR -> NEW ( IS_RELIABLE = TRUE ) ;
```

which will attach to all the processors in group 1 a unique IS_RELIABLE vertex of the indicated value.

Now to select all those processors with an I/O disk unit attached which are also reliable is achieved by the reference-

```
@.GROUP.PROCESSOR.< VALUE( @.HAS_DISK)=TRUE AND
                    VALUE( @.IS_RELIABLE)=TRUE > ;
```

Another example is where the disk units may have additional properties, such as disk access speed and storage size. These properties may be described and attached to the processor vertices by the following specifications

```
PROCEDURE DISK_UNIT (
   C : SET ;
   DISK_SPACE_VALUE : INTEGER ;
   DISK_SPEED_VALUE : REAL ) ;
BEGIN
   C -> NEW ( DISK ) ->
   (  NEW ( DISK_SPACE = DISK_SPACE_VALUE ) ,
      NEW ( DISK_SPEED = DISK_SPEED_VALUE ) ) ;
END ;
```

And this procedure is used to specify that some of the processors have disks,

```
DISK_UNIT ( GROUP(3).PROCESSOR(5) , 512 , 2 ) ;
DISK_UNIT ( GROUP(3).PROCESSOR(6) , 512 , 2 ) ;
DISK_UNIT ( GROUP(4).PROCESSOR(1) , 1024 , 1 ) ;
```

```
    DISK_UNIT ( GROUP(5).PROCESSOR(3) , 2048 , 4 ) ;
    DISK_UNIT ( GROUP(5).PROCESSOR(4) , 128 , 0.5 ) ;
```

Here the first number could be interpreted as the number of blocks in a disk and the second number is the number of milliseconds needed to access an average block. In this specification only 5 of the processors have disks attached.

With this specification a reference that will access all processors with disks attached can be

```
    @.GROUP.PROCESSOR. < NOT EMPTY ( @.DISK ) >
```

and to reference all processors with a disk attached of size 1024 blocks or bigger and an access time of 1ms or less requires

```
       @.GROUP.PROCESSOR.< NOT EMPTY ( @.DISK ) > .
          < VALUE( @.DISK.DISK_SPACE ) = 1024   AND
            VALUE( @.DISK.DISK_SPEED ) <= 1 >
```

This reference will still work when there are two or more disks attached to a processor vertex, the result will be any processor vertex with at least one disk attached that satisfies the constraint.

CHAPTER(5)

(5.1) USER CONSTRAINT SPECIFICATION

The user constraint directives specify constraints upon the placement of program elements onto the computer architecture. Generally the resource allocator will perform its task of generating a legal and efficient mapping without user intervention. However additional user constraints, based on application specific information, may be imposed by the programmer to guide the allocator.

There are several forms of constraints. They operate upon a collection of program elements and a collection of resource elements. The following sections will discuss

How to specify the resource elements of a constraint.

How to specify the program elements of a constraint.

How to specify the constraint action itself.

(5.2) SPECIFICATION OF RESOURCE ELEMENTS

Firstly the specification of the resource elements is outlined.

The computer architecture is specified by the ISL graph structure, and so the specification of resources uses this structure. Each resource of the computer is represented by a PROCESSOR, MEMORY, PORT or INTERRUPT vertex in the graph. Therefore a reference can be used to access these. Thus the resource references are a restricted form of the general ISL graph reference. They can be described by the syntax

```
Resource_Reference = "@",
   { ".", Vertex_Name, [ Vertex_Index ] } ;
Vertex_Name = Identifier ;
Vertex_Index = "(", Number, ")" ;
```

An example is

    @.GROUP.PROCESSOR(2)

(5.3) SPECIFICATION OF THE PROGRAM ELEMENTS

Now the specification of the program elements is given.

Program elements may be processes, variables, procedures and, depending upon the kind of computer architecture, modules. Only those elements which are named in the program are accessible to a constraint. This name is always the identifier that is given to the element in its definition or declaration. Such elements are specified by a path name constructed from information in the program text. There is one path name per element, and these are combined together into collections called objects. It is these objects that are used in a constraint directive.

(5.3.1) PATH NAMES

All program elements are accessed within the high level language program by their names, and so the specification uses these names also. However due to the scoping rules of many languages, these names may not be unique throughout the whole program. The technique adopted here is to construct a path name for each element. This path name consists of the name of the element, and the name of every enclosing scope. These are combined into one reference using the record dot notation (as used in Pascal).

The syntax for a path name is

```
Path_Name =
   { Scope_Name, "." }, Element_Name, [      Indication ] ;
Indication = ":","CODE"|"STACK" ;
Scope_Name = Identifier ;
Element_Name = Identifier ;
```

A path name like

Element_Name1

refers to an element in the program which is in the outer most scope. A path name like

Scope_Name1.Element_Name1

refers to an element named Element_Name1 which is defined in a scope called Scope_Name1.

Thus to refer to the integer variable in the following Pascal code

```
PROCEDURE DEMONSTRATE ;
VAR I : INTEGER ;
   PROCEDURE NUMBER_ONE ;
   VAR I : INTEGER ;
   BEGIN ... END ;
BEGIN ... END ;
```

requires one of the following two references

```
DEMONSTRATE.I          DEMONSTRATE.NUMBER_ONE.I
```

depending on whether or not the first or the second occurrence of the I integer is wanted.

If this is a recursive procedure, then the reference will refer to all instances of the variable. Consequently the variable will occupy the same address for all invocations. The other local variable of the procedure will be allocated as normally expected.

(5.3.2) WITH BLOCKS

Since these path names can rapidly become very long with deeply nested programs, a WITH block is allowed in the specification directives.This encloses a syntactically complete collection of resource allocator specifications and allows the first part of a long path name to be specified.

```
With_Block = "WITH", Path_Name, "DO",
        { Object_Assignment | With_Block }, "END", ";" ;
( Object_Assignment is to be defined, contains Path_Names ).
```

A With_Block like

```
WITH Path_Name1 DO
   ..... Path_Name2 .....
END WITH ;
```

is equivalent to

```
..... Path_Name1.Path_Name2 .....
```

if this path name is able to refer to an element. If, however, this path name does not refer to any elements in the program, then this is equivalent to

```
..... Path_Name2 .....
```

in other words the path name inside the With_Block is used without alteration.

As an example

```
PROGRAM DEMONSTRATION ;
   PROCEDURE NUMBER_ONE ;
      PROCEDURE NUMBER_TWO ;
         PROCEDURE NUMBER_THREE ;
         VAR
            A , B , C , D , E : DATA_TYPE ;
         BEGIN . . . END ;
      BEGIN . . . END ;
   BEGIN . . . END ;
BEGIN . . . END .
```

Here the variables A to E may be referred to inside a resource allocator specification by the following

```
WITH DEMONSTRATION.NUMBER_ONE.NUMBER_TWO.NUMBER_THREE DO
   . . .
   use the identifiers A, B, C, D or E.
END ;
```

(5.3.3) PROCESS CODE AND STACK ELEMENTS

Consider a path name to a process called P in some program. This will be like S1.S2. ... P, where S1, S2... are the surrounding scope names. Now is this path name a memory element referring to the code of the process, or is it a process element referring to the processor that executes the process? To resolve this, a path name like

```
S1.S2. ... P
```

will always refer to a process element by default. If it is required to indicate the code of the process, then the path name is

```
S1.S2. ... P:CODE·
```

Variables never contain executable code, and so this type of specification is not needed when using path names to refer to a variable. Similarly a procedure or module element. is always a memory element, since the execution of the code of a procedure or module is performed by a process.

To refer to all of the local variables of a process the following is used,

```
S1.S2. ... P:STACK
```

(5.3.4) OBJECT DEFINITIONS AND ASSIGNMENTS

A Path_Name is used to refer to a particular program element. The resource allocator generally uses collections of elements when it is performing its allocation. A collection of program elements is called an object. Objects are defined in the resource allocation language, and are used in the user constraints. Object_Assignments are used to specify which elements these objects refer to. An object can contain many program elements, but each object can contain process elements only, or memory elements only.

Program elements may only be assigned to objects that have already been defined. An object definition consists of the name of the object and its kind, either a process object or a memory object. The syntax is

```
Object_Definition_Part =
   "DEFINITION", { Object_Definition }, "END", ";" ;
Object_Definition =
   Object_Name, { ",", Object_Name }, ":", Object_Kind ;
Object_Name = Identifier ;
Object_Kind = "PROCESS" | "MEMORY" ;
```

For example

```
DEFINITION
   OBJECT_1 : PROCESS ;
   OBJECT_2 , OBJECT_3 : MEMORY ;
END ;
```

will create three objects, the first object will contain processes and the next two will contain program memory elements.

After the Object_Definition_Part appears the Object_Specification_Block. This contains Object_Assignments which specify the elements that each object is to refer to. An object may appear in only one Object_Assignment, and program elements can also only appear in one Object_Assignment. The syntax for this is

```
Object_Specification_Block = "SPECIFICATION"
   { Object_Specification }, "END", ";" ;
Object_Specification =
   Object_Name, ":=", "[", Program_Path_List, "]", ";" ;
Program_Path_List = Path_Name, { ",", Path_Name } ;
```

Thus

```
OBJECT_1 := [ DEMONSTRATION.PROCESS_A ] ;
```

will result in OBJECT_1 referring to the indicated process in the program,

```
    OBJECT_2 := [ DEMONSTRATION.A , DEMONSTRATION.B ] ;
    OBJECT_3 := [ DEMONSTRATION.PROCEDURE_ONE:CODE ] ;
    OBJECT_4 := [DEMONSTRATE.PROCESS_A:STACK];
```

and the above two assignments will result in OBJECT_2 referring to the variables A and B and in OBJECT_3 in referring to the code of the procedure.

(5.3.5) PROGRAM SPECIFICATION BLOCK

The Object definition and specifications appear in a complete Object_Specification block, whose syntax is

```
    Object_Specification_Block =
       "OBJECT",
          Object_Definition_Part,
          Object_Specification_Block,
       "END", ";" ;
```

A complete example, bringing together the separate examples of the last section, is

```
    OBJECT
       DEFINITION
          OBJECT_1 : PROCESS ;
          OBJECT_2, OBJECT_3 : MEMORY ;
       END ;
       SPECIFICATION
          OBJECT_1 := [ DEMONSTRATION.PROCESS_A ] ;
          OBJECT_2 := [ DEMONSTRATION.A , DEMONSTRATION.B ] ;
          OBJECT_3 := [ DEMONSTRATION.PROCEDURE_ONE:CODE ] ;
          OBJECT_4 := [DEMONSTRATE.PROCESS_A:STACK];
       END ;
    END ; .
```

(5.4) CONSTRAINT SPECIFICATION

Finally the constraints themselves are described. There are two main types of user constraints. These are

A) General constraints. These specify to which group of resource elements a program element may be assigned.

B) Address constraints. These allocate variables and interrupt calls to specific addresses within physical memory modules.

The syntax and usage of these are described in the following.

(5.4.1) GENERAL CONSTRAINTS

A general constraint will specify the processors of the architecture to which the given process elements of the program can be assigned, or it will specify the physical memory resources to which the given program memory elements may be assigned. There are two kinds of general constraints, which are

A) Assignment constraints. These specify a list of resource elements to which the indicated program elements may be assigned.

B) Proximity constraints. These impose constraints upon the placement of program elements depending on the locations of other already placed program elements. The two types of proximity constraints may be to either place the program elements onto the same resources as some other program elements, or to place them onto different resources from some other program elements.

Firstly the assignment constraint are described, followed by the proximity constraints.

(5.4.1.1) ASSIGNMENT CONSTRAINTS

An assignment constraint has the syntax

```
Assign_Constraint =
    "ASSIGN", Object_List, "->", Resource_List, ";" ;

Object_List = "(", Object_Name, { ",", Object_Name }, ")" ;
Resource_List =
   "(", Resource_Reference, { ",", Resource_Reference }, ")" ;
Object_Name = Identifier ;
```

An assignment statement like

```
ASSIGN ( Object_Name1 ) ->
   ( Resource_Reference1, ... Resource_ReferenceN ) ;
```

will indicate to the resource allocator that all the program elements specified by the Object_Name1 object will be assigned only to some resource element which is a member of the resource list. The types of the program elements in the object must agree with the types of the resources in the resource list. That is program elements that are processes can only be assigned to resource elements that are processors. Similarly program memory elements are only assigned to physical memory resources.

An assignment like

```
ASSIGN ( Object_Name1, ... Object_NameN ) -> Resource_List1 ;
```

is equivalent to the separate assignments

```
ASSIGN ( Object_Name1 ) -> Resource_List1 ;
   ...
ASSIGN ( Object_NameN ) -> Resource_List1 ;
```

while two assignments like

```
ASSIGN ( an object list containing program element X )
   -> Resource_List1 ;
ASSIGN ( an object list containing program element X )
   -> Resource_List2 ;
```

will specify two constraints upon the program element X. In this case the intersection of the resource list sets gives the constraint for X. Thus the constraint on X is

```
ASSIGN ( an object containing only X ) ->
   a resource set equal to Resource_List1 AND Resource_List2 ;
```

Conflicting constraints can be detected at this stage if the combined resource of an element becomes empty. This indicates that the element can not be assigned to any resource without violating one or another of the constraints imposed upon it.

(5.4.1.2) PROXIMITY CONSTRAINTS

Program elements may also be constrained to locations depending upon the proximity of the assignment of other program elements. There are only two degrees of proximity allowed, either a program element may be assigned to the same resource as some other program element, or it may be assigned to a different resource. The syntax for these are

```
Proximity_Constraint =
   "ASSIGN", Object_List, "->",
   ( "SAME" | "DIFFERENT" ), Proximity_Resource_List, ";" ;
Proximity_Resource_List =
   "(", Proximity_Resource_Reference,
   { ",", Proximity_Resource_Reference } ;
Proximity_Resource_Reference = "@",
   { ".", Vertex_Name, [ "(", ( Number | "*" ), ")" ] } ;
```

A proximity constraint like

```
ASSIGN Object_List1 ->
   SAME ( Proximity_Resource_Reference1 ,
                  ...
             Proximity_Resource_ReferenceN ) ;
```

will ensure that each program element in the object list will go to the same group of resource elements from amongst Proximity_Resource_Reference1 to Proximity_Resource_ReferenceN. Alternatively a proximity constraint like

```
ASSIGN Object_List1 -> 
   DIFFERENT ( Proximity_Resource_Reference1 ,
                  ...
               Proximity_Resource_ReferenceN ) ;
```

will ensure that each program element in the object list will each be allocated to a different group of resource elements.

The references used may be similar to those used in the assignment constraints. Alternatively the references may also have a "*" character in the vertex indices. In a reference like

```
Vertex_Name1. ... Vertex_NameX(*). ... Vertex_NameN
```

the "*" character in the index reference is used to represent all possible index values. If this vertex has a possible index range of 1..5 then the reference above is equivalent to the references

```
Vertex_Name1. ... Vertex_NameX(1). ... Vertex_NameN ,
Vertex_Name1. ... Vertex_NameX(2). ... Vertex_NameN ,
Vertex_Name1. ... Vertex_NameX(3). ... Vertex_NameN ,
Vertex_Name1. ... Vertex_NameX(4). ... Vertex_NameN ,
Vertex_Name1. ... Vertex_NameX(5). ... Vertex_NameN
```

and thus this notation is a shorthand method of writing out the resource reference lists.

An example of a proximity constraint is

```
ASSIGN Object_List1 ->
   SAME ( GROUP(1).PROCESSOR , GROUP(2).PROCESSOR ) ;
```

(using the graph definition of section(4.4.2.3)). This will ensure that, however the resource allocator performs is allocation, all of the processes in the Object_List1 will always end up in either the processors of GROUP(1) or the processors of GROUP(2). Another example is

```
ASSIGN Object_List1 ->
   DIFFERENT ( GROUP(*).PROCESSOR(*) ) ;
```

This is equivalent to the expanded constraint

```
ASSIGN Object_List1 ->
   DIFFERENT ( GROUP(1).PROCESSOR(1) , GROUP(1).PROCESSOR(2) ,
               GROUP(1).PROCESSOR(3) , GROUP(1).PROCESSOR(4) ,
               GROUP(2).PROCESSOR(1) , GROUP(2).PROCESSOR(2) ,
                  ...
               GROUP(4).PROCESSOR(3) , GROUP(4).PROCESSOR(4 ) ) ;
```

assuming 4 groups and 4 processors per group. Thus this constraint will ensure that each process in the object list will end up in a processor by itself. Note that if there are more processes in the object list than there are processors in the resource list, then the constraint can not be satisfied for all processes simultaneously, and so the resource allocation mapping will fail.

(5.4.2) ADDRESS CONSTRAINTS

The constraint directives described above are applicable to the control of the over all allocation strategy of the resource allocation by the user. The location directives now described are used to specify explicitly the interrupt addresses for procedure calls and the addresses of normal variables and I/O variables.

(5.4.2.1) I/O VARIABLE ADDRESSES

To access memory mapped input/output information a variable of the correct size can be positioned at the memory mapped I/O address. This variable can be used within the program exactly like any other variable. The only difference, from a high level language point of view, is if this variable is in a local procedure declaration space. In this case each activation of the procedure will access that same variable address, instead of having a new variable created on the procedure invocation stack each time. From the point of view of the hardware, the variable address corresponds to a memory mapped I/O port.

The syntax is

```
Location_Constraint = "LOCATE", "(", Variable_Path_Name, ")",
   "->", "(", Resource_Reference, ")", ";" ;
Variable_Path_Name = Path_Name ;
```

and an example is

```
LOCATE ( Variable_Path_Name1 ) -> ( Resource_Reference1 ) ;
```

Here the variable referenced by the variable path name will be allocated to the address given for the input/output port specified by the resource reference in the information graph. A variable path name is used instead of an program element reference, since usually the address of only one variable at a time needs to be set. A variable can only be assigned to one location and so the resource element has to refer to one store module only.

(5.4.2.2) INTERRUPT CALLS.

Interrupts that are to be accessed explicitly by the programmer are implemented as external calls to a user written procedure. This procedure is written according to the usual high level language conventions. An interrupt call to it is equivalent, at the programming language level, to a call from an anonymous process written in the language.

In the following the resource interrupt reference refers to a memory address specification in the information graph. The syntax to indicate the binding between the procedure in the program and the interrupt call is

```
Interrupt_Constraint -> "INTERRUPT", "(", Procedure_Path_Name,
   ")", "->", "(", Resource_Reference, ")", ";" ;
Procedure_Path_Name = Path_Name ;
```

and so an interrupt constraint is like

```
INTERRUPT ( Procedure_Path_Name1 ) -> ( Resource_Reference1 ) ;
```

where the procedure specified by the path name will be called whenever there is the appropriate interrupt to the processor.

(5.4.3) MULTIPLE CONSTRAINTS

If a program element appears in more than one constraint, then the final allocation for that element must satisfy all such constraints simultaneously. For example the constraints

```
ASSIGN Object_List1 -> ( Reference1 ) ;
ASSIGN Object_List2 -> ( Reference2 ) ;
ASSIGN Object_List3 -> SAME ( Reference3 ) ;
ASSIGN Object_List4 -> DIFFERENT ( Reference4 ) ;
```

will ensure that if program element X is in all four object lists, then the assignment of X to the architecture architecture will be such that

X is assigned to a resource in Reference1.
X is assigned to a resource in Reference2.
X is assigned to the same resource in Reference3 as all the
   other program elements in Object_List3.
X is assigned to a different resource in Reference4 from all
   the other process elements of Object_List4.

If such a resource does not exist, then the map allocation will fail.

(5.5) FINAL SYNTAX

The syntax for the entire allocation specification program is

```
Allocation_Program =
   "ALLOCATION",
      Graph_Specification_Block,
      Object_Specification_Block,
      Constraint_Block,
   "END", "." ;

Constraint_Block =
   "CONSTRAINT", { Constraint }, "END", ";" ;
Constraint = Assign_Constraint | Location_Constraint |
             Interrupt_Constraint | Proximity_Constraint ;
```

A complete example using the specification language is given in Appendix(F).

CHAPTER (6)

(6.1) THE CALCULATION OF THROUGHPUT

An allocation program needs to be able to produce efficient mappings of programs onto computer architectures. An efficient implementation of a program can include many factors, such as using the minimum memory space, executing in the fastest time or having maximum reliability. When an allocation program is used it is presented with an already written program and a fixed architecture. The most important efficiency measure it can influence is the execution speed of the program. Decreasing the memory usage is outside its capabilities, because this depends upon the design of the program. Increasing reliability, by placing important data and processes onto reliable memory modules or processors, is not directly carried out by the allocation program. Instead the user imposes these requirements with the aid of constraints. Thus the sole efficiency measure that can be optimized by the allocation program is the execution time or throughput of the final allocation. Consequently it needs to be able to obtain an estimation of this throughput for any allocation mapping.

This execution time estimation may be produced in two different ways, either by solving an analytic probability model or by running a simulation program. For this thesis an analytic model was derived from work by [44]. The model described will calculate the general memory interference in a multiprocessor computer involving bus conflicts and bus induced delays. The results obtained from this model were tested by using a simulation model, a brief description of this model is also given. This chapter finishes with a discussion on the relative performance of both the simulation and the analytic model solutions.

(6.2) ANALYTIC PROBABILISTIC THROUGHPUT MODEL

In the following an analytic probabilistic model is described which can be used to calculate the throughput of a concurrent program to be executed on a multiprocessor system. The basic mathematical model comes from [44] which takes into account the effects of memory interference.

To this has been added extensions to allow for different store cycle times and to include the effects of common store access buses.

The original model assumed that each processor was running an independent program. This implies that a processor only idles when it is waiting on a busy memory. This idling will occur on a cycle by cycle basis. However for the allocator problem this is no longer true, the processors execute code that is part of the single program. Thus the processors may spend some of the time idling, not because of memory contention, but because they are waiting on semaphore locks until some useful work becomes available. This kind of idling will occur over a much longer time scale than the first type. Accordingly the model has also been adapted to this requirement.

(6.2.1) MODEL DESCRIPTION

The model assumes a multiprocessor computer containing a number of processors and store modules. In the model any processor may access any store, although some access paths between processors and store modules may not utilized by the actual computer architecture hardware, and some access paths may go through common store access buses. The store modules and processors may have different access times and processor cycle times. The common buses may introduce access time overheads.

Each store has an access time followed by a recovery time. The access time is the time required to fetch or store one memory value. The recovery time is the time required by the memory to become ready for the next request. During this time the processor is released and may do useful work. Generally only older magnetic core memory technology will have nonzero recovery times. In the model there are M stores, and the stores are referenced by the index S.

Each processor has an average single instruction processing time during which it does not access the memory. This is followed by a memory fetch cycle, in which the processor idles until the memory request has been completed. It is assumed that the single instruction time is greater than the store recovery time. Thus a processor does not issue a memory request before the store has recovered from the last one. The model assumes that there are N processors, the index P is used to refer to a particular processor.

A processor may not be able to directly access a store, but has to access it through an intermediate common bus in competition with other processors. Such a bus will introduce an access overhead which is its bus delay. In the model the buses are referenced by a bus index called B .

The model is supplied with an array which gives a value for each processor store pair. This value is the number of memory accesses the processor makes to the store in an arbitrary time. For some pairs this will be zero, indicating the processor never accesses that particular store. This array, when normalized, will give the probability access pattern of the processors. The array is represented by

$$Ni\ (P,S)$$

where Ni is the input number of cycles, P is the processor number and S is the store number. It can be normalized by a constant factor C such that

$$C * \sum_{S=1}^{M} Ni\ (P,S) \leq 1 \qquad \ldots(1)$$

where for some P the summation equals one. As an example, an input number of cycles array could be

	stores	
processors	4	6
	3	5

For this the normalization factor is 0.1, giving the normalized array

	stores		row summation (per processor)
processors	0.4	0.6	1.0
	0.3	0.5	0.8

This array gives the access probability pattern of the processors. Thus the first processor spends 0.4 of its time accessing the first store, and 0.6 of its time accessing the second store. The second processor spends 0.3 of its time accessing the first store, 0.5 of its time accessing the second store and the remaining 0.2 of its time idling.

As its solution the model will produce an actual number of cycles array. This will give the calculated number of accesses between each processor and store in a unit time. The pattern of accesses will be the same as for the input number of cycles array, and so the two arrays will differ only by a multiplicative constant. This constant is used as the throughput. It represents the number of times per unit time period that the computer architecture can execute the given input number of cycles information. It is expressed as

$$Ni*Tp = Na \qquad ...(2)$$

where Tp is the throughput and Na is the actual number of cycles array. As an example, the actual number of cycles array may be

	stores	
processors	440	660
	330	550

This differs by a factor of 110 from the input number of cycles array. Thus the computer can execute the 4 accesses between the first processor and the first store 110 times a second. The throughput for this is therefore 110.

As a final note, the probabilities used in the model are concerned with the probability that some action will be proceeding in a given time period. This probability will be equivalent to the fraction of the time that the action is proceeding. If a unit time period is used, then this fraction of time will equal the actual time spent in the activity. Accordingly in the derivation either the probability or the time interpretation is used. This depends upon which is the most convenient.

(6.2.2) SIMPLIFICATIONS IN THE MODEL

In an executing processor, the address sequences will not be random but will display some serial correlation. This is especially the case for instruction fetching, where the addresses will be predominantly consecutive. [44] demonstrates that this effect is not important in most circumstances. Consequently in this model it is assumed that the throughput obtained from a random distribution of addresses will be a good approximation to the throughput obtained if the effects of address serial correlation were taken into account.

Another inaccuracy is due to the probability methods used, which assume that all time periods are infinitely divisible. However in the actual hardware the time over which the memory is actually accessed, or a processor executes a single cycle, comes in discrete time units of one processor or store cycle. The effect of this simplification is only noticeable over a large time period in special circumstances. One case, for example, is with two processors accessing a common store. Each processor has a one microsecond instruction execution time, and the store also has a one microsecond access time. In this situation, after a possible initial clash, the two processors will execute in lock step. They will alternate in using the store and executing an instruction. Thus there will be no conflict, even though the model predicts a degradation in the throughput of 12.5 percent compared to the actual throughput obtainable from the system. This difference becomes less when there are a larger number of processors and stores, and when the instruction execution times are not constant.

As well, there is an inaccuracy not present in Hoogendoorns original work. There the processors are assumed to be executing their instruction streams independently of each other. Thus the probability model assumes the processors are statistically independent. This is no longer true when the processors contain processes which communicate to each other. Thus two processors may be specified as having a 1000 memory fetches each to the same store, which by the model will cause execution time degradation via memory contention. However in actual practice the processors may be executing in turns, communicating between themselves via semaphores as to which processor is to execute next. In this situation the observable throughput will not be as predicted by a straight memory interference model.

Assume that the same processors and store are as used in the preceding example. When the processes on the processors execute independently, the processors will be in lockstep and the store will be occupied 100 percent of the time. If the processors operate dependently with turns of 1000 cycles each, then the store will be occupied only 50 percent of the time. If the model makes no allowances for processor dependencies, then it will assume simultaneous execution. Thus its estimation throughput will be twice as large as the actual throughput obtained when the processors execute in turns.

Notice, however, that for this to occur requires a program making no use at all of the parallelism possible with two processors. Most programs will have greater parallelism than this between their processes, and so there will greater overlap in the execution of different processors. Programs with large numbers of semaphores, executing on architectures with more than two processors and more than one store, will show less of this effect. Therefore this behaviour is not taken into account in the throughput models, it is assumed that the straight probability model will provide a sufficiently adequate throughput measure for the allocators purposes.

This leads to the final assumption made in this model. A set of processes cooperting in a single program synchronized by semaphores will have the overall rates of progress of the individual processes fixed by the application. This overall rate is used to define the throughput of the program. It is assumed that from this actual number of cycles array can be derived by the application of a single multiplicative factor, and that this has relevance on a cycle by cycle basis. This assumption is represented in equation 2.

In general the cycle by cycle behaviour of the program will not reflect this, since each processor will execute at full speed until it reaches a synchronization event, then block. The time period over which this occurs contains many processor cycles. The model and simulator both make use of this assumption, therefore the results from these can only be approximate. However note that any real program can show considerable variances in its execution time performance due to the dynamic nature of its environment, thus any estimate of the throughput will always be an approximate anyway.

(6.3) DERIVATION OF THE CONFLICT FUNCTION

Calculating the throughput of a concurrent program requires a means of working out the effects of processor access conflicts. In this section a general conflict model is derived.

The general conflict model assumes a number of users requesting service from a number of common resources. To develop this model the simple case of a number of processors accessing a number of stores is used for illustration. Each processor spends a certain proportion of its time accessing the store. This is called the combined probability and comprises the fraction of time that the processor waits while the store is busy servicing requests from other processors, plus the actual store access time. This last is the probability that the processor is actually accessing the store successfully. From this it can be assumed that

$$Pa(P,S) = Pc(P,S) * Cf(P,S) \qquad \ldots(3)$$

Here the probability array Pa represents the probability of processor P accessing store S successfully in a unit time period. The combined probability array Pc represents the probability of processor P accessing the store S or attempting to access the store S in unit time. The conflict function Cf is some value with a lower bound tending towards 0 and an upper bound of 1. This function can be regarded as representing the fraction of the total combined probability that any store request from a processor is actually able to successfully access the store.

If there are no competing processors, this function is equal to 1. If there are other processors, then this function is dependent upon the time spent by these other processors in attempting to also access the same store. Thus with N processors,

Pa(P,S) =

Combined probability of processor P accessing the store S (successfully of not) *
Combined probability that no other processors are accessing the store S (successfully or not).

+ 1/2 Combined probability of processor P accessing the store S (successfully or not) *
The combined probability that one other processor is accessing the store S (successfully or not).

+ 1/3 Combined probability of processor P accessing the store S
(successfully or not) *
The combined probability that two other processors are
accessing the store S (successfully or not).

+ ...

+ 1/N Combined probability of processor P accessing the store S
(successfully or not) *
The combined probability that all other N-1 processors are
accessing the store S (successfully or not).

...(4)

The first term gives that part of the probability when the processor is the only processor accessing the store. The second term gives the probability when one other processor is accessing the store. Since only one request is allowed at a time, and it is assumed that the store chooses new requests in an unbiased way, then either processor P or the other processor is randomly choosen 1/2 of the time. Thus this term has a 1/2 in front of it. The terms continue in this fashion until the last term, where the processor has 1/Nth of a chance of accessing the store when processor P and all other N-1 processors are simultaneously attempting to access it.

Expansion of this function gives

Pa(P,S) =

Combined Probability of processor P accessing the store
S (successfully or not) *
(Combined probability that no other processors are
accessing store S (successfully or not)
+1/2 Combined probability that one other processor is
accessing store S (successfully or not)
+1/3 Combined probability that two other processors are
accessing store S (successfully or not)
...
+1/N Combined probability that all N-1 other processors are
accessing store S (successfully or not))

...(5)

Therefore when this equation is compared with equation(3) it can be seen that the part in the brackets is the conflict function. Thus the conflict function can be written as

$$Cf(P,S) = \sum_{K=1}^{N} Pt(K,P,S)/K$$

...(6)

The probability term functions Pt inside the brackets represent the probability that K-1 processors are accessing the store (successfully or not) out of a total of N-1 processors. This probability is given by

$$Pt(k,P,S) = \sum_{r=1}^{rmax} \prod_{\substack{P'=1 \\ P'\neq P}}^{n} Fkrn(P') \left[\begin{array}{ll} = 0 & Pc(P',S) \\ \\ \neq 0 & 1 - Pc(P',S) \end{array} \right.$$

...(7)

The function Fkrn represents the sequence of all permutation lists of N-1 elements, each element being either 0 or 1. There are K-1 zeroes each permutation list, and R gives the permutation index number, for some given ordering of the permutation lists. The value Fkrn(P') gives the P' element in a permutation list, where each permutation list is of the form

Fkrn = (Fkrn(1), Fkrn(2), ... Fkrn(P-1), Fkrn(P+1), ... Fkrn(N))

...(8)

There is no element corresponding to the Pth index. Thus this function produces the probability of K-1 actively accessing processors out of N-1 processors in total. Using equations 6 and 7 , the full expansion of the conflict function is now

$$Cf(P,S) = \sum_{k=1}^{n} \frac{1}{k} \sum_{r=1}^{rmax} \prod_{\substack{P'=1 \\ P' \neq P}}^{n} Fkrn(P') \left[\begin{array}{l} = 0 \;\; Pc(P',S) \\ \\ \neq 0 \;\; 1 - Pc(P',S) \end{array} \right.$$

...(9)

As an example, the conflict function for the first processor of a system of three processors is

$$\begin{aligned} Cf(1,S) = {} & (1-Pc(2,S)) * (1-Pc(3,S) + \\ & \tfrac{1}{2} ((1-Pc(2,S)) * Pc(3,S) + (1-Pc(3,S)) * Pc(2,S)) + \\ & \tfrac{1}{3} (Pc\ (2,S) * Pc(3,S)) \end{aligned}$$

and similarly for the conflict functions for processors 2 and 3. The implementation algorithm used to derive a conflict function is described in appendix(B).

$$x \left[\begin{array}{l} = 0 \;\; a \\ \neq 0 \;\; b \end{array} \right.$$

Note, this means to use the value of a if the expression X has a value of 0, and to use the value b if the expression X is not equal to 0.

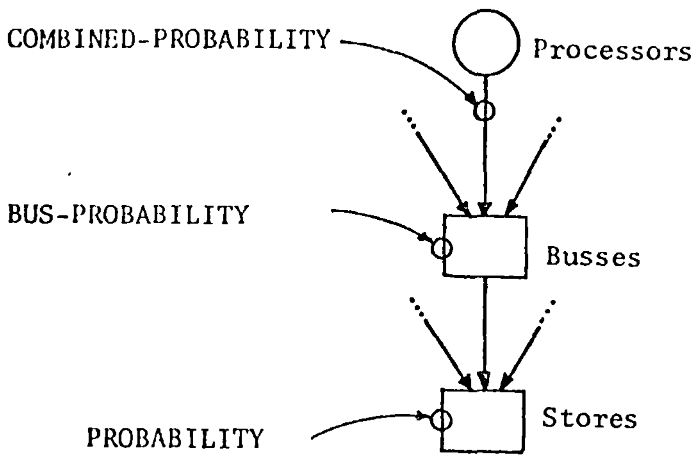


Figure 6.1

(6.4) DERIVING THE PROBABILISTIC EQUATIONS

The computer architecture to be modelled has a number of processors accessing a number of store blocks, both directly and via common store buses. In the following the full memory interference model for such a system is developed, using the conflict function derived above for the simple case.

Most of the time that a processor spends in attempting to access a store will be spent in waiting because other processors are blocking access. This conflict occurs at two places,

A) at the bus level where the processor is competing with the other processors to access the necessary bus,

B) and at the store level, where the processor now in control of the bus has to compete with the other buses to access the actual store.

This is represented in figure(6.1).

In view of this structure the mathematical model for such a system is developed in steps. It starts by deriving the amount of time that is spent by the processor in accessing store and from this is derived the amount of time wasted in waiting for the bus to become free, and the amount of time wasted while the store is occupied by other users. Finally some refinements are added.

The conflict for bus function Cb gives the conflict factor due to the interference of all the other processors accessing the same bus. If the processor has direct access to the store without any intervening buses then this factor is one.

$$Pb\ (P,S) = Pc(P,S) * Cb(P,S)$$

...(10)

The CONFLICT_FOR_BUS function gives the conflict factor due to the interference of all the other processors accessing the same bus. If the processor has direct access to the store without any intervening buses then this factor is one.

To derive the value of this conflict for bus function all of the other processors accessing the same bus are examined. The processor P is disregarded, since this is the processor to which the conflict function is applied to. Each of the other processors will access one or more stores through the bus. For any one processor the time spent in accessing the bus will be the summation of the total combined probability spent in accessing each of these stores through this bus. These bus probability terms are then used to generate the conflict function value. Thus

$$Cb(P,S) = \sum_{K=1}^{n} \frac{1}{k} \sum_{r=1}^{rmax} \prod_{\substack{P'=1 \\ P' \neq P}}^{n} Fkrn(P') \left[\begin{array}{ll} = 0 & Pbt(P',Bn(P,S)) \\ \neq 0 & 1 - Pbt(P',Bn(P,S)) \end{array} \right.$$

...(11)

where the bus probability term Pbt will be

$$Pbt(P',B) = \sum_{\substack{\text{All } S', \text{ where} \\ B=Bn(P',S')}} Pc(P',S')$$

...(12)

and where the bus number function Bn returns the index of the bus that the processor is to use to access the indicated store.

The resulting value is the time spent successfully accessing the bus. Now the time spend successfully accessing the store is

$$Pa(P,S) = Pb(P,S) * Cs(P,S)$$

...(13)

The conflict for store function Cs represents the conflict produced by all of the other buses that access the store. The total amount of time spent by any one of these other buses in accessing the store is given by the summation of the times each processor using it spends in successfully controlling the bus to access the store. This value from each bus is added together to give the conflict function value.

$$Cs(P,S) = \sum_{k=1}^{n} \frac{1}{k} \sum_{r}^{rmax} \prod_{\substack{B'=1 \\ B'\neq Bn(P,S)}}^{n} \quad Fkrn(B') \begin{cases} = 0 & Cbt(B',S) \\ \neq 0 & 1-Cbt(B',S) \end{cases}$$

...(14)

and the bus conflict terms Cbt are

$$Cbt(B',S) = \sum_{\substack{\text{All } P', \text{ where} \\ Bn(P',S)=B'}} Pb(P',S)$$

...(15)

This gives the probability equation for a system with processors competing for buses and stores. It has been derived assuming zero bus delay times. To include these, assume that the bus delay is modelled as an extra amount of time that a processor has to spend in the bus on top of the delays introduced by bus conflicts. Thus the equations above are modified to include this time by subtracting the time spent in the bus delay itself from the bus probability. Thus the new equation is

$$Pb(P,s) = Pc(P,S) * Cb(P,S) - Tbd(Bn(P,S)) * Na(P,S) \qquad \ldots(16)$$

where the actual number of cycles array Na(P,S) is the number of accesses that the processor P makes to store S in unit time, and the bus delay Tbd is the amount of delay introduced by the bus. Therefore the bus probability time is now the time spent in successfully controlling the bus and being able to actually request a store.

The final addition to this model for such a system is to include the circumstance where the store has a finite recovery time during which the processor is free to continue its processing but the store is still unavailable. This refinement is only required for older magnetic core stores which have a rewrite time, but is included to be in line with the original model of Hoogendoorns. In the original this time is modelled as if the processor was still in control of the store for this rewriting time. Thus the store cycle time is taken to be the store access time plus the store rewrite time and the cycle time of the processor is adjusted to be the processor cycle time minus the store rewrite time. In the current model this approach acts as if the processor is accessing the store, and thus holding the bus, for the access time plus the rewrite time. But in the actual hardware the time the processor is successfully accessing the store is the store access time, and the bus is only held for this amount of time. Consequently, to adjust the model for the provision of a store rewrite time requires subtracting the rewrite time from the total access time before the calculation of the bus access time, and then adding it back again later. Thus outside of the bus the model is as in the original. Inside the bus the bus conflicts are calculated only in terms of the actual time the bus is held.

Thus the final equation for the bus probability time is now

$$Pb(P,S) = (Pc(P,S) - Sr(S) * Na(P,S)) * Cb(P,S) + Tsr(S) * Na(P,S) - Tbd(Bn(P,S)) * Na(P,S)$$

...(17)

where Tsr is the store rewrite time. As well, the final equation for the bus probability term of equation(12) is now

$$Pbt(P',B) = \sum_{\substack{\text{all } S, \text{where} \\ B=Bn(P',S')}} Pc(P',S') - (Sr(S') * Na(P',S'))$$

...(18)

The complete equation for the probability of processor P successfully accessing store S is found by combining equations 13 and 17, giving

$$Pa(P,S) = Cs(P,S) * [Tsr(S) * Na(P,S) - Tbd(Bn(P,S))*Na(P,S) + (Pc(P,S) - Tsr(S) * Na(P,S)) * Cb(P,S)]$$

...(19)

(6.5) OBTAINING PROCESSOR UTILIZATION

In the following the probability equation derived above will be used to calculate the number of accesses a processor P makes to a store S in unit time, and the amount of time the processor idles.

The probability of successfully accessing a store is the same as the fraction of time that the processor spends using the store. If this time is divided by the store access cycle time then the result is the number of store accesses and thus the number of cycles the processor spends in accessing that store. Thus

$$Na(P,S) = Pa(P,S)/Tsa(S) \qquad \ldots(20)$$

where the store access time Tsa does not include the store rewrite time.

From this equation the total number of processor cycles is the summation of the number of accesses to each individual store of the processor, thus

$$Nap(P) = \sum_{S=1}^{M} Na(P,S)$$

where Nap is the actual number of cycles per processor.

...(21)

The time spent by the processor in doing useful work while not referencing store is the number of processor cycles multiplied by the average adjusted processor cycle time. This last quantity is the average processor cycle time minus the store rewrite time. Thus the time spent on useful work after accessing store S is

$$Tpsr(P,S) = Na(P,S) * Tcy(D) - Tsr(S)) \qquad \ldots(22)$$

where Tpsr is the processing time and Tcy is the processor cycle time.

This can be summed over all the stores that the processor accesses to obtain the total amount of useful time spent by the processor while not accessing store or attempting to access store. Now the combined probability time gives the fraction of time that is spent in accessing and attempting to access store, and so adding these two together will give the total amount of time the processor spends in accessing store and in execution. For a fully occupied processor this time should equal one. However in this model each processor is constrained in the amount of work that may be done in relation to all the other processors. Usually only one processor will be fully occupied, the other processors will have varying amounts of idle time. Thus

$$Ti(P) = 1 - \sum_{S=1}^{M} Tpsr(P,S) + Pc(P,S) \qquad \text{...(23)}$$

where Ti is the idling time per processor.

(6.6) NUMERICAL ITERATION SOLUTION FOR THE THROUGHPUT

Combining equation(19) with equation(20) gives the probability of processor P successfully accessing store S.

$$\begin{aligned} Pa(P,S) = Cs(P,S) * [&(Cp(P,S) - Tsr(S) * Pa(P,S)/Tsa(S)) * Cb(P,S) + \\ &Tsr(S) * Pa(P,S)/Tsa(S) - \\ &Tbd(Bn(P,S))*Pa(P,S)/Tsa(S)] \end{aligned} \qquad \text{...(24)}$$

This equation has the probability term on both sides. This is due to the introduction of the actual number of cycles information into the derivation of the probability. The number of cycles in turn is related directly to the probability value. If this equation is rewritten with all the probability terms brought together, then the following is obtained.

$$Pa(P,S) = Pc(P,S) * Cg(P,S)$$

...(25)

where Cg is the global conflict function and is derived by

$$Cg(P,S) = \frac{Cb(P,S) * Cs(P,S)}{1+Cs(P,S)*(Tsr(S)*Cb(P,S)-Tsr(S) + Tbd(Bn(P,S)))/Tsa(S)}$$

...(26)

Using this definition of the probability, and combining equations 2, 20 and 25, gives

$$Ni(P,S)*Tp = Pc(P,S) * Cg(P,S)/Tsa(S)$$

...(27)

Rearranging this results in

$$Pc(P,S) = Ni(P,S)* Tp * Tsa(S)/Cg(P,S)$$

...(28)

This equation gives the combined probability in terms of the several known values, plus the throughput and the global conflict value. When producing a numerical solution the global conflict function is defined in terms of the conflict for store and conflict for bus functions. These in turn are defined using the combined probability values. To make the numerical solution possible, the previous function values of the combined probability are used to calculate these conflict functions. This produces a new combined probability for a processor store pair as predicted by all of the other old combined probabilities. However at this stage the common throughput factor Tp is unknown.

This is found by making use of the constraint imposed upon the combined probability value by equation(23). This equation can be combined with eqaution(20) and equation(22) to produce

$$1\text{-}Tidle(P) = \sum_{S=1}^{M} Pc(P,S) + \frac{(Tcy(P) - Tsr(S))*Pa(P,S)}{Tsa(S)}$$

...(29)

Using equation(26) to substitute for the probability term, and equation(28) for the combined probability terms, produces

$$1\text{-}Tidle(P) = \sum_{S=1}^{M} \frac{Ni(P,S)*Tsa(S)*Tp}{Cg(P,S)} + Ni(P,S)*Tsa(S)*Tp* \frac{(Tcy(P) - Tsr(S))}{Tsa(S)}$$

...(30)

If the idle time is temporary assumed to be zero, then this can be rearranged into

$$Tp(P) = 1/(\sum_{S=1}^{M} Tpt(P))$$

...(31)

where the throughput term Tpt is

$$Tp(P)=Ni(P,S)*Tsa(S)* (\frac{1}{Cg(P,S)} + \frac{Tcy(P)-Tsr(S)}{Tsa(S)})$$

...(32)

and there is now a separate throughput term for each processor. When a program is running, only the busiest processor will be occupied fully. All the others will have some nonzero idling time. Any of the other processors, if allowed to run full speed without any idling, will naturally have a greater throughput than when they are forced to idle for some of the time. Therefore the busiest processor will have the smallest throughput when using the equation above. This is used as the throughput of the whole system.

Finally to obtain a new value of the combined probability function, equation(28) is used.

To explain why this should converge, consider the situation when one of the combined probability terms is too large. This corresponds to a processor making too many accesses to a store. This leads to greater interference for the other processors, and so the conflict function values for these other processor store pairs will decline. Thus the calculated throughput for these other processors will be lower. The minimum throughput is always choosen, and so if some of the throughputs of the processors are decreasing, then possibily the minimum throughput will also decrease. Thus the new value of the combined probability, obtained via equation(28), will also be lower. Briefly, equation(28) adjusts the individual values of the combined probability, while obtaining the minimum throughput from equation(31) will adjust up or down the whole array so that there is one processor with zero idle time.

(6.6.1) SUMMARY OF ITERATION STEPS

The iteration solution proceeds as follows

Step 1.An initial value for the combined probability array is made, perhaps by taking the normalized value of the input number cycles array

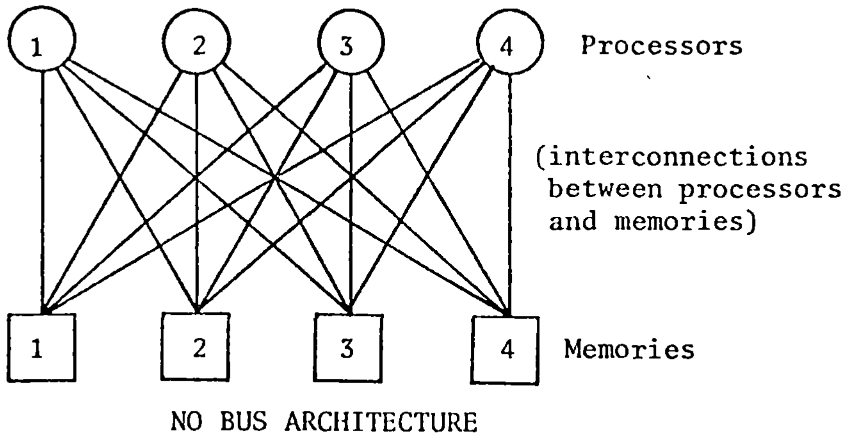
Step 2.An initial value of 0 is assumed for the last throughput.

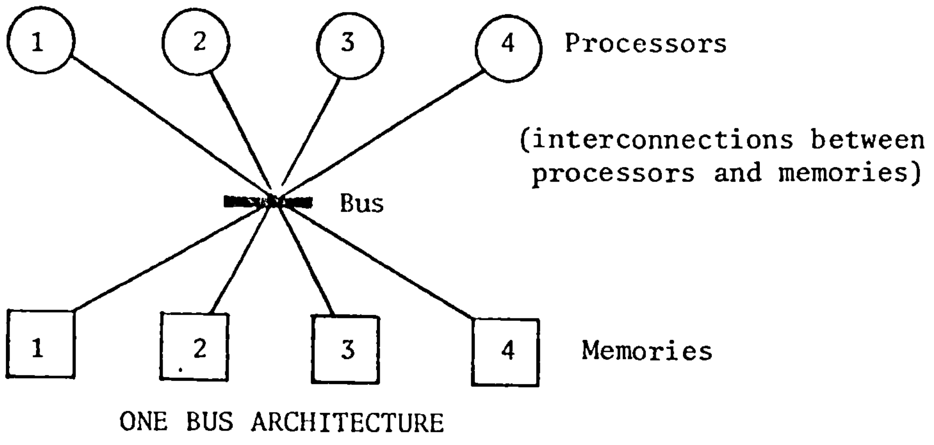
Step 3.A new value for the throughput is found by applying equation(31). If this differs by less than the error difference from the last throughput, then the iteration is finished.

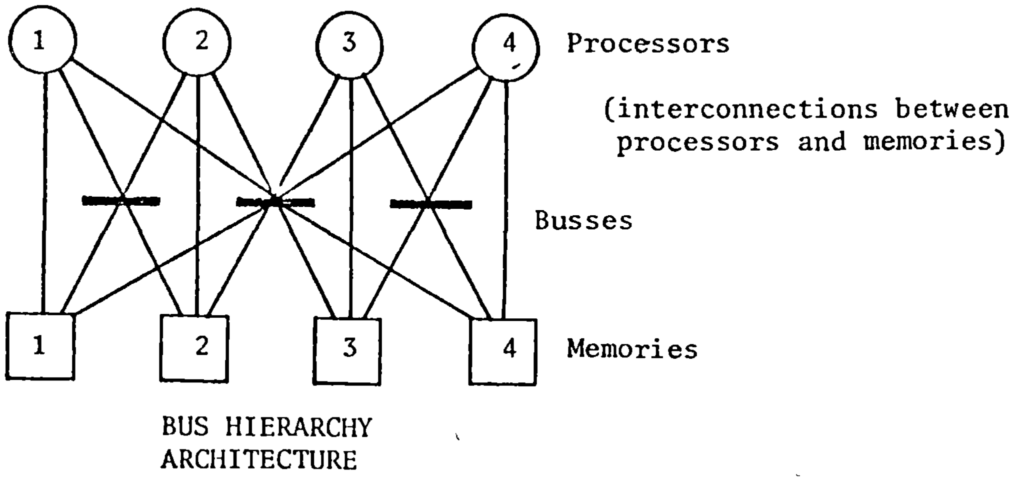
Step 4.Otherwise a new value for the combined probability array is found by using equation(28).

Step 5.Last throughput := throughput

Step 6.Go to step 3.






1
2
3
4
Processors
(interconnections between processors and memories)
Busses
1
2
3
4
Memories
BUS HIERARCHY ARCHITECTURE

The three different kinds of architecture used

Figure 6.2

(6.7) EXPERIMENTAL RESULTS

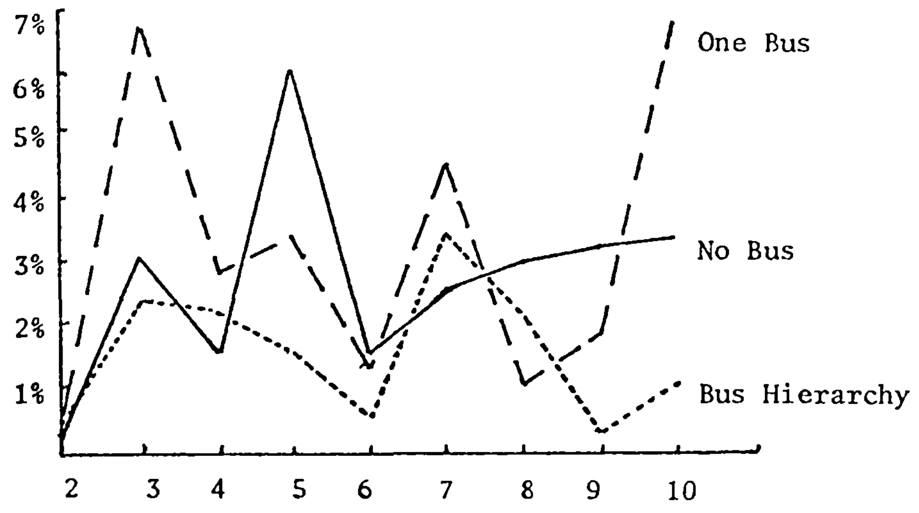
The performance and validity of the analytic general memory interference model was investigated by producing an implementation in Pascal. The performance of this was compared with a simulation program for a variety of input computer architectures. Originally the model was implemented according to the method described by Hoogendoorn. The results so obtained agreed exactly with those in his article [44]. Subsequently the model and simulation where altered to conform to the model developed in this thesis.

In the following text the verification results for the model are discussed. The accuracy and execution times of both the model and simulation are compared and it is found that, depending on the application, either the simulation or model may be the preferred implementation means of deriving the throughput for use by the allocator.

(6.7.1) MODEL VERIFICATION

In the trials three kinds of demonstration architectures were used. The first architecture has each processor directly accessing its stores without any intervening buses. The second has every processor connected to every store through a single common bus. In the third architecture each processor has direct access to its own store, shares a bus with one other processor to enable it to access that others store, shares another bus with three other processors enabling it to access the stores of those processors and so on. These architectures are pictured in figure(6.2).

The number of processors and stores in each architecture for each trial was successively increased from 2 to 10. The input number of cycles array was randomly filled with either 0 (half of the time) or with a number in the range 0 to 1.0. Similarly the speeds of the stores and processors were randomly selected over a small range. The bus delay time



Number of processors and stores.

Percentage difference between the simulation and analytic models.

Figure 6.3

for each bus was selected to be 0 for a bus linking a single processor to one store, and then increasing in proportion to the number of stores and processors that access the bus.

The verification trials were run with the models error difference set to 8 percent, this being adequate for the purposes of the allocator program. The error difference is the difference between two consecutive results obtained from the iteration algorithms used in the model.

Figure(6.3) gives the difference between the predicted throughput of the model and the actual throughput obtained from the simulation. As can be seen most of the differences are within this limit. The model generally converges within 2 iterations, and this explains why the results are generally much better than 8 percent. (The first iteration easily gets to within the required accuracy, but a second iteration is needed to obtain another throughput value for the error difference comparsion).

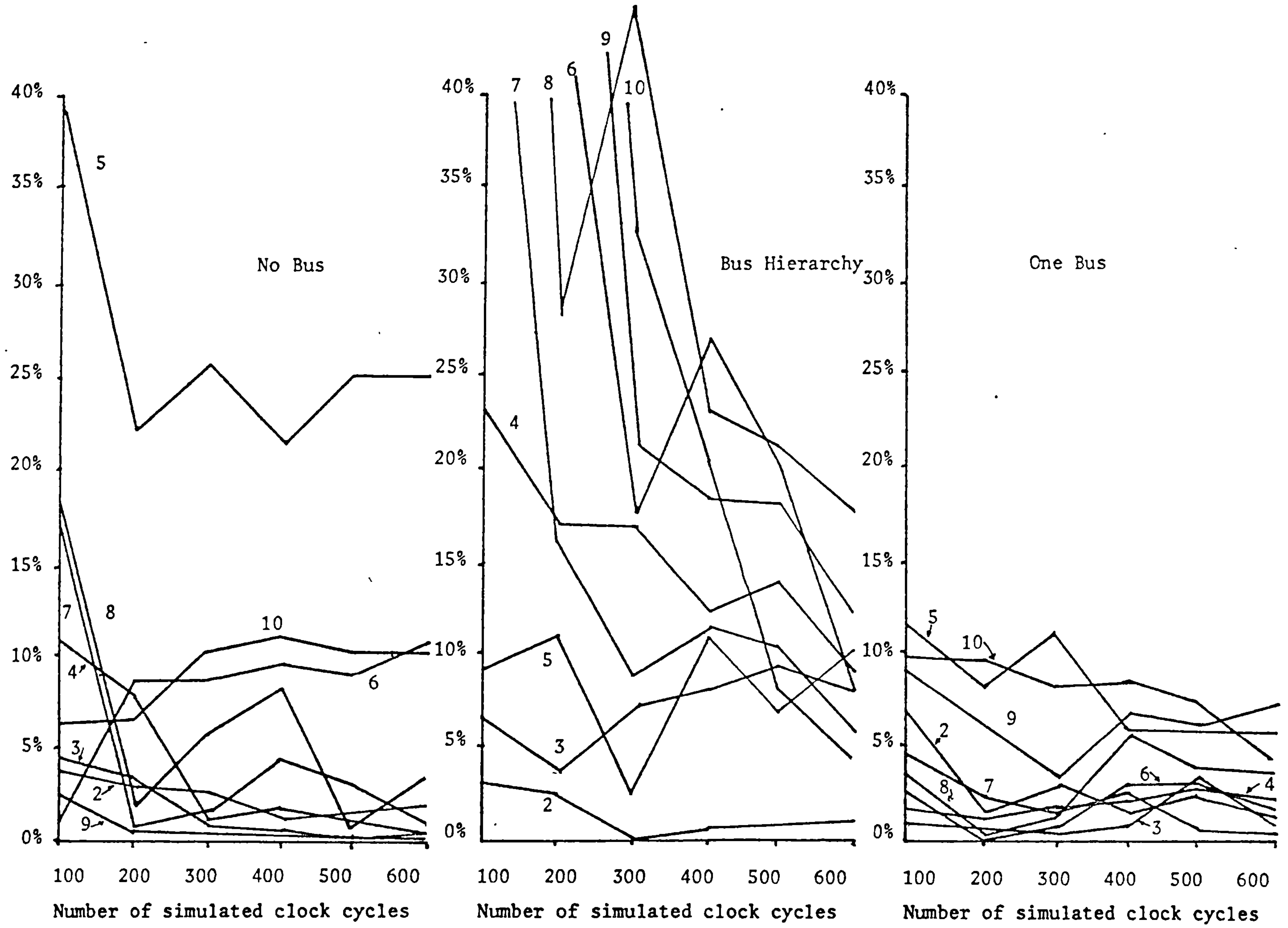


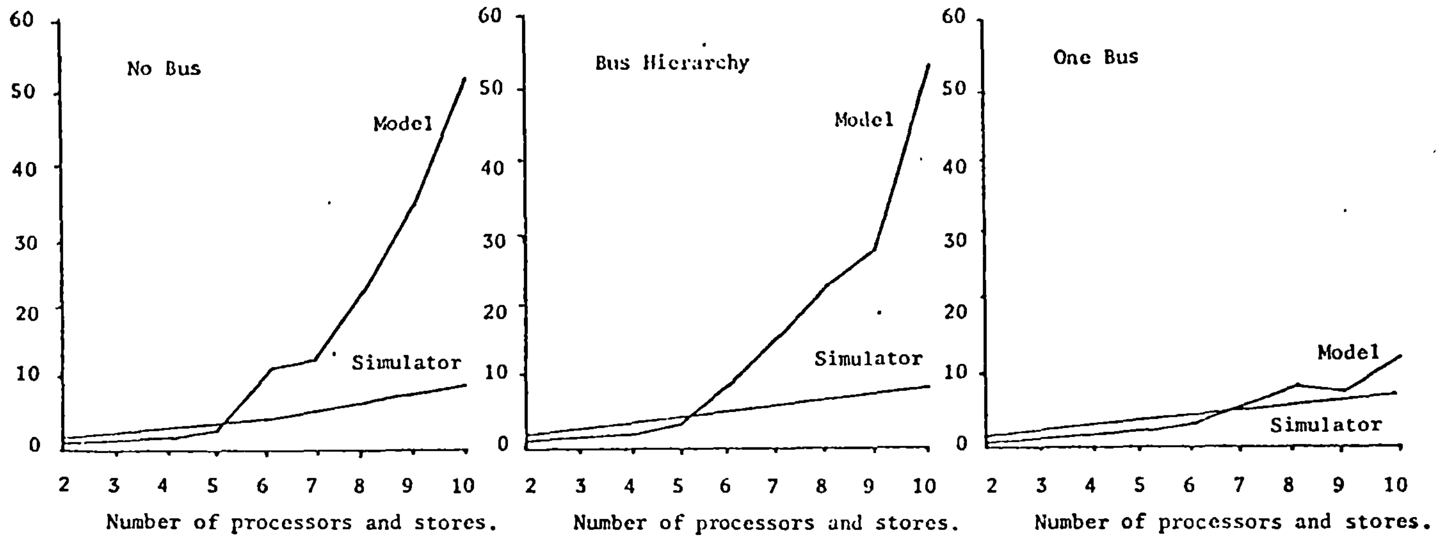
Figure 6.4

The percentage difference between the two implementations, for an architecture with 2 to 10 processors and stores.

(6.7.2) IMPLEMENTATION OF THE SIMULATOR

The original simulation for Hoogendoorns model is straight forward to implement, but for the model developed in this thesis, some extensions are required. When the processors are executing independent programs the simulation is written so that as soon as each processor finishes an execution cycle it makes a store fetch to start a new one. However when the processors work loads are dependent upon each other, provision has to be allowed to decide after each processor cycle if an idle cycle needs to be inserted or not. This is done in two different ways in the implementation of the simulation. The first,approximate, method is to run the analytic model first and have it produce a static access array, giving the probability that a processor will access a particular store. The cumulative probability of accessing the stores is one for the busiest processor, and less than one for the other processors. This difference represents the idle time for the other processors, and the simulation will choose between fetching a store and inserting an idle cycle accordingly. The derivation of the static access array is given in appendix(A). This relies upon the assumption about the relevance of the input number of cycles array on a cycle by cycle basis, as discussed in section 6.2.2.

An alternative method that does not rely on results produced by running the model first is to simply count the number of cycles of each processor and compare them to the input number of cycles array.Whenever a processor has done enough cycles it is idled until all of other processors have caught up with it. Then all of the processors are allowed to execute again. This can generate a better answer, since it reflects somewhat more closely the actual pattern of processor execution when synchronized by semaphores. As can be seen in figure(6.4) the results are just passable with a simulation run of 300 clock cycles (with the error difference ranging from 0 percent to 45 percent) and reasonable for a simulation run of 600 clock cycles.



Execution times in seconds for the simulator and model.

Figure 6.5

(6.7.3) EXECUTION TIMES

During the test runs a record was kept of the execution time consumed by each. This information is presented in figure(6.5). As can be seen, the models execution time increases much faster than that for the simulation. This can be explained by comparing the number of basic operations required by each.

For the model, consider the case where the architecture has no buses and there are N processors and N stores. In this case the number of processor store combinations is N^2, and the conflict function is called once for each of these, and this functions implementation requires operations proportional to N^2. (Here the ^ character is used to represent the exponential operator). Thus in this situation the model requires operations in proportion to N^4 for a constant number of iterations.

On the other hand the simulation, for a constant number of clock cycles, needs to select a random combination of processors each cycle, done in a maximum of N^2 operations, and then to select a random store, achievable in time LogN. Thus the total is a maximum of (N^2)logN. This, in the limit, is much less than the time for the model.

(6.7.4) SUMMARY

The simulation and probabilistic models differ in their execution times and accuracy of results. These are summarized here

A) Execution times.

The simulation model has a much slower rate of execution time increase for increasing N compared to the analytic model. For the implementations used in this thesis, the crossover point is at N equal to 5 or 6, for N processors and N stores. Below this point the analytic model is marginally faster, above this point the simulation is much faster.

B) Accuracy of results.

Both models introduce inaccuracies into the results. The results from the simulation model will be inaccurate due to

1) The approximate method used to include the effects of dependent processor execution workloads.

2) The approximations due to the use of random functions in the simulation model.

The probabilistic model can be inaccurate for some special cases, as for example when two processors are able to execute in lockstep without memory interference. This case is described in section(6.2.2).

Furthermore both models are equally inaccurate due to the influences of interactions between processors via semaphores. This is also discussed in section(6.2.2).

CHAPTER (7)

(7.1) INTRODUCTION TO THE ALLOCATION ALGORITHMS

In this chapter the algorithms for the allocation of a program to a computer architecture are described. These produce a final allocation by utilizing

A) information obtained from the description of the computer, as specified by the architecture specifications,

B) compiler supplied information about the memory and process elements of the program,

C) the constraint information derived from the constraint specifications,

D) the throughput estimation obtained from the input number of cycles information.

The overall information flow of the allocation can be seen by referring back to figure(1.1).

(7.1.1) PREVIOUS WORK

The StarOS research reported by [26] deals with specifying to an allocator the computer architecture and the allocation constraints. However no allocation algorithms were implemented to actually perform the allocation.

Another research paper, this time by [33], deals with the partitioning of computational objects onto a distributed computer system. Here the computer system consists of computer modules communicating via some interconnection system. This imposes a constant communication cost between each module. Their aim is to reduce the communication times for a system of programs which may need to run on a number of computer modules (e.g. need to access a disk from one module, and a terminal from another module). The method is to obtain a run time trace of the execution of a program. This is then used to generate a partition of the programs components so as to minimize the costs. An approximate graph optimization method is used.

This model does not apply very well to the allocation problem of this thesis. The model concentrates on separate computer modules which communicate between each other. Whereas in this thesis the computer architecture is modelled, not as computer modules, but as individual processors and stores. The costs to be minimized deal not with communication costs between computer modules, but with processing time within the processors and the store accessing times. As well their model has no provision to allow the effects of memory and bus contention to be taken into account.

(7.1.2) APPROACH USED

The algorithmic basis choosen for this research is to successively try out alternative allocation mapping solutions, calculating the throughput for each. A legal map with all program elements allocated is called a feasible solution. Whenever such a feasible solution is found, its throughput is compared with the throughput of the best feasible solution found so far. If it is better then this map becomes the incumbent solution. When the search terminates, the incumbent will be the optimal feasible solution, and becomes the allocation mapping for the program.

Since most of the program elements will be allocatable to more than one resource, then the enumeration of all possible mapping solutions will result in a tree pattern search. Thus to simply generate each possible combination of process to processors and memory to stores and then to check its legality is exponentially time consuming. Instead possible solutions are enumerated by starting with an initially unallocated program and assigning its elements one by one. At each such step the legality of the partial map solution and is execution time efficiency is examined. If it can be shown that no legal solutions can be derived from this partial solution, or that all possible solutions derived by completing this partial solution are less efficient than that incumbent, then this partial solution can be discarded. This allows all of the solutions, feasible or otherwise, that can be completed from this partial mapping to be discarded without examination. If enough partial solutions can be rejected in this way then the search space will be reduced to manageable proportions. In general, for all but the most trivially sized programs and computer architectures, this reduction in the search space size will be necessary to allow the generation of any feasible solutions at all. Thus the bulk of the allocation algorithms are concerned with the problem of detecting illegal or inefficient maps as early as possible.

This algorithm method is known as implicit enumeration with backtrack, and is described in [27]. The term implicit enumeration arises because the solutions of a partial map that are rejected can be considered to have been implicilty enumerated. This is in contrast to the other complete solutions that have been explicitly enumerated.

The following text will expand upon this introduction. Firstly the starting information for the allocation algorithms is described. The means of computing the though put is discussed, and the search method used is then introduced. Lastly the allocation map evaluation algorithms are detailed.

Finally a point on the notation. In the following discussion the terms

1) process 2) memory 3) processor 4) store

are used. These are taken to refer to

1) the process elements of a program.

2) the address space elements of a program.

3) the hardware processor resources of a computer architecture.

4) the hardware memory resources of a computer architecture.

(7.2) THE INPUT INFORMATION TO THE ALLOCATOR

The specification of the computer architecture, program structure and user required constraints have been discussed previously. This information is converted by a preprocessor and supplied to the allocation algorithms in a simplified form. This is outlined in the following.

The construction of the preprocessor, which would be part of the complete allocation package, poses no new problems and its design is not discussed, nor was an implementation produced.

(7.2.1) COMPUTER ARCHITECTURE STRUCTURE

The computer architecture specified by the ISL program would be converted into a simplified architecture graph. In this, the information that is kept is concerned with the description of the processors, stores, banks and buses, along with the access paths between these. This information is

A) For processors, cycle times and kinds are retained.

B) For stores, access speeds, starting address locations and address ranges are retained.

C) For banks, the bank access time is retained.

D) For buses, the bus access time is retained.

As well, the arcs connecting the vertices representing this information are rearranged. If a processor, bus or bank accesses a store, bus or bank, then there exists a direct arc between these two vertices.

The rest of the information in the original architecture specification graph is not required at this stage in the allocation activity. It has already been used in the production of the simplified architecture graph and in the construction of allocation constraints.

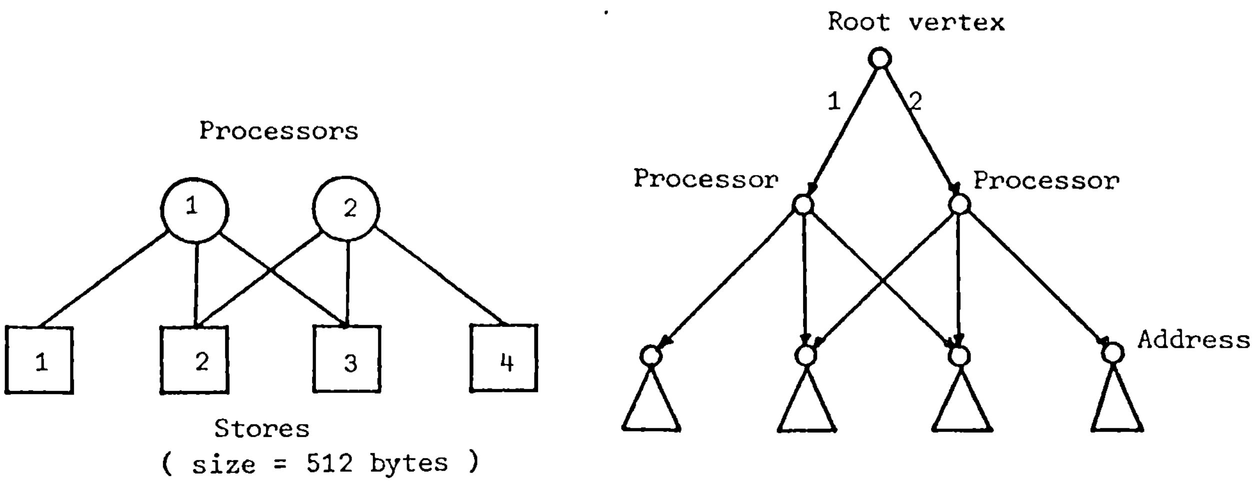


Figure 7.1

The resulting graph is available to the allocator, which can extract several kinds of information from it. Firstly the accessibility of one kind of vertex from another can be obtained by a function of the form

```
ACCESS_X_FROM_Y ( [ Y ] )
```

Here X and Y represent any of the four kinds of vertices PROCESSOR, STORE, BANK and MAP. The function takes an input set of one type of vertex and returns the set of all vertices of the other type that can be accessed from this input set, or accessed by this input set. As an example, consider the computer system as set out in figure(7.1). In this both a pictorial representation and a graph representation is given. For this structure, the following function calls would give the indicated results.

```
ACCESS_STORE_FROM_PROCESSOR ( [ PROCESSOR_1 ] )
   gives [ STORE_1, STORE_2, STORE_3 ]

ACCESS_STORE_FROM_PROCESSOR ( [ PROCESSOR_1, PROCESSOR_2 ] )
   gives [ STORE_1, STORE_2, STORE_3, STORE_4 .]

ACCESS_PROCESSOR_FROM_STORE ( [ STORE_1 ] )
   gives [ PROCESSOR_1 ]
```

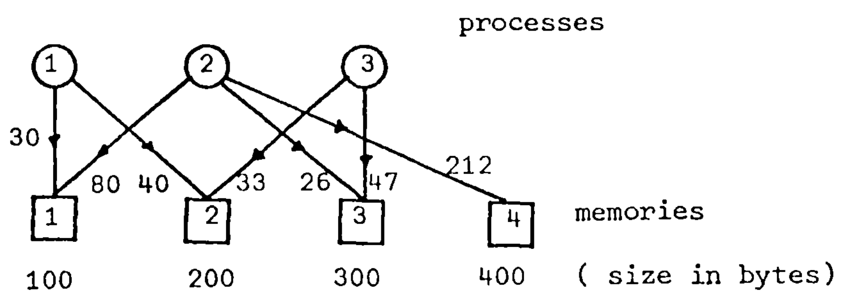


Figure 7.2

There are also functions that return the size of a store vertex, the cycle time of a processor or the access time of a store, bank or bus.

The implementation of such a graph structure on a computer is straight forward and is not discussed any further.

(7.2.2) SPECIFICATION OF THE PROGRAM

Also supplied to the allocator is an information graph depicting the structure of the program. This information is produced by the compiler and it is represented as a simple two level graph structure containing process vertices and memory vertices. An arc from a process to a memory vertex represents the use of that memory by the process. Each memory vertex has associated with it the size of the memory.

Also associated with each process memory combination is the number of memory accesses that the process makes to the memory in a given time unit. This information is obtained by compiling the program on an ordinary computer and executing it to gather memory access statistics. This is required to allow the production of the throughput estimations.

To access this graph structure, there are access functions of the form

```
ACCESS_X_FROM_Y ( [ Y ] )
```

which are used in the same way as the ones for the architecture specifications. There are also the functions to extract the memory size and number of memory accesses between a given process and memory.

As an example figure(7.2) represents a program with three processes and four memories. The directed arcs represent access from a process to a memory. Also given are the sizes of the memories, and the number of accesses between a process and a memory. From this can be extracted the information

```
ACCESS_PROCESS_FROM_MEMORY ( [ MEMORY_1 ] )
   gives [ PROCESS_1, PROCESS_2 ]

ACCESS_PROCESS_FROM_MEMORY ( [ MEMORY_1, MEMORY_3 ] )
   gives [ PROCESS_1, PROCESS_2, PROCESS_3 ]

SIZE_OF_MEMORY ( [ MEMORY_1, MEMORY_2 ] )
   gives 300

PROCESS_MEMORY_NUMBER_CYCLES ( PROCESS_1, MEMORY_1 )
   gives 30
```

(7.2.3) CONSTRAINT SPECIFICATION

Finally, simple constraints are derived from the user supplied object specifications, constraint specifications and the computer architecture specifications. There are three forms of constraints.

One constraint form specifies the set of processor or store resources that a process or memory element is allowed to be mapped to. This can be represented as

```
X -> [ Y ]
```

where X refers to a process or memory element and [Y] refers to a set of the appropriate resource elements. This information is accessed by a function like

```
    ALLOWED_X_FROM_Y ( [ Y ] )
```

Where X represents the name of either a resource element kind such as PROCESSOR or STORE, or a program element kind such as PROCESS or MEMORY. The Y refers to the corresponding program element or resource element kind name. This function returns the set of resource elements that the program elements in the set [Y] are allowed to be assigned to, or it returns the set of program elements that are allowed to the given resource elements of the set.

For example, if some process constraints are

```
    PROCESS_1 -> [ PROCESSOR_1, PROCESSOR_2 ]
    PROCESS_2 -> [ PROCESSOR_1 ]
    PROCESS_3 -> [ PROCESSOR_2 ]
```

then two function calls and their results may be

```
    ALLOWED_PROCESSOR_FROM_PROCESS ( [ PROCESS_2 ] )
       gives [ PROCESSOR_1 ]

    ALLOWED_PROCESS_FROM_PROCESSOR ( [ PROCESSOR_1 ] )
       gives [ PROCESS_1, PROCESS_2 ]
```

This function may act as its own inverse for some possible input values. As an example

```
    ALLOWED_PROCESSOR_FROM_PROCESS (
       [ PROCESS_1, PROCESS_2, PROCESS_3 ] )
       gives [ PROCESSOR_1, PROCESSOR_2 ]

    ALLOWED_PROCESS_FROM_PROCESSOR (
       [ PROCESSOR_1, PROCESSOR_2 ] )
       gives [ PROCESS_1, PROCESS_2, PROCESS_3 ]
```

The second constraint imposes a proximity constraint upon a set of process or memory elements. The relation may be to allocate each element to a different set of resources, or to the same set of resources. These constraints are known as Different_Constraints or Same_Constraints, and may be represented as

```
[ X ] -> DIFFERENT [  [ Y1 ] , [ Y2 ] , ... ]
[ X ] -> SAME      [  [ Y1 ] , [ Y2 ] , ... ]
```

Where [X] is the set of program elements upon which the proximity constraint is to be applied, and the right hand side lists the sets of target resources [Y] to which the elements may be mapped. An example Different_Constraint is

```
[ PROCESS_1, PROCESS_2 ] ->
   DIFFERENT [ [ PROCESSOR_1 ], [ PROCESSOR_2 ] ]
```

This will ensure that the two processes will go to the two different processors. Thus if PROCESS_1 ends up on PROCESSOR_1, then the only legal assignment for PROCESS_2 is to PROCESSOR_2. An example of a Same_Constraint is

```
[ MEMORY_1, MEMORY_2, MEMORY_3 ] ->
   SAME [ [ STORE_1, STORE_2 ], [ STORE_3, STORE_4 ] ]
```

This constraint will enforce the condition that the three memories will all be assigned to either the first two stores or the second two stores.

The proximity constraint specifications are accessible by several functions which retrieve either a specific proximity constraint, or all constraints that contain a given resource or program element.

Thirdly there are the address constraints. These act to fix a program element to some specific physical store address. Thus this can be treated as a nonproximity constraint acting on the program element.

(7.2.4) EXAMPLE MAP ALLOCATION

The allocation program, if it is successful, will produce an allocation mapping for the program onto the computer architecture. An example legal mapping is developed below to give a demonstration of a final map.

The computer architecture of figure(7.1) and the program in figure(7.2) are used. The constraints imposed by the user are those given in the examples above, and repeated below.

```
PROCESS_1 -> [ PROCESSOR_1, PROCESSOR_2 ]
PROCESS_2 -> [ PROCESSOR_1 ]
PROCESS_3 -> [ PROCESSOR_2 ]

[ PROCESS_1, PROCESS_2 ] ->
   DIFFERENT [ [ PROCESSOR_1 ], [ PROCESSOR_2 ] ]

[ MEMORY_1, MEMORY_2, MEMORY_3 ] ->
   SAME [ [ STORE_1, STORE_2 ], [ STORE_3, STORE_4 ] ]
```

Firstly it is seen that processes PROCESS_2 and PROCESS_3 are already fixed to their final processors. From this the Different_Constraint specifies that PROCESS_1 has to go to PROCESSOR_2, since PROCESS_2 is already assigned to PROCESSOR_1.

Next the memories are assigned. MEMORY_1 is accessed by both PROCESS_1 and PROCESS_2. Therefore it has to be assigned so that PROCESSOR_2 and PROCESSOR_1 can access it (since the processes are assigned to those processors). Thus the only allowable stores are STORE_2 and STORE_3. This results in the constraint

```
MEMORY_1 -> [ STORE_2, STORE_3 ]
```

A similar exercise will produce the constraints for the other memories

```
MEMORY_2 -> [ STORE_2, STORE_3, STORE_4 ]
MEMORY_3 -> [ STORE_2, STORE_3 ]
MEMORY_4 -> [ STORE_1, STORE_2, STORE_3 ]
```

The Same_Constraint specifies that the first three memories can go to either stores 1 and 2, or stores 3 and 4. If the arbitrary choice of stores 3 and 4 is made, then MEMORY_1 becomes fixed in STORE_3. All three memories cannot go to this store because they will not fit, and one possible assignment is

```
MEMORY_1 -> [ STORE_3 ]
MEMORY_2 -> [ STORE_4 ]
MEMORY_3 -> [ STORE_3 ]
```

Finally the MEMORY_4 element has three possible stores, so a selection like

```
MEMORY_4 -> [ STORE_1 ]
```

can be made. Thus a final legal allocation mapping has been generated. This is probably not the most efficient. In the following sections the systematic method of finding legal and efficient mappings developed during this research is described.

(7.3) THE ALLOCATOR SEARCH TECHNIQUE

The allocation search algorithm must be able to find a legal and efficient solution in as few trials as possible. How the search is carried out can greatly affect this. In this section the two techniques that can be used for search optimization are introduced. Basically these are to attempt the removal of unprofitable search branches, and strive to achieve legal and efficient mappings, as early as possible in the search.

(7.3.1) DETECTION OF UNPROFITABLE SEARCHES

During the enumeration of the solutions for a particular program, partial solutions will be discarded wherever possible. This occurs when

A) the current partial mapping solution can never be completed to produce a feasible solution, or

B) all possible feasible solutions produced by completing this partial map will have a throughput less than the throughput of the best feasible solution found so far.

(7.3.1.1) DETECTING ILLEGAL MAPS

The first, of predicting if a current partial mapping will ever lead to the generation of a feasible solution, is based upon the principle that once an illegal partial solution has been produced, all subsequent complete mappings derived from this will be illegal. An illegal map is one where the constraints upon a program element will prevent it from being assigned to any computer resource. These constraints arise from the amount of space left in the memory blocks, and the accessibility between processes and memories. Accordingly the assignment of any other, as yet unassigned, program elements can never remove any of these constraints. Thus this prevents any legal solution from ever being derived from an illegal partial solution.

The detection of such illegal partial maps is achieved by making use of the allocation constraint associated with each process and memory element. This constraint is originally just the user supplied constraint, when one is specified. For example

```
PROCESS_1 -> [ PROCESSOR_1 , PROCESSOR_2 ]
```

where the process élement PROCESS_1 is allowed to the processors PROCESSOR_1 and PROCESSOR_2. The technique is to reduce at each step this allowable constraint on each element as much as possible. This is done with the aid of constraint reduction operations. Sometimes the constraint may be narrowed down to only one resource, in which case the element has just become allocated to its final position. In most cases it will only be possible to reduce the constraint by a small amount, or not at all. However it might also be possible to reduce the set to the null set, that is under the current partial mapping there are no legal resources that the element may be assigned to. In this case the map allocation fails, and the current search branch can be dropped.

As an example of this consider the allocation mapping derived in section(7.2.4) above. As each user constraint was applied, the constraints on the program elements were reduced. The constraint for the PROCESS_1 element was originally

```
PROCESS_1 -> [ PROCESSOR_1, PROCESSOR_2 ]
```

but through the actions of the other two constraints for the processes, and the Different_Constraint, this was reduced down to

```
PROCESS_1 -> [ PROCESSOR_2 ]
```

If there had been the additional constraint

```
[ PROCESS_1, PROCESS_3 ] ->
   DIFFERENT [ [ PROCESSOR_1 ], [ PROCESSOR_2], [ PROCESSOR_3 ] ]
```

then since PROCESS_3 has already be assigned to PROCESSOR_2, the constraints are now in conflict. Both PROCESS_1 and PROCESS_3 are assigned to PROCESSOR_2, contary to the Different_Constraint. Thus in this situation the constraint on PROCESS_1 would be reduced to the empty set

```
PROCESS_1 -> [ ]
```

and the partial allocation can be rejected.

(7.3.1.2) DETECTING INEFFICIENT MAPS

The second means of detecting unprofitable searches is to check the calculated throughput of the current incomplete map at each search step. This is compared with the throughput of the best final map allocation found so far. If this throughput is less then the partial map of this current search can be terminated, since it will not lead to any final solution with a better throughput. Even better is to select only those incomplete maps that show a definite improvement over the best map found, such as a 10 percent greater throughput. This percentage is called the throughput factor. Using this would prevent the examination of a long series of almost identically performing allocations.

This method relies upon two principles-

A) For any partial solution a throughput can be found.

B) The throughput of any solution derived from a partial map can never exceed the throughput of the partial map. This is satisfied if the maximum upper bound throughput of each successive partial map is a monotonically decreasing function. Thus the throughput of a partial map will be the maximum upper bound upon the throughput of the final complete map.

The throughput of an allocation can be found by using the general memory interference model. This requires an INPUT_NUMBER_CYCLES array. Given the PROCESS_MEMORY_NUMBER_CYCLES array and the allocation of processes to processors and logical address spaces to memories, the input number of cycles array can be calculated by

```
FOR ALL PROCESSOR DO
   FOR ALL STORE DO
      INPUT_NUMBER_CYCLE[PROCESSOR,STORE] := 0 ;
      FOR ALL PROCESS FIXED TO PROCESSOR DO
         FOR ALL MEMORY FIXED TO STORE DO
            INPUT_NUMBER_CYCLE[PROCESSOR,STORE] :=
               INPUT_NUMBER_CYCLE[PROCESSOR,STORE] +
               PROCESS_MEMORY_NUMBER_CYCLES[PROCESS,MEMORY] ;
         END ;
      END ;
   END ;
END ;
```

This calculation provides the throughput for a complete solution. The throughput for a partial solution can also be defined by this. This just implies that, since some of the processes and memories of the program are not yet assigned, then some of the PROCESS_MEMORY_NUMBER_CYCLES values will not be included.

This throughput for the partial solution has yet to be shown to be the maximum upper bound throughput. Consider some partial allocation mapping. This has a throughput that can be defined for each processor as

```
THROUGHPUT(PROCESSOR) =
   CONFLICT_FACTOR(PROCESSOR) / THROUGHPUT_TIME(PROCESSOR)
```

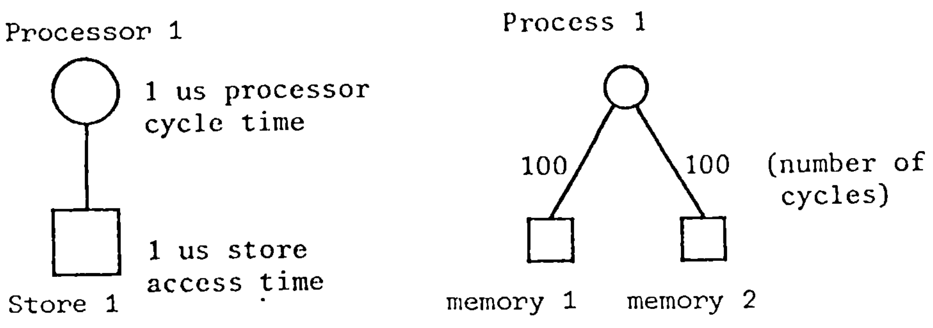


Figure 7.3

where the THROUGHPUT_TIME is given by

THROUGHPUT_TIME(PROCESSOR) =

$$\sum_{\text{For all STORE}} \text{INPUT\_NUMBER\_CYCLE(PROCESSOR,STORE)} * (\text{PROCESSOR\_CYCLE\_TIME(PROCESSOR)} + \text{STORE\_ACCESS\_TIME(STORE)} + \text{BUS\_DELAY\_TIME(PROCESSOR,STORE)})$$

The summation represents the time spent in each processor cycle and each memory fetch cycle, assuming no memory interference. The interference is represented by the conflict function , which is 1 for no interference and 0 for complete interference. The throughput of the complete system will be the minimum of the throughput terms above. Note that this conflict factor is not directly given by any of the equations derived in chapter 6 on the analytic probability model.

The memory interference can only decrease if the number of memory accesses is decreased. However when successively allocating processes and memory address spaces, the number of memory accesses will always increase. Thus the memory interference is always increasing, and so the conflict factor is a monotonically decreasing function. Similarly the INPUT_NUMBER_CYCLES values can never decrease in this situation. Thus the summation will be a monotonically increasing function. Therefore the throughput function will be a monotonically decreasing function.

The minimum throughput function for any partial allocation mapping is taken to be the throughput estimation. The throughput estimation for a complete allocation solution is the actual estimated throughput. Since the function is monotonically decreasing, then any throughputs of partial solutions must therefore be a maximum upper bound throughput.

To demonstrate this with an example, consider the architecture and program in figure(7.3). If the program has been partially allocated so that PROCESS_1 has been assigned to PROCESSOR_1 and MEMORY_1 has been assigned to STORE_1, then the calculated throughput will be

```
ThroughPut = 1000000 / ( Time to execute 100 memory accesses )
                         in microseconds )
           = 1000000 / (  (1+1) * 100 )
           = 5000
```

where the instruction cycle time is 1 microsecond and the store access time is also 1 microsecond. If MEMORY_2 is also allocated, then the throughput calculations will now give

```
           = 1000000 / (  (1+1) * (100+100) )
           = 2500
```

In other words this calculated throughput is half that of the first throughput. Adding more memories will always decrease the throughput. Similarly with the addition of extra processes.

(7.3.1.2.1) IMPROVING THE THROUGHPUT CALCULATIONS

The throughput is calculated from the partial map at each search step. At shallow levels in the search there will only be a few processes and memories allocated and thus the throughput calculated from these will generally be an over estimation of the throughput of the final complete map. For example, in the first partial allocation of the example immediately above,

```
PROCESS_1 -> [ PROCESSOR_1 ]
MEMORY_1 -> [ STORE_1 ]
```

the throughput is only calculated upon the accesses that PROCESS_1 makes to MEMORY_1. The other accesses to other memories are ignored, and thus the program will seem to run faster than it actually would. In the following is discussed a means of increasing the throughput accuracy for the initial stages of a map allocation.

Consider the case where there is only one process and memory allocated so far in a partial map. The throughput can only be calculated based upon the number of times the process accesses this memory. If the process spends an equal amount of time accessing ten other not yet assigned memories as well, then this throughput will be an over estimation by a factor of ten at least. The work load represented by the accesses to these memories may be partly incorporated if, for the purposes of calculating the throughput, each memory is assumed to be residing in a separate new store by itself. These stores are to be directly accessible to each processor of the computer, and they have cycle times that are as fast as the fastest normal store. The processors accessing these stores will suffer memory interference, but only with other processors accessing the same memory in this store. Thus only the absolutely unavoidable memory interference is included. The throughput calculated under these conditions will never be lower than the final actual throughput. Indeed it will provide a better maximum upper bound for the calculated throughput. These stores are called phantom stores, since they do not exist in the actual computer architecture and can never have any memory elements assigned to them by the allocation program. Instead the throughput algorithms use these stores to hold any unassigned memories whenever it calculates the throughput of a partial map.

Exactly the same technique is applied to unassigned processes. Each unassigned process is assumed by the throughput algorithms to reside in a phantom processor which is as fast as the fastest real processor in the system, and is directly connected to every store in the system. Thus for a partial map with only one process and one memory assigned, the phantom stores hold all of the other memories to which the process may access, and the phantom processors hold all of the other unassigned processes. These processes will access both the assigned memory and the unassigned memories. Thus the effect of memory interference will be incorporated into the calculated throughput from both the assigned processes and memories and the unassigned processes and memories.

Another way of looking at this is to regard the phantom processors and stores as implementing an ideal computer architecture. Each processor is as fast as the fastest real processor. Each phantom store is as fast as the fastest real store. Each processor has direct access to each store without any intervening buses. Finally there is a store and processor for each memory and process element in the program. Thus this provides a theoretical upper bound to the throughput for the particular program.

Finally, in the implementation of the allocation program, a separate phantom store for each memory was not implemented. This is because the number of memory elements in a program is generally greater than the number of stores or processors. Therefore adding a phantom store for each memory will significantly increase the total number of stores and processors that the throughput algorithms have to deal with. This increases the execution time. To reduce this only one phantom store is used. The simulator is modified so that each processor accessing this store can do so without any store interference from any other processor that may also be accessing it at the same time. This implies that the derived throughput no longer reflects the memory contention between different processors accessing the same unassigned memory element. However it still includes the affect of the time taken by a single processor to access these unassigned memories. Hence it is still sufficient in providing an improved upper bound upon the throughput.

(7.3.2) PRODUCING EFFICIENT MAPPINGS EARLY IN THE SEARCH

Another way of increasing the chances of producing good solutions is to order the search so as to maximize the chance of getting an efficient and legal mapping early in the search.

This can be achieved by selecting for assignment the busiest processes and the most heavily used memories early in the search, and leaving the processes with the least work to last. As well, at any step a process or memory is generally assigned first to the fastest processor or store that is allowed to it. This ordering will allow the most important processes and memories, from the viewpoint of execution time efficiency, to be assigned early in the search to the fastest processors and stores. This is not guaranteed to generate legal maps or the most efficient maps, but only to increase the chances of doing so. The details of this ordering will now be discussed.

(7.3.2.1) PROCESS AND MEMORY ORDERING

Firstly the processes and memories of the program are ordered into a process memory list. The first element in this list is the process which makes the most memory accesses to all of the programs memories. That is the process with the maximum of the function

$$\text{NUMBER\_CYCLES\_PER\_PROCESS(PROCESS)} = \sum_{\text{all memory}} \text{PROCESS\_MEMORY\_NUMBER\_CYCLES[PROCESS,MEMORY]} \;)$$

Thereafter the elements are selected one by one and appended to the list. The criterion used in this selection is based upon the evaluation of the following functions at each selection.

$$\text{NUMBER\_ACCESSES\_BY\_PROCESS(PROCESS)} = \sum_{\substack{\text{All memory}\\ \text{in the list}}} \text{PROCESS\_MEMORY\_NUMBER\_CYCLES (PROCESS,MEMORY)}$$

$$\text{NUMBER\_ACCESSES\_BY\_MEMORY(MEMORY)} = \sum_{\substack{\text{All processes}\\ \text{in the list}}} \text{PROCESS\_MEMORY\_NUMBER\_CYCLES (PROCESS,MEMORY)}$$

These values are computed for all the processes and memories that are not in the list. The element which has the highest NUMBER_ACCESS value is the one choosen.

As an example the program in figure(7.2) is used. The process with the most overall memory accesses is PROCESS_2, with 318 accesses. This becomes the first in the list. The next element will be a memory, and

MEMORY_4 is the one that PROCESS_2 accesses the most. The number of accesses between the elements of the list is 212. MEMORY_1 will be choosen for the third element, it increases the number of accesses by 80. The fourth element will be a process, PROCESS_1, since it increases the accesses the most with 30. The list would continue to be constructed in this manner, resulting in

PROCESS_2(0), MEMORY_4(212), MEMORY_1(80), PROCESS_1(30),
MEMORY_2(40), PROCESS_3(33), MEMORY_3(73).

The numbers in brackets represent the increment added to the total number of accesses for each element.

(7.3.2.2) PROCESSOR AND STORE ORDERING

The ordering for processes and memories is done only once for the entire allocation. However at each search step an ordered processor or store list is required for the element that is to be assigned next. The resources in this ordered list come from the allowed processor or store set of the element. To demonstrate this, the first element of the process memory list above might have the allocation constraint

PROCESS_2 -> [PROCESSOR_1, PROCESSOR_2]

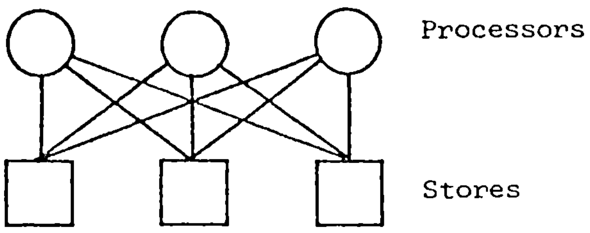
if the architecture of figure(7.1) is used. The second element is a memory and might have

MEMORY_4 -> [STORE_1, STORE_2, STORE_3, STORE_4]

These resource element sets may be reduced by various constraint reductions, but until that happens the resource sets as shown will be used. They may be ordered either

A) by calculating the throughput obtainable when the element is assigned to each resource element in turn, and using this to sort the list of resource elements, or

B) by ordering the processor or store list using some heuristic principle.



Homogeneous architecture

Figure 7.4

No definite algorithm providing optimal performance in all cases was found. Instead the methods used to sort the lists were choosen on the basis of what appeared to give the best results. The performance of these methods depend crucially upon the kind of computer architecture that is being used. Of course they will also be infiuenced by the structure of the program. However this structure will vary widely between different programs, while the computer architectures being used will show much less variation. Consequently, only the structure of the computer architecture is taken into account. In this application the kinds are best divided into two classes-

A) Homogeneous architectures, where every processor has access (directly or indirectly) to every store of the computer system. An example is the architecture in figure(7.4). A homogeneous architecture implies that a process may be assigned to any processor and still be able to access any of its memories, regardless of what stores they may end up being assigned to. Therefore processes can initially be assigned to any processor and still have a good chance of obtaining a legal, complete mapping. So in this case a good approach is to ignore the memories and to attempt to assign a process to the processor which has the least number of other processes already assigned to it or allowed to be assigned to it. In other words in a homogeneous architecture the processors are sorted upon the number of processes that are allowed to them.

For an example of this, assume the following constraints

PROCESS_1 -> [PROCESSOR_1, PROCESSOR_2, PROCESSOR_3]
PROCESS_2 -> [PROCESSOR_1, PROCESSOR_2]
PROCESS_3 -> [PROCESSOR_1]

In this situation only PROCESS_1 can ever be assigned to PROCESSOR_3 and so this is the best choice for this process (without knowing any additional information). On the other hand PROCESSOR_1 can have all three processes assigned to it, and so there is a greater chance, for any process being assigned here, of having to share the processor with another process. So this processor should be last in any list. With this in mind the processor list can be constructed. The processors have the following numbers of processes able to be assigned to them-

PROCESSOR_1 3, PROCESSOR_2 2, PROCESSOR_3 1

and so the processor list for PROCESS_1 is

(PROCESSOR_3, PROCESSOR_2, PROCESSOR_1)

and the list for PROCESS_2 is

(PROCESSOR_2, PROCESSOR_1)

Given this ordering for processors, the ordering stragety used for store list of a memory element is to order on the throughput information. That is, the memory is assigned to each of the stores in turn and the throughput obtained from the resulting partial allocation map is used for sorting.

B) Nonhomogeneous architecture, where the processors can only access some of the stores. This is the case for the architecture of figure(7.1), where STORE_1 and STORE_4 are not accessible to PROCESSOR_2 and PROCESSOR_1 respectively. In this situation it was found to be better to order the store list of each memory element in the following manner. The stores are ordered so that the stores that are closest to the processors are first in the list. Here the closeness of a store is taken to mean the number of processors that can access the store. Thus a store that has only one processor accessing it will be closer to that processor than a store that is accessed by this processor and one other processor. By example, the store STORE_1 is closer to PROCESSOR_1 (figure(7.1)) than is store STORE_2. This is because the first store is only accessed by the processor, while the second store is accessed by both processors.

The rational for this choice is that a memory placed upon a close store is less likely to be subject to memory interference from the memory accessing patterns of other processors. Furthermore close stores are more likey to be directly accessible to the processor, and thus have faster access times, while distant stores are likely to be accessed via buses, and be slower to access.

Having ordered the store list in this way, the processor list of a process element is ordered using the throughput information.

The ordering of each store or processor list is carried out by actually obtaining the allocation map for each possible assignment. This is done by starting with the current partial allocation map and assigning the memory or process to each of its allowed stores or processors. A new partial map is obtained in each case and these are ordered using the techniques discussed above. An example is where an allocation has proceeded to where the first element in the process memory list, PROCESS_2, has been assigned. The next element is to be MEMORY_4 and it may be assigned to the stores as shown-

MEMORY_4 -> [STORE_1, STORE_2, STORE_3]

(ignoring other constraints). From here the partial map corresponding to each of these stores is constructed, and the through throughput computed. Thus

Partial maps

```
PROCESS_2 -> [ PROCESSOR_1 ]
MEMORY_4 -> [ STORE_1 ]            throughput = 2x

PROCESS_2 -> [ PROCESSOR_1 ]
MEMORY_4 -> [ STORE_2 ]            throughput = 1x

PROCESS_2 -> [ PROCESSOR_1 ]
MEMORY_4 -> [ STORE_3 ]            throughput = 1x
```

where the throughput is in multiples of some constant x. (To obtain this throughput pattern it is assumed that the store STORE_1 is faster than the other stores.) From this, the stores may be ordered, which will produce

(STORE_1, STORE_2, STORE_3)

In this case the order is the same for both methods.

The throughput for each map is always calculated, irrespective of which method is used to order the resources. It is used to discard any map whose throughput is less than the throughput of the best final map produced so far. To illustrate, the example immediately above is used. If it had so happened that in some previous search a final mapping had been found, then its throughput will have been kept. If this was 1.5x then the two partial maps above for the stores STORE_2 and STORE_3 will be excluded from any further searches. They have a throughput that is less than 1.5x, so no matter what happens they will never generate a solution that has a better throughput than the one already found.

After this elimination stage, the first element in the newly ordered list is choosen and the map that was derived for this is used as the basis of the next search step. The other maps are not discarded but are retained and used for subsequent assignments at this search level, after backtracking. This can be regarded as a one level deep breadth first search performed at each search step to calculate and order the most profitable search paths to follow next.

(7.4) SUMMARY

In summary the search pattern choosen is a modified depth first search with backtrack. This is based upon the method of implicit enumeration. The general memory interference model and constraint reduction are used at each partial allocation to reduce the number of branches traversed. The order of the elements is choosen to obtain fast and legal allocations as early as possible in the search. In the following sections the constraint reduction rules are described in depth.

(7.5) CONSTRAINT REDUCTION

Constraint reduction involves examining each process and memory element. If there are any restrictions derivable from the information in the current mapping then this can be used to reduce the element's allocation constraint. Ideally, using a perfect constraint reduction algorithm, such restrictions would result in an optimal final mapping without the necessity for any backtracking searches. Unfortunately such an algorithm is not known, instead it is a case of constructing a set of examination and reduction rules which can be applied easily.

The rules that have been investigated utilize the following information

Memory and store size information.

Accessibility information.

The same and different constraint information.

From this information it is possible to derive rules to

Reduce the constraints to prevent the creation of illegal constraints.

Reduce the constraints by removing redundant allocations.

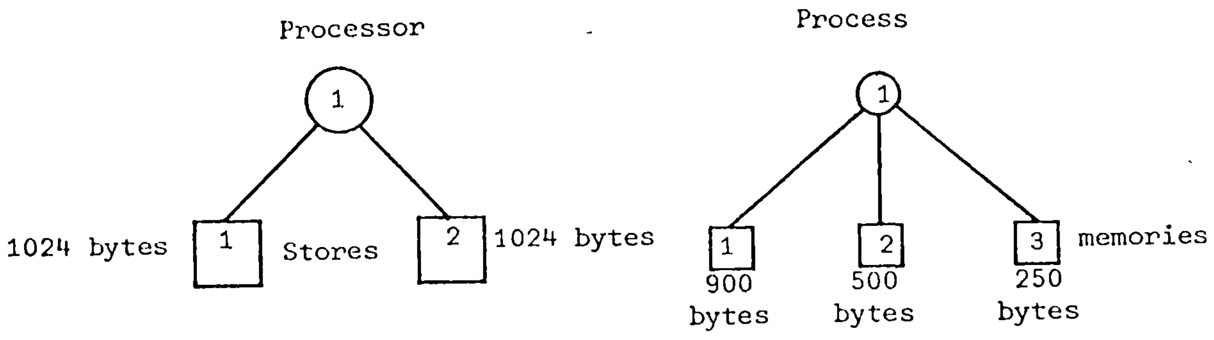


Figure 7.5

A discussion of these techniques, as demonstrated in the thesis research, is given in the following.

(7.5.1) CONSTRAINT REDUCTION USING STORE SIZE INFORMATION

In assigning memory elements to stores, only enough space to hold the memory is used. Also successive memory assignments are allocated to successive regions in the store. Furthermore there is no provision in the allocator algorithms for a memory element to straddle a store boundary. Thus the unused space in a store for a particular partial mapping is easily obtained, and only memories that will fit into this space are able to be assigned to that store. Consequently this can be used to restrict the allowable store set of a memory constraint. This is done by calculating the intersection of the allowable store set of each memory element with the set of all the stores that currently have enough space to accept this memory. In the demonstration program this is called the ALLOWED_MEMORY_SIZE constraint reduction operation.

As an example consider the computer architecture and program in figure(7.5). The initial constraints for the memories are

```
MEMORY_1 -> [ STORE_1, STORE_2 ]
MEMORY_2 -> [ STORE_1, STORE_2 ]
MEMORY_3 -> [ STORE_1, STORE_2 ]
```

If the MEMORY_1 element is assigned to the STORE_1 resource, then the remaining free space in this store is only 124 bytes. This is not

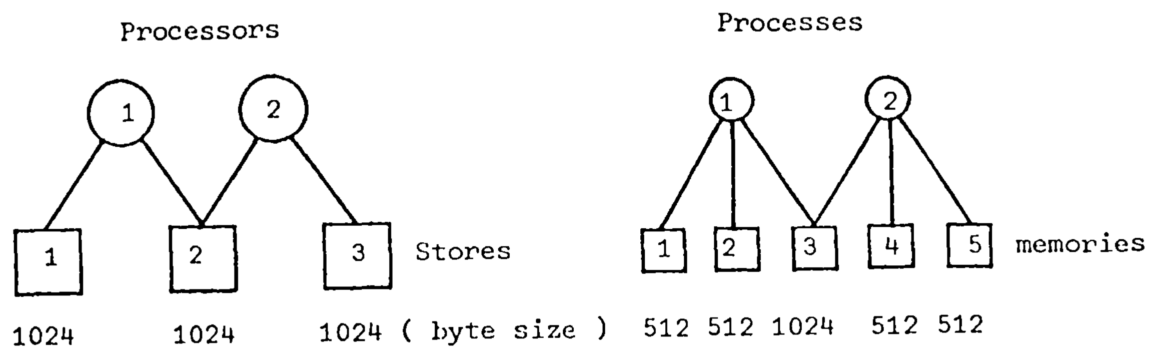


Figure 7.6

enough for the other two memories, and so their constraints will be reduced down to

```
MEMORY_2 -> [ STORE_2 ]
MEMORY_3 -> [ STORE_2 ]
```

A similar situation exists for the allocation of processes to processors. A processor accesses a number of stores, and these will have varying amounts of unused space. The processor may also have a number of processes already fixed to it, and these processes may access memory that has not yet been fixed. Thus the total unused store space available to a processor is found by summing its unused store and subtracting the space that will be occupied by all of the unfixed memory of those fixed processes. Thus any unassigned process whose combined unassigned memory size is larger than this total unused store space will not be able to be assigned to this processor. Note that there is still no guarantee that the process will fit even if the unused store space is big enough, since here the sizes of the individual memories and stores are not taken into account.

This constraint reduction is performed by obtaining the set intersections in the same fashion as the memory size constraint reduction. In the program this operation is given the name ALLOWED_PROCESS_SIZE.

To demonstrate this reduction, the architecture and program of figure(7.6) is used. Assume an initial assignment of PROCESS_1 to PROCESSOR_1 and MEMORY_3 to STORE_2. The nonfixed memory of PROCESS_2 will now be MEMORY_4 and MEMORY_5, giving a nonfixed memory size of

(512+512). The size of the unused store space attached to PROCESSOR_1 will be 1024 from STORE_1. However the two memories MEMORY_1 and MEMORY_2 that PROCESS_1 accesses have to be allocated to this store space, and so the size of the unused store space of PROCESSOR_1 is (1024-512-512), which is zero. Thus PROCESS_2 with a nonfixed memory size of 1024 can not go to this processor. Thus its allocation constraint is modified by

```
PROCESS_2 -> [ PROCESSOR_1, PROCESSOR_2 ] - [ PROCESSOR_1 ]
          -> [ PROCESSOR_2 ]
```

which in this case fixes the process.

Another constraint reduction based upon the detection of size mismatches is concerned with the Same_Constraints. If there is a Same_Memory_Constraint like

```
[ MEMORY_1 , MEMORY_2 ] -> SAME
   [ [ STORE_1 , STORE_2 ] , [ STORE_3 , STORE_4 ] ]
```

then the summation of all of the unused space in each of the same target sets (there are two in this example, one containing STORE_1 and STORE_2 and the other containing STORE_3 and STORE_4) has to be greater than or equal to the size of all the nonfixed memory elements in the same constraint. Otherwise these memories will not fit into the stores of the same target set as required by the constraint. For example, if STORE_1 and STORE_2 do not have enough combined space to fit all of the currently unfixed memory in the memory set, then this target store set can be eliminated and thus the constraint becomes

```
[ MEMORY_1 , MEMORY_2 ] -> SAME [ [ STORE_3 , STORE_4 ] ]
```

This is named SAME_MEMORY_SIZE constraint reduction in the demonstration program. An analogous operation called SAME_PROCESS_SIZE is also provided, which works in a similar way on Same_Process_Constraints.

Finally there is one more reduction operation based upon the examination of memory and store sizes which is applicable. This uses a set of memory elements that can be allocated to a set of store resource elements. If both of these sets are choosen so that none of the memory elements are assignable outside of this store set, then the total size of all the unfixed memory elements has to be less or equal to the total size of the unused space in this store set. If not then any further attempts to assign the memory elements will be bound to fail. The current map search can be terminated at this point. An example map allocation where this is applicable can be

```
MEMORY_1 -> [ STORE_1, STORE_2 ]
MEMORY_2 -> [ STORE_1, STORE_2, STORE_3 ]
MEMORY_3 -> [ STORE_2 ]
```

Here the store set is [STORE_1, STORE_2, STORE_3] and the memory set is [MEMORY_1, MEMORY_2, MEMORY_3].

Partitioning the memory elements into sets like these is easily achieved. To start, any memory element not yet fixed is selected, and the set of all its allowable store is obtained. Then from this store set the set of all unfixed memory that can be assigned to this is derived. If this memory set is identical to the starting memory set then a partition has been found. If not the process is repeated and eventually a partition will be found. Given such a partition, it is a simple step to check the sizes of the memories and stores. If there are any memories left over that are not in any partition found so far, then this algorithm is repeated.

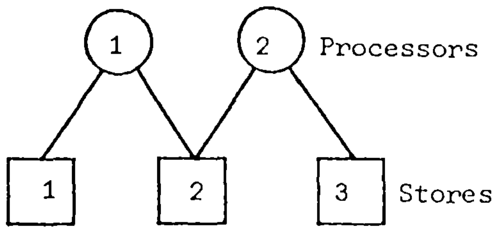
As an example the partition sets for the constraints above will be obtained. The starting point is taken to be the first constraint

```
MEMORY_1 -> [ STORE_1, STORE_2 ]
```

Now the memories that can be allocated to STORE_1 are [MEMORY_1, MEMORY_2] and the memories that can be allocated to STORE_2 are [MEMORY_1, MEMORY_2, MEMORY_3]. The union of these gives

```
[ MEMORY_1, MEMORY_2, MEMORY_3 ]
```

The stores that these may go to are [STORE_1, STORE_2], [STORE_1, STORE_2, STORE_3] and [STORE_2] respectively. The union of these sets gives



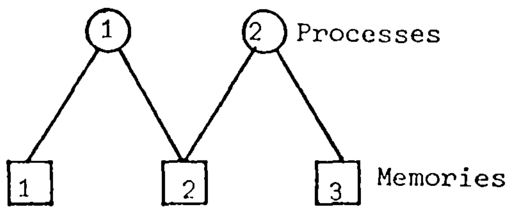


Figure 7.7

[STORE_1, STORE_2, STORE_3]

This activity is repeated, and will give the same two sets. Thus the partition sets have been found.

This method is similarly applicable to processes. In the program these two operations are known as MEMORY_PARTITION_SIZE and PROCESS_PARTITION_SIZE.

(7.5.2) CONSTRAINT REDUCTION BASED UPON ACCESSIBILITY

Any process in the program which accesses a particular memory must be able to reach this memory when the program is running on the architecture. Thus the store to which this memory is assigned must be accessible by the processor onto which the process has been assigned. Conversely the processor to which a process is assigned must also be able to access the store to which a memory of this same process is assigned.

This condition is used as the basis of a constraint reduction operation. If this constraint is to be applied to a memory element, then the first step is to find the set of all processes that access this memory. The set of all processors to which these processes may be assigned is found by using this process set. Next the set of all stores accessed by all of these processors is obtained. The resulting set of stores represents all the stores to which the memory can be assigned. The set intersection of this with the current set of allowable stores

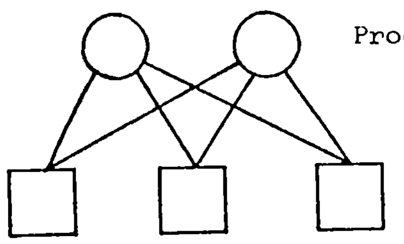


Figure 7.8

for this memory will then provide the new and possibily reduced allowable store set.

This constraint reduction proceeds similarly for an initial process element. These operations are known as the ALLOWED_MEMORY_SET and the ALLOWED_PROCESS_SET reduction operations. As an example the computer and program of figure(7.7) are used. If PROCESS_1 is assigned to PROCESSOR_1, then MEMORY_1 has to be assigned so that it is accessible from PROCESSOR_1. The only stores satisfying this are [STORE_1, STORE_2] and so the constraint on MEMORY_1 is

MEMORY_1 -> [STORE_1, STORE_2]

Under the circumstances where the computer architecture design is such that every processor is able to access every store (either directly or indirectly via buses), then this constraint reduction operation will never result in any changes in the constraints. Thus the application of this operation may be avoided as an implementation efficiency measure.

The architecture of figure(7.8) is a typical example. No matter where a process may be positioned, it can access every store and so there is no restrictions on the allowed store sets. The same applies to allowed processor sets.

(7.5.3) PROXIMITY CONSTRAINT INFORMATION

A Same_Constraint like the following

```
[ MEMORY_1 , MEMORY_2 ] ->
   SAME [ [ STORE_1 , STORE_2 ] , [ STORE_3 , STORE_4 ] ] ;
```

requires that both MEMORY_1 and MEMORY_2 must be allocated either to the stores in first target set or to the stores in the second target set. However if it so happens that any one of the memories can not be assigned to any of the stores STORE_1 and STORE_2 of the first target set, then this Same_Memory_Constraint can be modified by eliminating this now redundant target store set. This results in

```
[ MEMORY_1 , MEMORY_2 ] -> SAME [ [ STORE_3 , STORE_4 ] ] ;
```

Such reductions are equally applicable to both Same_Process_Constraints and Same_Memory_Constraints, and are known as SAME_PROCESS_SET_INDIVIDUAL and SAME_MEMORY_SET_INDIVIDUAL reductions.

A similar kind of operation is possible with Different_Constraint sets. Given

```
[ MEMORY_1 , MEMORY_2 ] -> DIFFERENT
   [ [ STORE_1 , STORE_2 ] , [ STORE_3 , STORE_4 ] ] ;
```

then if none of the memories can be assigned to the stores STORE_1 and STORE_2, this target store set may be removed from the constraint. Again this is applicable to processes, and these two operations have the names DIFFERENT_MEMORY_SET_INDIVIDUAL and DIFFERENT_PROCESS_SET_INDIVIDUAL.

The difference in these operations upon the same and the different constraint arise because a Same_Constraint specifies that all of its elements are to be allocated into the same resource target set. Whereas a Different_Constraint specifies that only one element is to be assigned to any one target set.

These operations will reduce the constraint sets of the proximity constraints. There are several reduction operations that work in the opposite direction, and reduce the constraints of elements based upon the information in the proximity constraints. This kind of reduction is demonstrated in the following,

```
SAME [ MEMORY_1 , MEMORY_2 ] -> SAME
   [ [ STORE_1 , STORE_2 ] [ STORE_4 , STORE_5 ] ]

MEMORY_1 -> [ STORE_1 , STORE_2 , STORE_3 , STORE_4 , STORE_5 ]
MEMORY_2 -> [ STORE_1 , STORE_2 , STORE_3 , STORE_4 , STORE_5 ]
```

In this example the Same_Constraint restricts the two memory elements to being either on stores 1 and 2, or stores 4 and 5. STORE_3 is never possible, and so this store can be removed from the two following memory constraints. In general this is achieved by finding the union of all of the constraint sets in the Same_Constraint, and then obtaining the intersection of this with the memory constraint set. This produces the new memory constraint set. The above reduction operations are equally applicable to processes and memories, and to different and same proximity constraints. Their names, as used in the implementation, are SAME_PROCESS_SET, SAME_MEMORY_SET, DIFFERENT_PROCESS_SET and DIFFERENT_MEMORY_SET.

A reverse activity, of reducing the same sets to correspond to the memory element constraint sets, is also possible. For example consider

```
[ MEMORY_1 , ... ] -> SAME
   [ [ STORE_1 , STORE_2 , STORE_3 ] ... ]
MEMORY_1 -> [ STORE_1 , STORE_2 ]
```

Here the STORE_3 resource can never be assigned to the MEMORY_1 element and so can safely be eliminated from the Same_Constraint. However the proximity constraints are only used to restrict the element constraints, they are not used to generate any element constraint directly. Thus it turns out that any superfluous resources in the constraints sets like in the above do not matter and so their reduction is not carried out.

To make this clearer, consider an example of a SAME_MEMORY_SET constraint reduction. It initially starts with the constraints

```
[ MEMORY_1, MEMORY_2 ] ->
   SAME [ [ STORE_1, STORE_2 ], [ STORE_3, STORE_4 ], [STORE_5] ]
MEMORY_1 -> [ STORE_1, STORE_2, STORE_5, STORE_6 ]
MEMORY_2 -> [ STORE_3, STORE_4, STORE_5, STORE_6 ]
```

The SAME_MEMORY_SET constraint will reduce the constraints for the memories to

```
MEMORY_1 -> [ STORE_1, STORE_2, STORE_5 ]
MEMORY_2 -> [ STORE_3, STORE_4, STORE_5 ]
```

since STORE_6 is not in the Same_Constraint. If now it is assumed that some other constraint results in the STORE_5 resource being removed from the memory constraints,

```
MEMORY_1 -> [ STORE_1, STORE_2 ]
MEMORY_2 -> [ STORE_3, STORE_4 ]
```

then this store could also be removed from the Same_Constraint. However irrespective of whether or not STORE_5 is present, the SAME_MEMORY_SET constraint reduction will not influence the memory constraints in any way. Thus there is no need to remove it.

Finally there are some extra constraint reductions applicable only to the Different_Constraints. Starting with a constraint of the form

```
[ PROCESS_1 , PROCESS_2 ] -> DIFFERENT
   [ [  PROCESSOR_1 , PROCESSOR_2 ] , [ PROCESSOR_3 ] ]
```

If the PROCESSOR_3 target set is removed in some other constraint reduction step, there will be only one target set remaining. Consequently the two processes can not be assigned to different targets sets and so the current mapping will fail. This reduction operation, of counting and comparing the number of elements, is valid for both process and memory Different_Constraints and is known as DIFFERENT_PROCESS_NUMBER and DIFFERENT_MEMORY_NUMBER.

Alternatively, using the same example, if in some previous search move the PROCESS_2 element had been assigned to PROCESSOR_3, then any other processes in this constraint can not be assigned to the same target set containing this processor. This fact is recorded by removing the PROCESS_2 element from the process set and removing the target set containing PROCESSOR_3. Thus the Different_Constraint set now left is

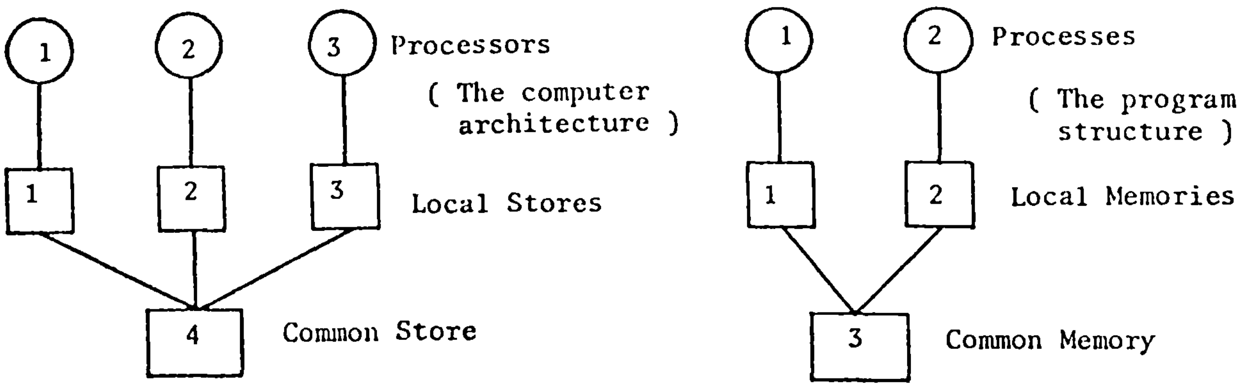


Figure 7.9

[PROCESS_1] -> DIFFERENT [[PROCESSOR_1 , PROCESSOR_2]]

This operation and its partner are called DIFFERENT_PROCESS_REMOVE and DIFFERENT_MEMORY_REMOVE.

(7.5.4) ELIMINATION OF SYMMETRICAL SEARCHES

Consider a computer architecture of three identical processors. Each processor has its own identical local memory and all processors access a common global memory. To be allocated to this architecture is a program with two processes, each accessing a local memory and both processes accessing a common memory. These are depicted in figure(7.9).

If the search method so far described is used, then the first process of the program will be assigned to one of the processors, followed by an attempt to assign all of the others. At the completion of this search a successful assignment may have been found, in which case it will have been recorded. The search will then proceed by reassigning the first process to the second processor, and carrying out the search again to find a new assignment. This would be repeated and another assignment found for the third processor. In this situation, however, the three processors and their memory structures are identical. The final map produced at the completion of any of the three searches can only have identical efficiencies. Thus the subsequent two searches are unnecessary. The first process can be correctly assigned to only one of the processors without eliminating any significant search branches.

In the following the detection of such symmetries or redundancies in the search, and their removal, is described. This is divided into the topics

Under what conditions do symmetries exist?
How can they be detected?
How can they be eliminated from the search?
How can the detection of symmetries be made more efficient?

(7.5.4.1) DEFINITION OF A SYMMETRICAL ALLOCATION.

A symmetrical allocation situation exists for a program element if two or more of its allowable resource elements are judged to be equivalent. The conditions for a pair of resource under which this equivalence exists are

They are the same kind, either BANK, BUS, PROCESSOR or STORE.

They have identical properties, depending on the kind. For example stores must have identical rewrite and access times and be of the same size.

They are connected to other resources in an identical pattern. For example if one processor has access to two stores, then any other equivalent processor will also have access to two stores.

They are connected to equivalent resources, that is, in the previous example, the two stores of the first processor need to be equivalent to the two stores of the second processor.

Finally if two processors (or stores) are identical then the sets of processes (or memories) that can be allocated to these resources must be identical.

To demonstrate these conditions, the simple computer structure defined at the start of this section is used.

The three processors and the three local stores have identical properties. Thus the three processors of figure(7.9) are identical since they are the same kind, have the same properties, are each connected to one local store and one global store, and each local store is also equivalent. Similarly the three local stores are equivalent. It can be seen that the definition for equivalence is recursive, since the processors are only equivalent if their attached stores are, and the stores are only equivalent if their accessing processors are.

The last condition listed for equivalence has not been mentioned in this example. To demonstrate this condition, consider the program of figure (7.9). If the user had imposed the constraint that PROCESS_1 is only allowed to be allocated onto either PROCESSOR_1 or PROCESSOR_2, then the three processors are no longer equivalent. This arises from the observation that if PROCESS_2 is fixed to PROCESSOR_3 then it can never be in the same processor as the other process. If PROCESS_2 is allocated to PROCESSOR_1 or PROCESSOR_2, then it may eventually be assigned to the same processor as the other process. In these two cases, the execution speeds of the final map allocations will be different.

Thus in this situation only the processors PROCESSOR_1 and PROCESSOR_2 are equivalent. This therefore implies that only stores STORE_1 and STORE_2 are equivalent, since now STORE_3 is accessed by a processor not equivalent to the processors accessing the first two stores.

(7.5.4.2) DETECTING EQUIVALENCE

A set of equivalent resource elements is called an equivalent partition set. To find these sets the whole resource graph is examined. For any architectures four initial partitions are always produced, one each for all the bus, bank, store and processor resource elements in the resource graph. These sets are then split up into further separate sets on the basis of information such as the cycle speeds and store sizes of each particular architecture. This information is called nontopological information. Any resource element that ends up in a partition set by itself has no equivalents. If there are any nonsingleton partition sets left after this stage, then the sets are further partitioned using topological information. Topological information is information gained from considering the connection patterns of the computer architecture.

To achieve the topological partitioning, every resource element that still has a chance of being equivalent to some other is examined. A list of all the other resource elements that it accesses or is accessed by it is produced. These attached resource elements are described by the current partition they belong to. This allows the elements of the same partition to be compared on the basis of their attachments, and any two elements that differ in this are no longer in the same partition and are separated. This comparison of every likely resource element is repeated until no further partit ion reductions are made, or until every resource element is in its own partition. The resulting partition sets contain the equivalent resources.

As an example of this the step by step derivation of the equivalence partitions of the architecture in diagram(7.9) is given. This exercise assumes that the user has imposed a constraint of fixing a process to the PROCESSOR_1 resource.

At first the partitions are

1. [PROCESSOR_1 , PROCESSOR_2 , PROCESSOR_3]
2. [STORE_1 , STORE_2 , STORE_3 , STORE_4]

where the numbers represent an arbitrary unique labeling of the sets.

The only nontopological information applicable here is the fact that PROCESSOR_1 already has a process element assigned to it (via the assumed user constraint). Therefore PROCESS_1 is different from both PROCESS_2 and PROCESS_3, thus the new partitions are

1. [PROCESSOR_1]
2. [PROCESSOR_2 , PROCESSOR_3]
3. [STORE_1 , STORE_2 , STORE_3 , STORE_4]

Now the topological information is applied by constructing the attached sets. In the following list the resource element appears on the lefthand side. The set of attached resources that it accesses or is accessed by is in the middle. On the right is a list representation of this set containing the labeling of the partition set which the resource element belongs to.

```
PROCESSOR_1  [ STORE_1 , STORE_4 ]    ( 3 , 3 )
PROCESSOR_2  [ STORE_2 , STORE_4 ]    ( 3 , 3 )
PROCESSOR_3  [ STORE_3 , STORE_4 ]    ( 3 , 3 )
STORE_1      [ PROCESSOR_1 ]          ( 1 )
STORE_2      [ PROCESSOR_2 ]          ( 2 )
STORE_3      [ PROCESSOR_3 ]          ( 2 )
STORE_4      [ PROCESSOR_1 ,
               PROCESSOR_2 ,
               PROCESSOR_3  ]         ( 1 , 2 , 2 )
```

From this it can be seen that STORE_1 and STORE_4 are different from the other stores and from each other, so the new derived partition sets are

1. [PROCESSOR_1]
2. [PROCESSOR_2 , PROCESSOR_3]
3. [STORE_1]
4. [STORE_2 , STORE_3]
5. [STORE_4]

And so redoing the accessibility sets gives

```
PROCESSOR_1  [ STORE_1 , STORE_4 ]    ( 3 , 5 )
PROCESSOR_2  [ STORE_2 , STORE_4 ]    ( 4 , 5 )
PROCESSOR_3  [ STORE_3 , STORE_4 ]    ( 4 , 5 )
STORE_1      [ PROCESSOR_1 ]          ( 1 )
STORE_2      [ PROCESSOR_2 ]          ( 2 )
STORE_3      [ PROCESSOR_3 ]          ( 2 )
STORE_4      [ PROCESSOR_1 ,
               PROCESSOR_2 ,
               PROCESSOR_3  ]         ( 1 , 2 , 2 )
```

This now indicates that PROCESSOR_2 and PROCESSOR_3 are equivalent and that STORE_2 and STORE_3 are equivalent, with no other equivalencies existing. Since this agrees with the last derived partitioning, the process can terminate with this as the final partition.

(7.5.4.3) SPEEDING UP THE PARTITIONING OPERATION

This partitioning into equivalence sets can be speeded up. This is done by making use of the observation that generally there will not be any equivalent resource elements found. This particularly applies after the initial stages of the map allocation search, where the program elements already fixed to a resource will by definition make that resource no longer equivalent to any other resource. Thus the detection of nonequivalence as soon as possible is the best policy. This is achieved by initially only considering the nontopological information such as memory and store size and the like. This does not consume much time. If the target resource elements are not reduced to singleton partition sets, then the full partitioning operation has to be applied.

(7.5.4.4) PERFORMING THE CONSTRAINT REDUCTIONS

After producing the partition sets of a program element, the next step is to use these sets to reduce the elements constraint. Since the partition sets are produced by considering the entire architecture, they may contain resources to which the program element can not be allocated. These are removed at this stage by producing the intersection of the program elements allocation constraint set with the partition sets of the appropriate kind and then using these resulting sets. For example if a constraint is

```
PROCESS_1 -> [ PROCESSOR_1, PROCESSOR_2, PROCESSOR_3 ]
```

and the equivalence partition set is

```
[ PROCESSOR_2, PROCESSOR_3, PROCESSOR_4 ]
```

then this partition set is reduced to the set

```
[ PROCESSOR_2, PROCESSOR_3 ]
```

In this reduction, some of these partition sets may now be empty, representing sets of equivalent resource elements that the program element can not be assigned to. These are discarded, along with all partition sets containing only one element. From the remainder one set is choosen and used for the reduction. This will contain a collection of resource elements to which the program element may be assigned with equivalent effects, and so all but one of these resources may be removed. This is done by simply deleting one element from the partition and set subtracting the resulting set from the constraint. So for the partition of [PROCESSOR_2, PROCESSOR_3], one of the processors is removed, perhaps resulting in [PROCESSOR_3], and this is subtracted from the PROCESS_1 constraint above, giving

```
PROCESS_1 -> [ PROCESSOR_1, PROCESSOR_2 ]
```

At this stage other partition sets possessing more than one element may still exist. These can not be used to reduce the constraint straight away, since the first reduction may have interacted with other constraints to change the allocation of still other elements. Two resources are only equivalent if they have the same elements able to be allocated to them, and thus this interaction may result in two originally equivalent resources becoming nonequivalent. Therefore the entire symmetry detection operation is repeated for each reduction step.

(7.5.4.4.1) EXAMPLE SYMMETRY REDUCTION

A complete example of symmetry removal for the original problem in figure(7.9) is developed. It is assumed that there are now no user imposed constraints.

The starting point will be the constraint set for the PROCESS_1 element,

```
PROCESS_1 -> [ PROCESSOR_1 , PROCESSOR_2 , PROCESSOR_3 ]
```

Working on this, the symmetrical partition set produced will be

```
[ PROCESSOR_1 , PROCESSOR_2 , PROCESSOR_3 ]
```

In other words the three processors are regarded as being identical. In this example PROCESSOR_1 is choosen to be the one used, and so the constraint set for PROCESS_1 is

PROCESS_1 -> [PROCESSOR_1]

Now the redundant removal operation is repeated for the PROCESS_2 element, and will result in the symmetrical partition sets

[PROCESSOR_1] [PROCESSOR_2 , PROCESSOR_3]

Disregarding the first set since it only has one element, and reducing the second set by removing PROCESSOR_2, results in a partition set of

[PROCESSOR_2]

This is subtracted from the PROCESS_2 constraint allocation, and so the constraint now applicable is

PROCESS_2 -> [PROCESSOR_1 , PROCESSOR_2]

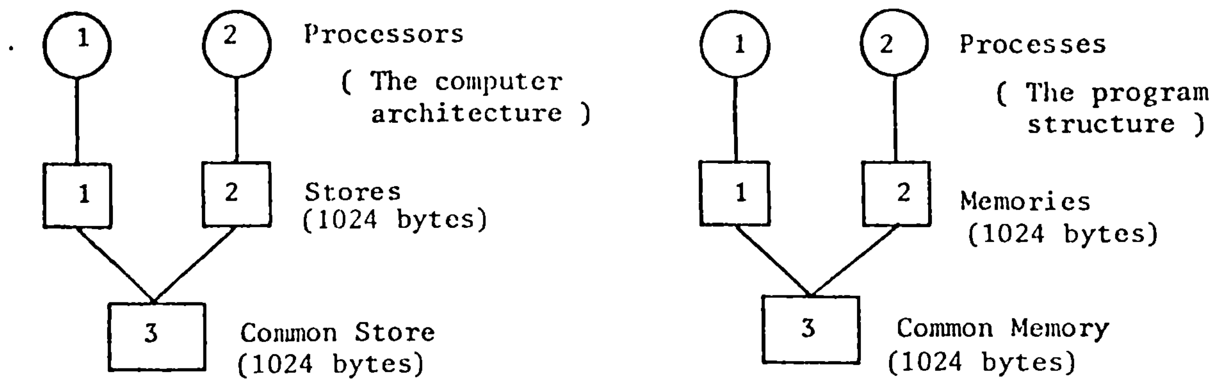
If there had been a third process element to be assigned, the partition sets for it will be

[PROCESSOR_1] [PROCESSOR_2] [PROCESSOR_3]

and so no symmetrical reduction would have been possible for it.

(7.5.4.5) RESTRICTIONS IN THE IMPLEMENTATION

One factor influencing the equivalence of resource elements not discussed in the above is their membership in Same_Constraints or Different_Constraints. In the general case these may be taken into account also, but the implementation was simplified by regarding any resource in such a constraint to be nonequivalent to any other.



1
2
Processors
(The computer architecture)
1
2
Stores (1024 bytes)
3
Common Store (1024 bytes)
1
2
Processes
(The program structure)
1
2
Memories (1024 bytes)
3
Common Memory (1024 bytes)

Figure 7.10

(7.5.5) CONSTRAINT REDUCTION PROPAGATION

So far the constraint reduction operations have mostly been developed independently of each other. However it will often happen that reducing the constraint of one element will thereby make possible the constraint reduction of other elements. In extreme cases the changes due to just one constraint reduction may propagate and result in all of the remaining elements being fixed and thus producing a final map allocation. More commonly the changes will either not propagate so far, or just result in the production of an illegal map.

In the literature one example of constraint propagation is given by [23]. This is for a graph problem whose vertices may take on values from a value set. Constraints are imposed upon the values that vertices connected by a common arc may take on. The problem is to derive a value mapping where all constraints are satisfied. This compares with current research where the constraint relations are imposed by the reduction operations, with the goal of having all constraints satisfied corresponds to a legal mapping.

As a demonstration of change propagation the following example has been constructed, using the architecture and program depicted in figure(7.10).

The reduction operations are carried out as follows.

At the start the constraints are

```
   PROCESS_1 -> [ PROCESSOR_1 , PROCESSOR_2 ]
   PROCESS_2 -> [ PROCESSOR_1 , PROCESSOR_2 ]
   MEMORY_1 -> [ STORE_1 , STORE_2 , STORE_3 ]
   MEMORY_2 -> [ STORE_1 , STORE_2 , STORE_3 ]
   MEMORY_3 -> [ STORE_1 , STORE_2 , STORE_3 ]
```

The only applicable operation is the symmetry removal operation. If this is applied to PROCESS_1 first, then the constraint set for this element will be reduced to

```
   PROCESS_1 -> [ PROCESSOR_1 ]
```

and now no futher reductions based upon symmetry are possible.

Following this the ALLOWED_MEMORY_SET reduction operation will result in

```
   MEMORY_1 -> [ STORE_1 , STORE_3 ]
   MEMORY_3 -> [ STORE_1 , STORE_3 ]
```

because these memories are accessed by PROCESS_1 and when this element is fixed to PROCESSOR_1 the only stores accessible are STORE_1 and STORE_3.

From here the ALLOWED_PROCESS_SIZE reduction operation will produce

```
   PROCESS_2 -> [ PROCESSOR_2 ]
```

because the amount of space taken up by PROCESS_1 on PROCESSOR_1 is 2048. The total store space on PROCESSOR_1 is only 2048 and so PROCESS_2 can no longer fit there.

Now the ALLOWED_MEMORY_SET operation will reduce the constraint set of the MEMORY_3 element, since it has to be accessible to both PROCESS_1 and PROCESS_2, which are now on different processors. Thus

```
MEMORY_3 -> [ STORE_3 ]
```

which means that ALLOWED_MEMORY_SIZE will operate on the constraint sets of MEMORY_1 and MEMORY_2. This produces

```
MEMORY_1 -> [ STORE_1 ]
MEMORY_2 -> [ STORE_2 ]
```

and this completely fixes the program to the architecture without any searches being necessary. Of course in the general situation this rapid conclusion will rarely occur.

In the allocation program the change propagation is implemented by organizing the constraint reductions into passes. Each pass performs all of the required reduction operations, and a record is kept of all elements and proximity constraints which actually change. At the end of each pass this information is used as the basis for choosing which elements are to be examined in the next pass. This process is terminated when a pass does not generate any changes. Deciding which elements to inspect in the next pass are is fairly straightforward. For example, if a process is fixed to a processor, then there may be stores that this processor cannot access. Therefore the memories of the process can no longer be assigned to these stores. Thus in the next pass all the memories of all processes that have just been fixed need to be examined by the ALLOWED_MEMORY_SET constraint reduction operation. The complete list of such rules is described in greater detail in appendix(C).

(7.6) EXPERIMENTAL RESULTS

An implementation in Pascal was produced to demonstrate the allocation search algorithms. This implementation worked as expected in producing legal allocations from a reduced search space. However the reduction achieved in the search space was only sufficient to allow the optimal allocation of small programs.

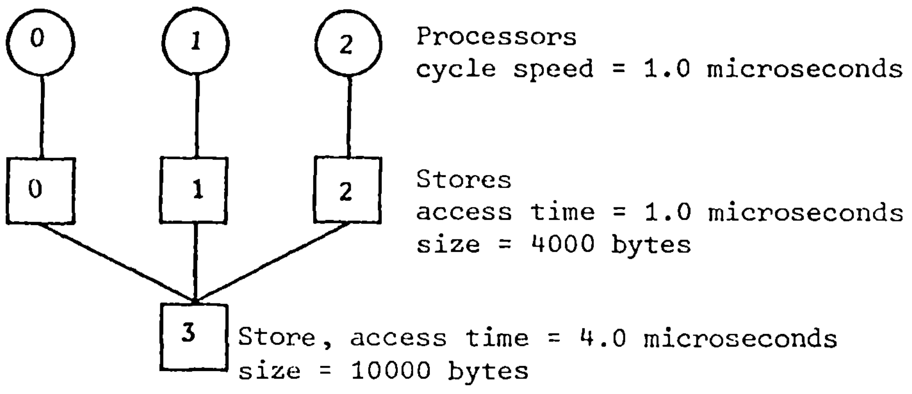

0
1
2
Processors
cycle speed = 1.0 microseconds
0
1
2
Stores
access time = 1.0 microseconds
size = 4000 bytes
3
Store, access time = 4.0 microseconds
size = 10000 bytes

Figure 7.11

(7.6.1) DEMONSTRATION PROBLEM

A typical demonstration problem was the allocation of a three process eleven memory program to a three processor four store computer architecture. In this architecture each processor has its own local store and all the processors access the fourth global store, as is shown in figure(7.11). The program used was the following-

Size of the memories, randomly generated.

MEMORY_0	237	MEMORY_1	848	MEMORY_2	1406	MEMORY_3	540
MEMORY_4	663	MEMORY_5	507	MEMORY_6	397	MEMORY_7	1277
MEMORY_8	2117	MEMORY_9	1348	MEMORY_10	1656		

Number of process to memory accesses, randomly generated such that half of the accesses are zero, and the other half are between 0 and 5000.

	PROCESS_0	PROCESS_1	PROCESS_2
MEMORY_0	1786	0	2214
MEMORY_1	582	3054	0
MEMORY_2	0	0	2825
MEMORY_3	1909	0	0
MEMORY_4	0	3232	0
MEMORY_5	2246	0	3763
MEMORY_6	4226	0	0
MEMORY_7	1324	2634	0
MEMORY_8	0	0	1061

MEMORY_9	4610	0	2123
MEMORY_10	0	1849	0

Here this information is presented in an array format for convenience.

For this example the total number of possible final allocations, ignoring all size and access constraints for the moment, is

$3^3 * 4^{11} = 10^8$ (approximately) Three processes to three processors
Eleven memories to four stores.

but the number of actual search steps performed by the allocation program was only 11 for the particular program specified. The total execution time required for this (on a Burroughs B6800) was 130 seconds or about 12 seconds per search step. Of this time 60 percent was spent within the simulator code obtaining throughput estimations. The allocation map found was

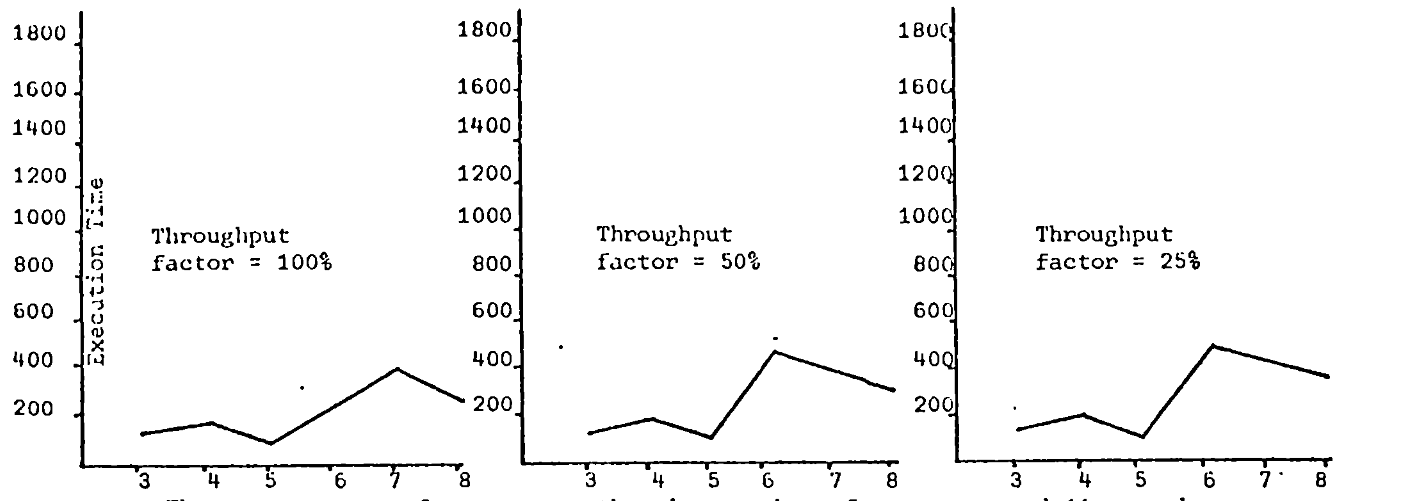
```
PROCESS_0 -> [ PROCESSOR_1 ]  PROCESS_1 -> [ PROCESSOR_0 ]
PROCESS_2 -> [ PROCESSOR_1 ]

MEMORY_0 -> [ STORE_1 ]       MEMORY_1 -> [ STORE_3 ]
MEMORY_2 -> [ STORE_1 ]       MEMORY_3 -> [ STORE_3 ]
MEMORY_4 -> [ STORE_0 ]       MEMORY_5 -> [ STORE_1 ]
MEMORY_6 -> [ STORE_1 ]       MEMORY_7 -> [ STORE_3 ]
MEMORY_8 -> [ STORE_3 ]       MEMORY_9 -> [ STORE_1 ]
MEMORY_10-> [ STORE_0 ]
```

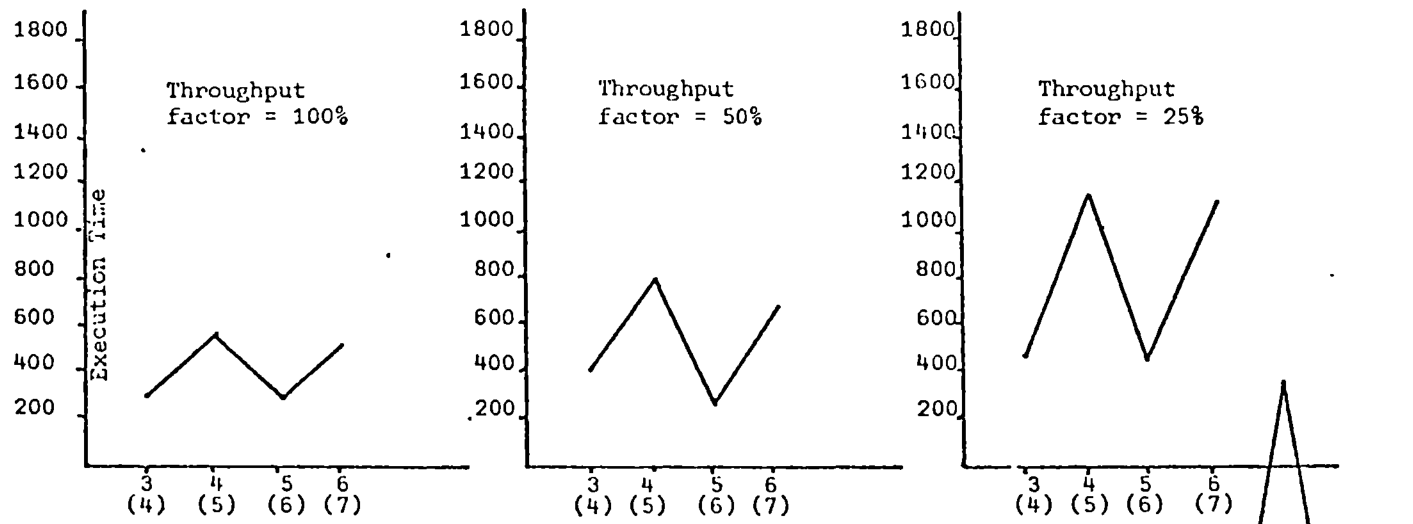
The throughput calculated for this was 14.2.

(7.6.2) LARGER PROBLEMS

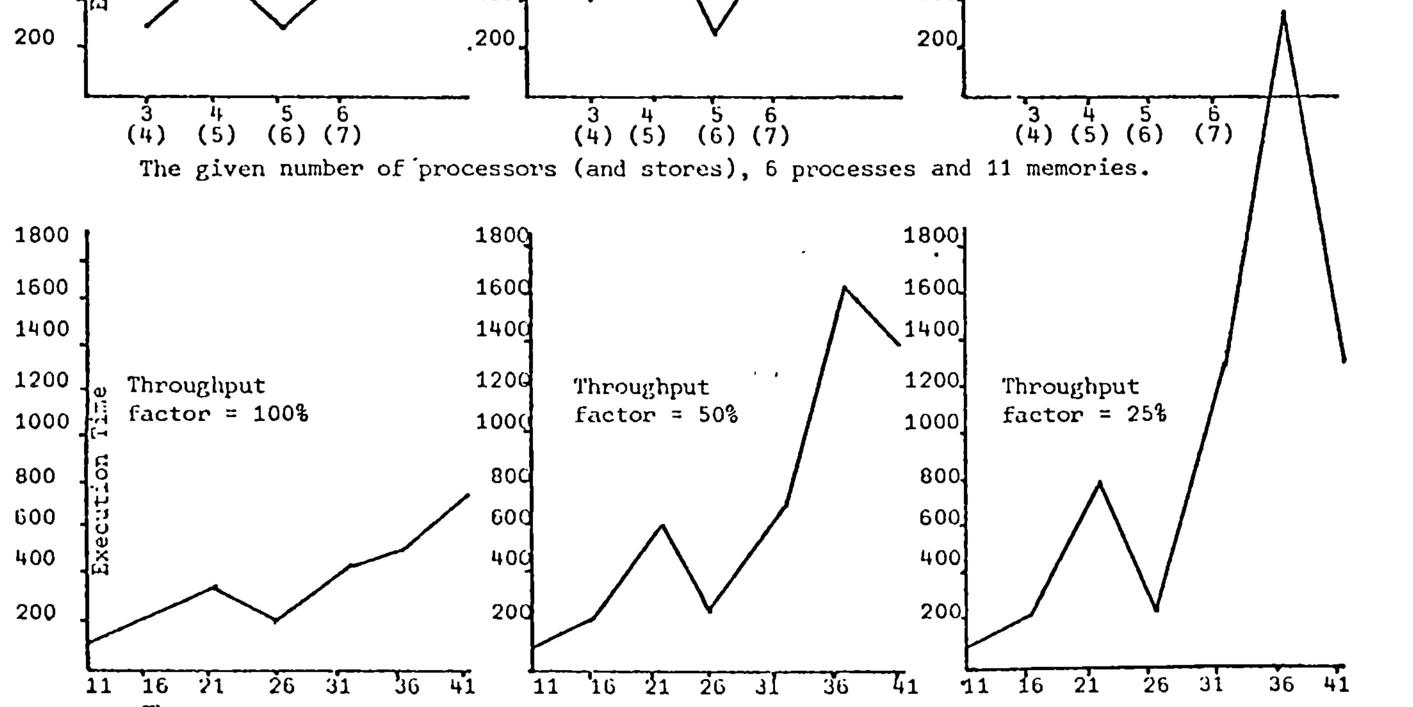
Unfortunately, for larger problems the allocation program does not complete the search in a reasonable time period. The graphs in figure(7.12) presents the times to completion for a range of computer architecture sizes and program sizes. The architecture used for this is shown in figure(7.13,left). The size of each memory of the program was choosen randomly, as were the number of cycles value between each process and memory.



Three processors, four stores, the given number of processes and 11 memories.



The given number of processors (and stores), 6 processes and 11 memories.



Three processors, four stores, three processes and the given number of memories.

Figure 7.12

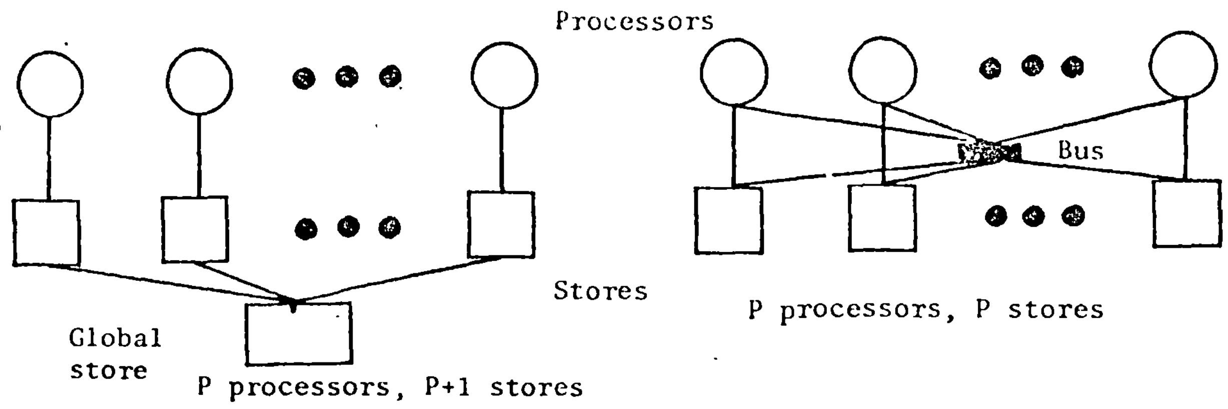


Figure 7.13

There were three sets of trials performed. They were for

1) An architecture with 3 processors and 4 stores. A program with 11 memories and a number of processes that is varied from 3 to 8.

2) An architecture where the number of processors is varied from 3 to 6, and the corresponding number of stores is varied from 4 to 7. A program with 11 memories and 6 processes.

3) An architecture with 3 processors and 4 stores. A program with 3 processes and a number of memories that is varied from 11 to 41 by fives.

Each of these three trials was performed with the throughput factor having the values 100 percent, 50 percent and 25 percent. This throughput factor specifies how much better a partial solution has to be in comparsion to an already obtained final solution before it is investigated any further. Thus if the throughput factor is 100 percent, only those partial solutions that have a throughput that is twice that of the throughput of the latest final map will be considered any further.

It can be seen that even for the smallest of these trials the execution time is high, and this increases with increasing problem size. It does not increase in a uniform manner, since the variations in the programs and architectures allow the search algorithms to perform better than usual in some cases.

Also presented below is a table showing the number of search steps needed for the searches shown in figure(7.12), and the maximum possible number of search steps.

Problem kind		Number of search steps of the trial with a throughput factor of 100	50	25	Maximum possible search length (approximately)
Problem 1	N = 3	8	8	10	10^3
3 processors	N = 4	7	7	7	10^4
4 stores	N = 5	4	4	4	10^4
11 memories	N = 6	14	19	23	10^4
N processors	N = 7	10	10	10	10^5
	N = 8	10	13	17	10^5
Problem 2	N = 3	14	19	23	10^4
N processors	N = 4	26	36	47	10^5
N+1 stores	N = 5	9	9	13	10^6
11 memories	N = 6	15	25	41	10^7
6 processes					
Problem 3	N = 11	8	8	10	10^3
3 processors	N = 16	11	11	11	10^4
4 stores	N = 21	24	42	56	10^6
3 processes	N = 26	16	16	20	10^7
N memories	N = 31	22	37	79	10^9
	N = 36	27	87	200	10^10
	N = 41	34	69	72	10^12

One reason why the allocation is so slow is the length of time needed for one step, which in these trials ranges from 15 to 40 seconds. Little attempt was made to improve the efficiency of the implementation code used for the search algorithms. It is therefore quite possible that this execution time per search step can be substantially improved.

For comparsion, the total number of possible search steps to find a solution using enumeration alone is also listed in this table. This number is computed by assuming that each process may be assigned to any processor, and that each memory may be assigned to only two stores. A memory can only be assigned to the global store, or to the store that is local to the processor that accesses that memory. This explains why the memory is not assumed to be assignable to all stores. It is readily seen that there are sizable reductions in the search space size for all trials.

The execution time of the allocation program will vary depending upon the following factors

1) Number of processors and stores in the architecture, and the number of processes and address spaces in the program.
2) The structure of the computer architecture.
3) The choice of the throughput factor.
4) The user specified constraints.

These are discussed in turn.

(7.6.3) SIZE OF THE ARCHITECTURE AND PROGRAM

Firstly the effects due to the numbers of the resource and program elements have already been displayed in figure(7.12). It is easy to see why the size of the problem will generally increase the execution time. For example, if there are P processors and C processes, then the number of combinations of process to processor allocations is P^C. In most cases the actual number of combinations will be less than this maximum due to restrictions placed upon the assignment of processes to processors. Some example restrictions will be due to user specified constraints, accessibility constraints and memory size constraints. It is for this use that the constraint reduction operations are provided.

(7.6.4) STRUCTURE OF THE COMPUTER ARCHITECTURE

Secondly, the structure of the computer architecture can be important. As one example, for P identical processors and C processes, the maximum number of different combinations possible for the process to processor allocations is

```
        C!                  if C <= P
        P! * P^(C-P)        if C > P
```

and these values are less than the value P^C used in the section above. This decrease is made possible by symmetry redundancy removal. Thus for C processes, C less than P, the first process will only have one processor to be assigned to, since all the others are identical. The second process will have 2 processors to be assigned, since there is one processor with a process already assigned, and P-1 other identical processors. Thus the total number of combinations is C!. If the number of processes is greater than the number of processors, then the remaining processes can be allocated to any processor, and so there are P^(C-P) possible combinations for these remaining processes. Thus when there are more processes than processors, the total number of combinations is P! * P^(C-P).

This reduction can be large and can mean the difference between a practical search and a computationally impossible one. However if the processors are not identical, as with differing processor cycle times, or if the stores are not identical, as with differing access times or store sizes, then the processors will no longer be identical. The symmetry reduction operations will not be possible. Therefore the more uniform the architecture the better the chances of obtaining a complete search.

There are other ways in which the computer architecture may influence the allocation program time. In figure(7.13) the number of processors is the same and the number of stores is almost the same for both architectures. However, given an initial process to processor allocation, the choice of possible stores for the address spaces of the processes in the first architecture is much more limited in comparsion to the second architecture. In the second architecture each store is accessible to each processor, and so even after a process has been fixed to a processor, there are no extra constraints applied upon its memories. The execution time difference can be seen in the table below. In each example pair here the two computer architectures have the same number of processors, and there are the same number of processes and address spaces in the program. As expected the time for the bus architecture is longer.

Problem kind	Number of search steps with a throughput factor of 100	50	25
Architecture 1, problem 1	130	130	160
Architecture 2, problem 1	130	160	310
Architecture 1, problem 2	220	220	220
Architecture 2, problem 2	380	380	380
Architecture 1, problem 3	170	170	170
Architecture 2, problem 3	160	200	>1000

For this table, Architecture 1 is that in figure(7.13,left),
Architecture 2 is that in figure(7.13,right),
Problem 1 has 3 processes and 11 memories,
Problem 2 has 3 processes and 16 memories,
Problem 3 has 4 processes and 11 memories.

In some circumstances the computer architecture may allow the program size to be increased with only a linear degradation in the execution time of the allocation. This occurs in special cases where it becomes possible to divide the program and architecture into separate sub problems and to solve these independently. This is most likely to happen where there is in effect two different kinds of computer architectures linked together. An example is shown in figure(2.10). Here the picture processor has several general purpose processors, with their own stores. As well there are the special purpose picture processing computer modules. In this circumstance the structure of the program will be written to reflect this design. Thus the main processes of the program will only run on the general purpose processors, and the picture processes will run on the special purpose processors. This division would be specified by the use of user constraints.

(7.6.5) THE CHOICE OF THE THROUGHPUT FACTOR

The allocation search will generally not find the theoretical optimal allocation mapping with respect to the throughput, but it will produce a result that can be made arbitrary close to it. How close is determined by the throughput factor. This gives the percentage by which the throughput of any subsequent solutions must exceed the throughput of the incumbent solution before they are investigated. If the throughput factor is set at 5 percent, then many more solutions will have to be examined than if only 100 percent precision is needed. This arises because there will generally be a larger number of solutions that vary only slightly in this throughput estimation.

The accuracy of the throughput estimation itself will also be important. If the throughput for an initial allocation map at the start of a search is close to the final optimal throughput, then fewer partial solutions will be examined. This is most clearly seen in an example where the throughput factor is set at 100 percent. If the initial throughput estimation is within a factor of 2 of the final optimal throughput, then the allocation will generally be able to derive the first solution without backtrack. Thereafter, since the throughput of this is within 100 percent of the initial throughput, no other solutions need be examined.

This also demonstrates another way in which the computer architecture structure may determine the search length. Some architectures will produce a better initial throughput estimation than others. The throughput of an initial unallocated map is found by assuming the program is mapped to an architecture that is ideal for it. Thus the further away from such an ideal machine the actual computer architecture is, the more inaccurate will the initial throughput guess be.

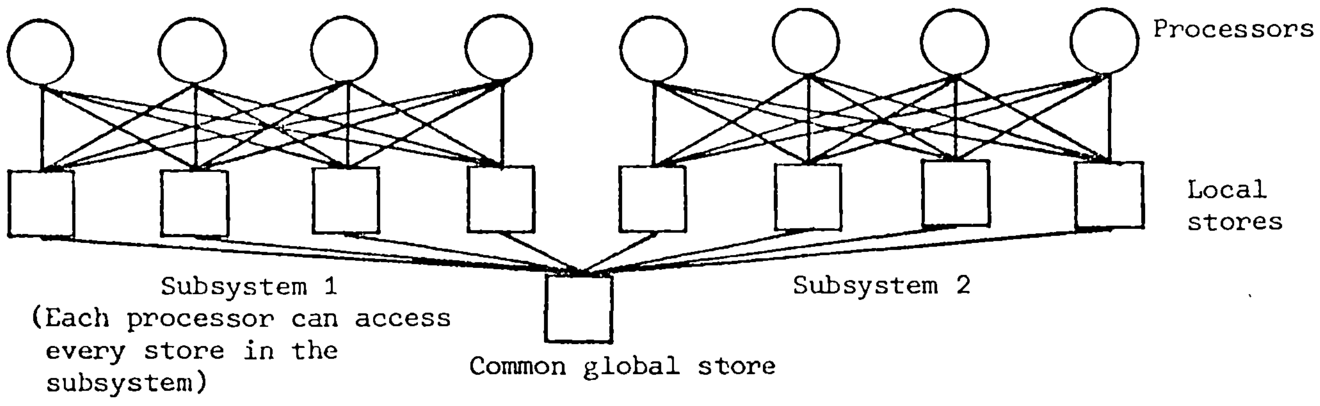


Figure 7.14

(7.6.6) USER IMPOSED CONSTRAINTS

Lastly, the user constraints may impact upon the search length. User imposed constraints may effect the allocation by

A) Changing the size of the solution space that needs to be searched.

B) Changing the length of the search needed to find the solutions.

None of the user constraints will ever increase the size of the solution space, however some constraints may increase the search length, and others may reduce it.

The size of the solution space is determined by the number of program elements that need to be allocated, and the number of resources that may be choosen for these. No constraints can increase either of these, and so constraints can never increase the solution space size. However constraints can reduce the number of allowed resources for each program element, and so they may certainly reduce the size.

Unfortunately decreasing the solution space will not always reduce the search length. If the search is to cover as little as possible of the solution space, and still implicilty examine the whole space, then the various means developed for reducing the search length must work to their best ability. These are the constraint reduction and symmetry redundancy removal operations, the heuristic search ordering techniques and the throughput estimation algorithms. The imposition of process to processor and memory to store constraints will generally not degrade their performance. An exception may be the symmetry redundancy removal operations. Constraining the elements to reduced resource target sets may make previously identical elements different. Thus this may inhibit symmetry reductions. In general the nonproximity constraints will reduce the search length. The judicious use of these may make the allocation larger problems feasible, with only minimal effort from the user.

As an example of this, consider the architecture of figure(7.14). This can be regarded as being two separate computer subsystems able to communicate with each other by the common global memory. Program allocations to this architecture may be performed in the same way as for any other architecture. Alternatively, if a programmer is writing a program specifically for this computer structure, then to achieve the best results it is probable that the program structure produced will reflect this structure. That is there will be two separate subsystems of processes, and these will communicate via common code and common variables having a small address space size. Thus in this circumstance the programmer can impose the constraints that the processes of one subsystem of the program are to be allowed only to the processors of one subsystem of the architecture, and similarly for the other subsystems. Little extra effort is required of the programmer for this, since the knowledge to achieve this is implicit in the program design. Therefore the complete allocation problem resolves into two smaller allocation problems of allocating a half sized program to a half sized architecture.

The imposition of proximity constraints will, however, degrade the performance of both types of constraint reduction operations. They impose higher level constraints between the individual process and memory constraints. Thus the constraint reduction of nonproximity constraints can no longer proceed independently but will interact. The action of these constraints upon the search will be to arbitrary remove some final mappings from the search space. This happens when the final mappings violate the proximity constraint. Since the implementation does not order its searches to take this possibility into account, then these reductions may occur at any position in the search. If they occur at a shallow level, then not much time will be spent in finding and eliminating the map allocations prohibited by these proximity constraints. If however these constraints are applied at points deep in the search, then a large amount of time may be wasted in backtracking up the search tree to try new searches .

(7.6.7) MAXIMUM PROBLEM SIZES

As indicated by figure(7.12) the practical maximum for a complete search with this type of architecture is about 4 processors and stores, for small programs of about 4 processes and 40 address spaces. Similar times apply to other styles of architecture.

In almost all cases the allocation program finds an initial solution straight away with little or no backtracking. Thereafter no better solutions are found, or the subsequent solutions that are found are generally not significantly better. This good behaviour is partly a result of the heuristically provided search order, and it also arises because only uniform architectures are used in the examples. This behaviour allows the use of the allocation program for larger problems, even when it does not complete a full search in a practical time period. Thus there is no proof that this is the best solution. However examination of the estimation throughput for the initial map will give an maximum upper bound to the throughput. From this it is known how much the given solution falls below this.

Another method to allow the allocation of larger programs is to clump together some of the separate address spaces of the program into single address spaces. Some of the address spaces will be procedure invocation stacks for processes and large global arrays. These would not be combined together with others. However there will also exist many small procedure code bodies, and many small size global variables.

In general, if groups of these are combined then it may reduce the allocation programs chance of performing some possible optimizations. For example the combined address space may be just slightly too large to fit into any one available space, whereas its individual memory components would have. Alternatively the individual address spaces may be accessed by only one process each. Thus they could be assigned to storage which is local to the appropriate processors. However the combined memory element would need to be accessed by every one of the accessing processes, and so could only be assigned to global stores that are accessible to all of the appropriate processors.

To minimize these problems, a suitable clumping strategy would be to only combine small address spaces into combined address spaces that do not exceed the size of the available stores by a suitably small factor. A possible value would be 10 percent. This would decrease the chance of the combined memory elements from being too large to fit anywhere. As well only memory elements that have the same set of accessing processors should be combined. This would imply that the combined memory element can be allocated to exactly the same set of stores that each of its individual components could be.

Using these rules, and assuming a program with many small memory elements, then a large decrease in the number of memory elements could be achieved. For example, if this was by a factor of ten, then a medium sized program with up to 400 memory elements (before clumping) could be handled by the present allocation program implementation.

(7.6.8) SUMMARY

The constraint reduction and search ordering algorithms work in reducing the size of the search space to be examined and in producing legal final maps. However for all but very small programs the allocation still takes an excessive amount of time to perform a complete search. However final legal maps which are good approximations to the optimal final map can still be found even with an incomplete search. Furthermore the user may speed up the allocation by the imposition of suitable constraints. Hence it is quite feasible to use the allocation program, with user guidance, to find good solutions for small to medium sized programs.

CHAPTER (8)

(8.1) CONCLUSIONS

In the introduction to this thesis the concept of a resource allocator was introduced and its application areas discussed. The methods of specifying the computer architecture and the program structure to the allocator were described. Also detailed were the means whereby the user can guide this allocation activity.

The work that is described by the thesis proper falls into three main parts. These are the sections on the information structure language, the general memory interference model and the allocator algorithms.

A) Information Structure Language

The information structure language is used to specify the structure of a multiprocessor computer architecture to the resource allocator. It is also used to specify the structure of the program and to enter the user constraints. The research work was to develop this language. The thesis derives a language definition and describes in detail how it is to be used for its intended purpose. The language syntax is borrowed from other languages, but the definition of the semantics of the language for the use in a resource allocator is new.

B) General Memory Interference Model

The general memory interference model is used by the resource allocator in its production of the throughput estimation of a resource allocation. The original memory interference model used was taken from the literature. The research consisted of developing this model to fit the resource allocator requirements. This resulted in an analytic model capable of generating the required throughput. As well it was shown how a simulation model will produce the throughput estimation in a shorter time than this analytic model.

C) Allocator Algorithms

Finally the allocator algorithms are those that actually perform the allocation of the program elements to the resources of the computer architecture. The research was to find and develop suitable algorithms to perform this. The basic solution relies on a simple tree search on the whole solution space. To make the search more practical, an algorithm called implicit enumeration with backtrack is used to minimize the search path length. With this as a start other methods were also found to reduce the size of the search. These are based upon the ordering of the search to increase the chances of quickly finding an acceptable solution, and the use of constraints upon the program elements to decide if partial solutions can be rejected.

A large Pascal program was written to implement and demonstrate these algorithms. Trial runs using this demonstrated that the constraint reduction algorithms, the implicit enumeration and the use of probability ordering of the search will reduce the size of the search, and find solutions.

The aim of the research was the development of a resource allocator for medium size programs onto multimicroprocessor computer architectures. The thesis describes and demonstrates how this may be done. However the implementation of the final allocator algorithms can produce allocations for only some allocation problems. It can not, without user guidance, perform a complete search to find the optimal solution in a practical time for reasonable sized computer architectures or programs. Nor have the effect of proximity constraints been properly included.

The research that needs to be done to make the resource allocator feasible for production programs can be divided into two areas-

A) The development of the allocator algorithms to cope with larger computer architectures and larger programs. This can be done by improving the existing techniques for ordering searches, and by adding more constraint reduction operations. It can also be done by using a more complex search algorithm, to allow the incorporation of specialized information into the allocation activity. For example the optimal search strategy of a systolic computer architecture could be made different from the optimal search strategy for other architectures. The allocator would need the ability to determine what kind of architecture it is using, and to select the appropriate search strategies.

B) The implementation of an actual allocator system, capable of starting with a concurrent program and converting this into code to run on a multiprocessor architecture. This would required converting an existing compiler to generate suitable code, the implementation of the information structure language, and the provision of a linker loader to place the code and data elements of the program onto the architecture as dictated by the resource allocation mapping.

APPENDIX (A)

(A.1) CONSTRUCTION OF THE STATIC ACCESS ARRAY

This section describes the algorithm used within the analytic model to construct the static access array for the simulation program, in the situation where the simulation is to be run in its accurate mode.

The static access matrix is used by the simulation program to obtain the next store fetch for a processor. The probability of processor P picking a store S is given by

$$Sa(P,S) \qquad \ldots(33)$$

and for the busiest processor P', the following equation holds

$$\sum_{S=1}^{M} Sa(P',S) = 1$$

where Sa represents the static access. ...(34)

However in general the above equation is more correctly expressed as

$$\sum_{S=1}^{M} Sa(P,S) \leq 1 \qquad \ldots(35)$$

where the summation equals 1 for the busiest processor, and less than 1 for all others. These processors perform idle cycles, and the model knows how much idle time is spent, this is given in equation(30). Thus

$$Tidle(P) = 1 - \sum_{S=1}^{M} Sa(P,S)*Tcy(P) \qquad \ldots(36)$$

This corresponds to the simulation using the static access terms for the probability, and since the summation of these is less than one, then on the occasions when no store is picked, it just executes an idle processor cycle.

This static access matrix is derived directly from the actual number of cycles array used in the model,

$$Sa(P,S) = Na(P,S) * D(P)$$

where D is the adjustment factor. ...(37)

The number of cycles array gives the unnormalized probability distribution for the processor to store access pattern. Thus in the above the adjustment factor normalizes each row (one row per processor) to give the static access array. Using the above two equations gives

$$D(P) = (Tcy(P) - Tidle(P))/(Tcy(P)*Nap(P))$$

...(38)

where Nap is the actual number of cycles per processor array. So the static access matrix can be calculated from the actual number of cycles array and the idle time array, which are both known to the probability model.

APPENDIX (B)

(B.1) CALCULATING THE CONFLICT FUNCTION

The algorithm used in the implementation of the probability model to produce the conflict function is taken directly from [44]. To illustrate how it works, consider the following version of the conflict function

$$Cf = \sum_{k=1}^{n} \frac{1}{k} \sum_{r=1}^{fmax} \prod_{f=1}^{n} Fkrn(f) \begin{cases} = 0 & F(f) \\ \neq 0 & 1-F(f) \end{cases} \qquad ...(39)$$

Here the probability terms are represented by the F function. The expansions of the conflict function for 1, 2 or 3 F function terms are

CONFLICT_FUNCTION = (1-A) + 1/2(A)

CONFLICT_FUNCTION = (1-A)(1-B) + 1/2((1-A)B + A(1-B)) + 1/3AB

CONFLICT_FUNCTION = (1-A)(1-B)(1-C) +
+ 1/2((1-A)BC + A(1-B)C + AB(1-C))
+ 1/3((1-A)(1-B)C + (1-A)B(1-C) + A(1-B)(1-C))
+ 1/4(ABC)

where A = F(1), B = F(2) and C = F(3). If the second expansion is taken and listed as a series of terms-

L(0) = (1-A)(1-B)
L(1) = (1-A)B + A(1-B)
L(2) = AB

then multiplying each term by (1-C) to produce one new series, and by C to produce another new series, will result in

$$L(0) = (1-A)(1-B)(1-C)$$
$$L(1) = (1-A)B(1-C) + A(1-B)(1-C)$$
$$L(2) = AB(1-C)$$

$$L(0) = (1-A)(1-B)C$$
$$L(1) = (1-A)BC + A(1-B)C$$
$$L(2) = ABC$$

From inspection it can be seen that adding L(n) from the first series immediately above to L(n-1) of the second series will produce the terms of the third conflict function expansion.

Thus the recursive definition of this is

$$new\_L(n) = old\_L(n)\ (1-F(f)) + F(f)\ old\_L(n-1) \qquad \ldots(40)$$

where L(0) = 1 - F(1) and L(1) = F(1). Thus the complete algorithm to generate the conflict function were there are N function terms is

```
OLD_L(0) := 1-F(1) ;
OLD_L(1) := F(1) ;
FOR J := 2 TO N DO
   NEW_L(0) := OLD_L(0) * (1-F(J)) ;
   OLD_L(J) := 0 ;
   FOR M := 1 TO J DO
      NEW_L(M) := OLD_L(M) * ( 1-F(J) ) + F(J) * OLD_L(M-1) ;
   END ;
   OLD_L := NEW_L ;
END ;
```

This will result in the L arrays containing the terms of the conflict function. Now all that is needed is to combine these together

```
CF := 0 ;
FOR J := 0 TO N DO
   CF := CF + OLD_L(J)/(J+1) ;
END ;
CONFLICT_FUNCTION := CF ;
```

APPENDIX (C)

(C.1) PROPAGATION TABLE

The reduction operations can alter a mapping by reducing the constraint set of a process or memory element, or by altering a Same_Constraint or Different_Constraint. In all cases the changes are reflected in the sets

JUST_CHANGED_PROCESS_SET
JUST_CHANGED_MEMORY_SET
JUST_CHANGED_DIFFERENT_PROCESS_CONSTRAINT_SET
JUST_CHANGED_DIFFERENT_MEMORY_CONSTRAINT_SET
JUST_CHANGED_SAME_PROCESS_CONSTRAINT_SET
JUST_CHANGED_SAME_MEMORY_CONSTRAINT_SET

Which records all of the changes produced in the latest pass of the constraint operations. Any element which becomes fixed is also recorded in the sets

JUST_FIXED_PROCESS_SET
JUST_FIXED_MEMORY_SET

A newly fixed Same_Constraint or Different_Constraint is detected, in the implementation, by accessing each such constraint and determining how many constraint elements they possess.

If any of these sets are not empty at the end of a pass, then the information is transferred to the sets

JUST_CHANGED_PROCESS_SAVE_SET
JUST_CHANGED_MEMORY_SAVE_SET
JUST_FIXED_PROCESS_SAVE_SET
JUST_FIXED_MEMORY_SAVE_SET
JUST_CHANGED_DIFFERENT_PROCESS_CONSTRAINT_SAVE_SET
JUST_CHANGED_DIFFERENT_MEMORY_CONSTRAINT_SAVE_SET
JUST_CHANGED_SAME_PROCESS_CONSTRAINT_SAVE_SET
JUST_CHANGED_SAME_MEMORY_CONSTRAINT_SAVE_SET

and this information is used in the next pass to select which elements are to be examined for constraint reduction. This is done as follows,

ALLOWED_MEMORY_SIZE

All nonfixed memory that can be assigned to a store that contains a just fixed memory element are examined.

ALLOWED_PROCESS_SIZE

All nonfixed processes that can be assigned to a processor that contains a just fixed process element or which accesses a store that contains a just fixed memory element are examined.

SAME_MEMORY_SIZE

All Same_Memory_Constraints that are not fixed and contain a reference to a store which has just had a memory element fixed to are examined. (A fixed Same_Constraint or Different_Constraint is one where all of the constraints have been removed and so it is an empty constraint).

SAME_PROCESS_SIZE

All Same_Process_Constraints that are not fixed and contain a reference to a processor which has just had a process element fixed to it, or which accesses a store which has just had a memory element fixed to, are examined.

MEMORY_PARTITION_SIZE

All nonfixed memory elements are examined.

PROCESS_PARTITION_SIZE

All nonfixed process elements are examined.

ALLOWED_MEMORY_SET

All nonfixed memory elements that are accessed by processes which access a just changed memory element are examined.

ALLOWED_PROCESS_SET

All nonfixed process elements that access memory elements which are accessed by just changed process elements are examined.

SAME_MEMORY_SET_INDIVIDUAL

All nonfixed Same_Memory_Constraints which contain a just changed memory element are examined.

SAME_PROCESS_SET_INDIVIDUAL

All nonfixed Same_Process_Constraints which contain a just changed process element are examined.

DIFFERENT_MEMORY_NUMBER

All just changed Different_Memory_Constraints are examined.

DIFFERENT_PROCESS_NUMBER

All just changed Different_Process_Constraints are examined.

DIFFERENT_MEMORY_REMOVE

All nonfixed Different_Memory_Constraints which contain a just fixed memory element are examined.

DIFFERENT_PROCESS_REMOVE

All nonfixed Different_Process_Constraints which contain a just fixed process element are examined.

DIFFERENT_MEMORY_SET

All just changed nonfixed Different_Memory_Constraints are examined.

DIFFERENT_PROCESS_SET

All just changed nonfixed Different_Process_Constraints are examined.

SAME_MEMORY_SET

All just changed nonfixed Same_Memory_Constraints are examined.

SAME_PROCESS_SET

All just changed nonfixed Same_Process_Constraints are examined.

DIFFERENT_MEMORY_SET_INDIVIDUAL

All nonfixed Different_Memory_Constraints which contain just changed memory elements are examined.

DIFFERENT_PROCESS_SET_INDIVIDUAL

All nonfixed Different_Process_Constraints which contain just changed process elements are examined.

APPENDIX (D)

(D.1) ALGORITHMS AND MAP OPERATORS

In the following the symmetry redundancy removal algorithm and the search algorithm are described. The Pascal language is used, with upper case text representing actual Pascal coding. Lower case text represents pseudocode that has not been expanded all the way into actual Pascal code.

After this is a list of all the operators that can be used to access the state of a partial or complete map allocation.

(D.1.1) SYMMETRY REDUNDANCY REMOVAL ALGORITHM

```
TYPE
   REDUNDANCY_SET_TYPE (* This is a set of resources, it contains
      resources that are equivalent to each other, or
      resources that have not yet been shown to be
      nonequivalent *)
   LIST_TYPE (* This contains a list of redundant sets,
      In this list, the ordinal number of the first redundancy
      set is 1, the second is 2 and so on *)
VAR
   LIST : LIST_TYPE ;
   Create four redundancy sets, one each for PROCESSOR, STORE, BUS
      and BANK. Initialise each set to contain all the processors,
      stores, buses and banks of the architecture ;
   LIST := empty list ;
   Insert these four redundant sets into LIST ;

PROCEDURE REMOVE_SYMMETRY_REDUNDANCIES ;
BEGIN
   REPEAT
      NONTOPOLOGICAL_SEPARATION ; (* Separate the redundancy sets
         into subsets to make further redundancy sets, depending
         upon nontopological information *)
      WHILE any redundancy sets in LIST with more than one element
      remain DO BEGIN
         TOPOLOGICAL_SEPARATION ; (* Separate the redundancy sets
            into further subsets depending upon topological
            information *)
         IF no changes where made in last step THEN
            exit while loop ;
      END ;
      REDUCE_SETS ; (* Based upon the contents of the
         redundancy sets, reduce the allowed constraints *)
   UNTIL no reductions were made in the last repeat loop ;
END ;

PROCEDURE NONTOPOLOGICAL_SEPARATION ;
VAR
   WORK_LIST : LIST_TYPE ;
   NEW_REDUNDANCY , OLD_REDUNDANCY : REDUNDANCY_SET_TYPE ;
BEGIN
   Put all redundancy sets into a list called WORK_LIST ;
   Initialise LIST to be empty ;
   WHILE the WORK_LIST is nonempty DO BEGIN
      OLD_REDUNDANCY := a redundancy set extracted from WORK_LIST;
      Initialise the set NEW_REDUNDANCY to empty ;
      FOR all elements in the OLD_REDUNDANCY, except for the first
         element DO BEGIN
         IF NONTOPOLOGICAL_DIFFERENT (* if the properties
            of the first element differ from this element *) THEN
         BEGIN
            Extract this element from the OLD_REDUNDANCY, insert
               it into NEW_REDUNDANCY set ;
         END ;
      END ;
      IF NEW_REDUNDANCY set is nonempty THEN BEGIN
         Place it into the WORK_LIST ;
      END ;
   Insert OLD_REDUNDANCY into LIST ;
   END ;
END ;
```

```
PROCEDURE TOPOLOGICAL_SEPARATION ;
VAR
   WORK_LIST : LIST_TYPE ;
   NEW_REDUNDANCY , OLD_REDUNDANCY : REDUNDANCY_SET_TYPE ;
BEGIN
   REPEAT
      Put all redundancy sets into a list called WORK_LIST ;
      Initialise LIST to be empty ;
      WHILE the WORK_LIST is nonempty DO BEGIN
         OLD_REDUNDANCY := a redundancy set extracted from
            WORK_LIST ;
         Initialise the set NEW_REDUNDANCY to empty ;
         FOR all elements in the OLD_REDUNDANCY set, except for
            the first element DO BEGIN
            IF TOPOLOGICAL_DIFFERENT (* if the properties of
               the processors, banks, buses and stores that access
               or are accessed by this element are different
               from the kind and properties of those of the first
               element *) THEN BEGIN
               Extract this element from the OLD_REDUNDANCY set,
                  insert it into the NEW_REDUNDANCY set ;
            END ;
         END ;
         IF NEW_REDUNDANCY set is nonempty THEN BEGIN
            Place it into the WORK_LIST ;
         END ;
      Insert OLD_REDUNDANCY into LIST ;
      END ;
   UNTIL no new redundancy sets are created in the last loop ;
END ;

PROCEDURE NONTOPOLOGICAL_DIFFERENCE
BEGIN
   CASE kind of element OF
   processor element :
      Two processors are different if they have different
         cycle speeds,
         brand names,
         number of stores attached,
      total size of all the stores attached,
      process sets, as allowed by the process to processor
         constraints.
  store element:
    Two stores are different if they have different
      access speeds,
      rewrite recovery times,
      memory sets, as allowed by the memory to store
         constraints,
      number of accessing processors.
  bus element:
    Two buses are different if they have different
      number of processors accessing them,
      number of attached stores,
      bus delay times.
  bank element:
    Two banks are different if they have different
      bank access times.
  END ;
END ;

FUNCTION TOPOLOGICAL_DIFFERENCE
  Two elements are different if they have different
    attachments lists. The attachment set of a processor
    element is found by using PROCESSOR_ATTACHMENT,
    similarly for the others.

PROCEDURE PROCESSOR_ATTACHMENT
BEGIN
  Create an initially empty processor attachment list.
  FOR all stores that the processor accesses DO BEGIN
    Insert the ordinal number of the redundancy set that
      contains the store element into the attachment list.
  END ;
  FOR all buses that the processor accesses DO BEGIN
    Insert the ordinal number of the redundancy set that
      contains the bus element into the attachement list.
  END ;
  FOR all banks that the processor accesses DO BEGIN
    Insert the ordinal number of the redundancy set that
```

```
            contains the bank element into the attachement list.
        END ;
        Order the attachment list
    END ;

    PROCEDURE REDUCE_MEMORY_SETS ;
    VAR
        MEMORY : MEMORY_SET_TYPE ;
        POSSIBLE_REDUNDANT_STORES : STORE_SET_TYPE ;
        STORE : STORE_SET_TYPE ;
        REDUNDANCY_SET : RESOURCE_SET_TYPE ;
    BEGIN
        FOR MEMORY := all memory DO BEGIN
            FOR REDUNDANCY_SET := all redundancy sets containing store
                elements DO BEGIN
                POSSIBLE_REDUNDANT_STORES :=
                    ALLOWED_STORE_FROM_MEMORY ( MEMORY ) *
                    REDUNDANCY_SET ;
                IF number of elements in POSSIBLE_REDUNDANT_STORES > 1
                THEN BEGIN
                    STORE := first element from
                        POSSIBLE_REDUNDANT_STORES ;
                    Change the allowed stores from MEMORY to
                        ALLOWED_STORE_FROM_MEMORY ( memory ) -
                        POSSIBLE_REDUNDANT_STORES + STORE ;
                    Exit procedure (* a reduction has been made *)
                END ;
            END ;
        END ;
    END ;

        Similarly for the processes.
```

(D.1.2) THE SEARCH ALGORITHM

```
TYPE
    MAP_TYPE (* This will contain one partial or complete
        map allocation. This includes the process to
        processor and memory to store constraints, and the
```

```
      proximity constraints. *)
    PROCESS_MEMORY_LIST_TYPE (* This is a list of process and
      memory elements *)
    ELEMENT_TYPE (* Will contain either a process element
      or a memory element *)
    RESOURCE_TYPE (* Will contain either a processor resource element
      or a store resource element *)
    MAP_ELEMENT_TYPE = RECORD
      MAP : MAP_TYPE ;
      RESOURCE : RESOURCE_TYPE ;
      THROUGHPUT : REAL ;
    END ;
    MAP_LIST = list of MAP_ELEMENT_TYPE ;

VAR
    BEST_EVER_THROUGHPUT : REAL ; (* This contains the throughput
      of the best ever final map so far found. If no such map
      has been found yet, it contains 0 *)
    GLOBAL_SUCCESS : BOOLEAN ; (* This is set to true when a
      complete solution is found *)
    FINAL_MAP : MAP_TYPE ; (* This will contain the best complete
      solution found, if one is found at all *)

PROCEDURE ALLOCATION ; VAR
    PROCESS_MEMORY_LIST : PROCESS_MEMORY_LIST_TYPE ;
    MAP : MAP_TYPE ;
BEGIN
    GLOBAL_SUCCESS := FALSE ;
    Initialise the PROCESS_MEMORY list, by
      inserting all the process and memory elements into
      the list, then sorting them into order.
    MAP := Initial input map as specified by the user constraints ;
    SEARCH ( MAP , PROCESS_MEMORY_LIST ) ;
END ;

PROCEDURE SEARCH (
    MAP : MAP_TYPE ;
    PROCESS_MEMORY_LIST : PROCESS_MEMORY_LIST_TYPE ) ;
VAR
    MAP_LIST : MAP_LIST_TYPE ;
    NEXT_ELEMENT : ELEMENT_TYPE ;
    TEMPORARY : MAP_ELEMENT_TYPE ;
    MAP_ELEMENT : MAP_ELEMENT_TYPE ;
    RESOURCE : RESOURCE_TYPE ;
BEGIN
    IF empty_list ( PROCESS_MEMORY_LIST ) THEN BEGIN
       GLOBAL_SUCCESS := TRUE ;
       BEST_EVER_THROUGHPUT := Throughput ( MAP ) ;
       FINAL_MAP := MAP ;
    END ELSE BEGIN
       NEXT_ELEMENT := First element in PROCESS_MEMORY_LIST ;
       MAP_LIST := empty list ;
       FOR RESOURCE := all resources to which NEXT_ELEMENT may be
          assigned, as specified by MAP   DO BEGIN
       BEGIN
          TEMPORARY.MAP := MAP ;
          Using TEMPORARY.MAP, constrain NEXT_ELEMENT to RESOURCE ;
          IF legal map created ( TEMPORARY.MAP ) THEN BEGIN
             IF throughput ( TEMPORARY.MAP ) >
                THROUGHPUT_FACTOR * BEST_EVER_THROUGHPUT THEN BEGIN
                TEMPORARY.THROUGHPUT := throughput ( TEMPORARY.MAP ) ;
                TEMPORARY.RESOURCE := RESOURCE ;
                Insert TEMPORARY into MAP_LIST ;
             END ;
          END ;
       END ;
       (* MAP_LIST now has a list of the possible resources for the
          NEXT_ELEMENT, together with their associated map allocations
          and throughputs *)

       IF the computer has a homogeneous architecture THEN BEGIN
          IF NEXT_ELEMENT is a process THEN BEGIN
             Sort the MAP_LIST upon
                number of processes in the set (
                   ALLOWED_PROCESS_FROM_PROCESSOR (
                      MAP_LIST^.RESOURCE ) )
          END ELSE BEGIN
             Sort the MAP_LIST upon MAP_LIST^.THROUGHPUT ;
             Reverse the list ; (* puts the maps with the
                highest throughput first *)
```

```
        END ;
    END ELSE BEGIN
        IF NEXT_ELEMENT is a memory THEN BEGIN
            Sort MAP_LIST upon
                number of processors in the set (
                    ACCESS_PROCESSOR_FROM_STORE (
                        MAP_LIST^.RESOURCE ) )
        END ELSE BEGIN
            Sort the MAP_LIST upon M/P_LIST^.THROUGHPUT ;
            Reverse the list ; (* puts the maps with the
                highest throughput first *)
        END ;
    END ;
    (* Have now sorted the MAP_LIST so that the most
        promising resource targets for NEXT_ELEMENT come first in
        the list *)

    FOR MAP_ELEMENT := all map elements in MAP_LIST DO BEGIN
        IF MAP_ELEMENT.THROUGHPUT >
            > THROUGHPUT_FACTOR * BEST_EVER_THROUGHPUT THEN BEGIN
            SEARCH ( MAP_ELEMENT.MAP ,
                        PROCESS_MEMORY_LIST - NEXT_ELEMENT ) ;
        END ;
    END ;
END ;
```

(D.1.3) OPERATOR NAMES

In the list that appears below the names and uses of the operators that have been mentioned in the thesis are given. These operators are implemented as Pascal functions that return set type values. Since Pascal functions can not actually return set types, these are modified accordingly in the actual Pascal program coding.

```
ALLOWED_MEMORY_FROM_STORE ( MAP : MAP_TYPE ;
    STORE : STORE_SET_TYPE ) : MEMORY_SET_TYPE ;
    This returns the set of all memory elements M such that
    there exists at least one store S in the STORE set where
    M is allowed to be assigned to S.
```

ALLOWED_STORE_FROM_MEMORY (MAP : MAP_TYPE ;
MEMORY : MEMORY_SET_TYPE) : STORE_SET_TYPE ;

This returns the set of all store resources S such that there exists at least one memory M in the MEMORY set where M is allowed to be assigned to S.

ALLOWED_PROCESS_FROM_PROCESSOR
ALLOWED_PROCESSOR_FROM_PROCESS

Similar to the above.

ACCESS_PROCESSOR_FROM_STORE (STORE : STORE_SET_TYPE) :
PROCESSOR_SET_TYPE ;

This returns the set of all processor resources P such that there exists at least one store S in the STORE set where processor P can access store S.

ACCESS_STORE_FROM_PROCESSOR (PROCESSOR : PROCESSOR_SET_TYPE) :
STORE_SET_TYPE ;

This returns the set of all store resources S such that there exists at least one processor P in the PROCESSOR set where processor P can access store S.

ACCESS_PROCESSOR_FROM_BUS
ACCESS_PROCESSOR_FROM_BANK
ACCESS_STORE_FROM_BUS
ACCESS_STORE_FROM_BANK
ACCESS_BUS_FROM_BANK
ACCESS_BUS_FROM_PROCESSOR
ACCESS_BUS_FROM_STORE
ACCESS_BANK_FROM_BUS
ACCESS_BANK_FROM_PROCESSOR
ACCESS_BANK_FROM_STORE

Similar to the above two definitions.

FIXED_MEMORY (MEMORY : MEMORY_SET_TYPE) : MEMORY_SET_TYPE ;

This returns all memory M that are in the MEMORY set and have been allocated to a single store.

FIXED_PROCESS
   Similar to the above.

SIZE_UNUSED_STORE (MAP : MAP_TYPE ; STORE : STORE_SET) : INTEGER
   This returns the size of the unused memory space in the stores of the STORE set.

SIZE_NONFIXED_MEMORY (MAP : MAP_TYPE ;
   MEMORY : MEMORY_SET_TYPE) : INTEGER ;
   This returns the size of all the nonfixed memory elements in the MEMORY set.

SIZE_PROCESSOR_UNUSED_STORE (MAP : MAP_TYPE ;
   PROCESSOR : PROCESSOR_SET_TYPE) : INTEGER ;
   This returns the size of all the unused memory space of all the stores that are accessible by the processors in the PROCESSOR set.

SIZE_NONFIXED_MEMORY_OF_PROCESS_FIXED_TO_PROCESSOR (
   MAP : MAP_TYPE ;
   PROCESSOR : PROCESSOR_SET_TYPE) : INTEGER ;
   This returns the size of all the nonfixed memories that are accessed by all the processes that are fixed to the processors in the PROCESSOR set.

SIZE_NONFIXED_MEMORY_OF_NONFIXED_PROCESS (MAP : MAP_TYPE ;
   PROCESS : PROCESS_SET_TYPE) : INTEGER ;
   This returns the size of all the nonfixed memories that are accessed by all the nonfixed processes that are in the PROCESS set.

SIZE_THIS_PROCESSOR (MAP : MAP_TYPE ;
   PROCESSOR_SET_TYPE) : INTEGER ;
   This returns the total store space in all the stores that the processors of the PROCESSOR set can access.

NONFIXED_SAME_PROCESS_CONSTRAINT (MAP : MAP_TYPE ;
   SAME_PROCESS_CONSTRAINT :
      SAME_PROCESS_CONSTRAINT_SET_TYPE) :
   SAME_PROCESS_CONSTRAINT_SET_TYPE ;

This returns all the SAME_PROCESS proximity constraints that are in the SAME_PROCESS_CONSTRAINT set and which contain processes that are not yet fixed.

NONFIXED_SAME_MEMORY_CONSTRAINT
NONFIXED_DIFFERENT_PROCESS_CONSTRAINT
NONFIXED_DIFFERENT_MEMORY_CONSTRAINT
Similar to the above

PROCESS_FROM_SAME_PROCESS_CONSTRAINT (MAP : MAP_TYPE ;
SAME_PROCESS_CONSTRAINT : SAME_PROCESS_CONSTRAINT_TYPE) :
PROCESS_SET_TYPE ;
This returns with all processes P such that P is mentioned in at least one of the SAME_PROCESS proximity constraints in the SAME_PROCESS_CONSTRAINT set.

MEMORY_FROM_SAME_MEMORY_CONSTRAINT
PROCESS_FROM_DIFFERENT_PROCESS_CONSTRAINT
MEMORY_FROM_DIFFERENT_MEMORY_CONSTRAINT
Similar to the above

ORED_PROCESSOR_FROM_SAME_PROCESS_CONSTRAINT (MAP : MAP_TYPE ;
SAME_PROCESS_CONSTRAINT : SAME_PROCESS_CONSTRAINT_SET_TYPE) :
PROCESSOR_SET_TYPE ;
This returns the set of all processors P such that P is in at least one of the target processor sets of at least one SAME_PROCESS constraint in the SAME_PROCESS_CONSTRAINT set.

ORED_STORE_FROM_SAME_MEMORY_CONSTRAINT
ORED_PROCESSOR_FROM_DIFFERENT_PROCESS_CONSTRAINT
ORED_STORE_FROM_DIFFERENT_MEMORY_CONSTRAINT
Similar to the above.

ALL_DIFFERENT_PROCESS_CONSTRAINTS_WITH_PROCESS (MAP : MAP_TYPE ;
PROCESS : PROCESS_SET_TYPE) :
DIFFERENT_PROCESS_CONSTRAINT_SET_TYPE ;
This returns with all the DIFFERENT_PROCESS proximity constraints in the map that contain at least one of the processors P, where P is also in the PROCESS set.

ALL_DIFFERENT_MEMORY_CONSTRAINTS_WITH_MEMORY
ALL_SAME_PROCESS_CONSTRAINTS_WITH_PROCESS
ALL_SAME_MEMORY_CONSTRAINTS_WITH_MEMORY
   Similar to the above.

```
ALL_DIFFERENT_PROCESS_CONSTRAINTS_WITH_PROCESSOR (
   MAP : MAP_TYPE ;
   PROCESSOR : PROCESSOR_SET_TYPE ) :
      DIFFERENT_PROCESS_CONSTRAINT_SET_TYPE ;
```
   This returns with all of the DIFFERENT_PROCESS constraints in the map that mention processor P, where P is also a member of the PROCESSOR set.

ALL_DIFFERENT_MEMORY_CONSTRAINTS_WITH_STORE
ALL_SAME_PROCESS_CONSTRAINTS_WITH_PROCESSOR
ALL_SAME_MEMORY_CONSTRAINTS_WITH_STORE
   Similar to the above.

```
FIXED_PROCESS_FROM_PROCESSOR ( MAP : MAP_TYPE ;
   PROCESSOR : PROCESSOR_SET_TYPE ) : PROCESS_SET_TYPE ;
```
   This returns all the processes P such that process P is fixed to processor PSR, where PSR is a member of the PROCESSOR set.

FIXED_MEMORY_FROM_STORE
   Similar to the above.

APPENDIX (E)

(E.1) INFORMATION SPECIFICATION LANGUAGE

The information specification language (ISL) allows a machine understandable definition of a computer architecture to be constructed. It also provides the user with the facilteis to guide the resource allocation activity.

This appendix will describe in detail the basic structure of this language, and introduce the parts of the language concerned with the definition of a computer architecture. It starts with a section on reference, or how to access a particular vertex from a given starting vertex. After this the operations of creating new vertices and attaching them to the existing graph are explained. These allow the construction of an ISL graph structure.

Eventually other parts of the ISL, which deal with the declaration of the names used in the language and the grouping of the ISL statements, are described.

(E.2) STATEMENTS

An ISL program consists of statements and definitions. Statements are used to perform the actions of creating a graph. Definitions are used to define various identifiers that are used by the statements. In the following statements will be described first, followed by definitions.

Firstly, the syntax of a statement block is

```
    Statement_Block = { Statement }- ;

    Statement = Assignment_Statement |
                Attach_Statement     |
                For_Statement        |
                If_Statement         |
                Procedure_Call_Statement ;
```

These statement kinds are discussed in turn.

(E.2.1) ASSIGNMENT STATEMENTS

An assignment statement will assign a value to a variable. The syntax is

```
    Assignment_Statement =
       Variable_Identifier, ":=", Expression, ";" ;

    Expression =
       Simple_Expression,
       [ Comparison_Operator, Simple_Expression ] ;

    Comparison_Operator = "<" | ">" | "<=" | ">=" | "=" | "<>" ;

    Simple_Expression =
       [ Unary_Operator ], Term, { Term_Operator, Term } ;
```

```
Unary_Operator = "+" | "-" ;

Term_Operator = "OR" | "+" | "-" ;

Term = Factor, { Factor_Operator, Factor } ;

Factor_Operator = "*" | "/" | "AND" ;

Factor = Unsigned_Constant | Variable_Identifier |
         Reference | Special_Function |
         Bracketed_Expression | Not_Factor ;

Not_Factor = Not_Operator, Factor ;

Not_Operator = "NOT" ;

Bracketed_Expression = "(", Expression, ")" ;

Unsigned_Constant = Constant_Identifier | String |
                    Unsigned_Number ;
```

This syntax definition allows standard arithmetical expressions using integers, reals, booleans and strings to be constructed. It provides for scalar variables and constants in these expressions. It also provides References and Special_Functions. These are used in statements that access a graph structure.

(E.2.2) OPERATOR DEFINITIONS

The operators used in an expression are given below, in their precedence order.

```
Comparison_Operator        <   >   <=  >=  <>   =
Term_Operator              OR  +   -
Factor_Operator            *   /   AND

Not_Operator               NOT
```

The Not_Operator is a monadic operator, it accepts one argument to generate its result. The two Unary_Operators are also monadic. The other operators are dyadic operators, they accept two arguments to generate one result. Each operator requires arguments of the appropriate type. Furthermore for dyadic operations the types of the two arguments used must be identical. The type of the output result may depend upon the type of the arguments.

The allowable types of an expression are INTEGER, REAL, STRING, BOOLEAN and SET. The first four have the standard properties, while the SET type refers to sets of vertices of a graph.

The operators with their allowed argument types and the corresponding result types are listed in the table below.

Operator	Argument type	Result type
< > <= >=	Integer, Real	Boolean
= <>	Integer, Real, String	Boolean
= <>	Set, Boolean	Boolean
OR	Boolean	Boolean
+ -	Integer	Integer
+ -	Real	Real
+ -	Set	Set
* /	Real, Integer	Real
*	Set	Set
AND	Boolean	Boolean
NOT	Boolean	Boolean
unary + -	Integer	Integer
unary + -	Real	Real

The operations that are specific to the ISL are those concerned with SET type arguments. Such sets contains vertices of the graph. The operations of set union, set subtraction and set intersection which are defined upon these have the usual set semantics.

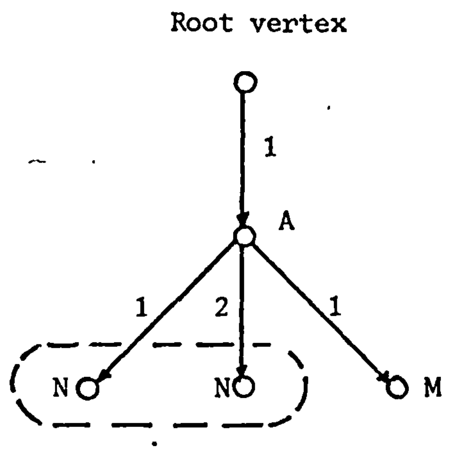


Figure E.2

(E.2.3) REFERENCES

Given an information graph structure, a means of accessing individual elements within this is required. The use of references for this purpose will now be described.

For a graph G=(X,M), the attached name set of a vertex Xi, for the name N, can be defined. It is the set of all vertices Xj that are attached to Xi and which have a name function Fn(Xj) of N. This set is represented by the notation Fattach(Xi,N).

The vertices in an attached name set are ordered, forming the attached name list (Xj1,Xj2,Xj3,...). Generally the vertices are ordered in the same sequence in which they are created, this is discussed fully in section(E.2.4), on attach statements. Any vertex in an attached name set can be referred to uniquely by giving its ordinal position in the attached name list. This is called the index of the vertex Xj with respect to Xi. This is represented by the notation Findex(Xi,Xj).

In the graph of figure(E.2) the vertices in the attached name set of the vertex A, for the name N, are circled. The numbers on the arcs leading to these vertices represent their index values.

Every reference starts from some vertex or set of vertices. This set is called the starting set of the reference. The reference will refer to the vertices of this starting set, or it will refer to vertices that are attached to the vertices of this starting set. The vertices that the reference refers to are called the reference set of the reference.

(E.2.3.1) REFERENCE SYNTAX

The syntax for a reference is

```
    Reference =
       Reference_Start, { ".", Selector_Reference } ;

    Reference_Start = "@" | Reference_Set_Variable_Identifier ;

    Selector_Reference = Vertex_Selector          |
                         Conditional_Selector     |
                         Bracketed_Reference   ;

    Bracketed_Reference = "(", Reference_Set_Expression, ")" ;
    Reference_Set_Expression = Expression ;

    Vertex_Selector = ".", Vertex_Name_Identifier, [ Selector_Index ];

    Selector_Index = "(", Integer_Value, ")" ;
```

(E.2.3.2) SELECTOR REFERENCES

The simplest reference is

```
    @
```

and this will refer to all of the vertices in the references starting set. This starting set may be the root vertex of the graph, in which case this reference will refer to just the root vertex.

NOTE. The BNF format used to define the syntax follows the British Standard BS 6154 as described in [77]. In the following syntax definitions integers are never used in the metaidentifiers of a definition. In an example of a definition a metaidentifier may appear with an integer immediately after it. This refers to an actual (unspecified) example of the metaidentifier. Thus a syntax definition may be

```
A = B, C, B ;
B = "bb" | "bbb" ;
C = "cc" | "ccc" ;
```

Two specific examples of an A are

```
bb cc bbb
bb cc bb
```

A generalized example of an A could be

```
B1 cc B2
```

where B1 and B2 refer to some (unspecified) actual expansion of B. In the following

```
B1 cc B1
```

B1 refers to the same (unspecified) expansion of B in both cases.

The next simplest reference is by using a Vertex_Selector,

```
@.Vertex_Name_Identifier1
```

This reference will produce a reference set which contains the vertices

```
Fattach(Xr1,Vertex_Name_Identifier1)
Fattach(Xr2,Vertex_Name_Identifier1)
    ...
Fattach(XrN,Vertex_Name_Identifier1)
```

where the set { Xr1, Xr2,...XrN } is the reference set of the simple reference @. In other words, the reference set contains all vertices of

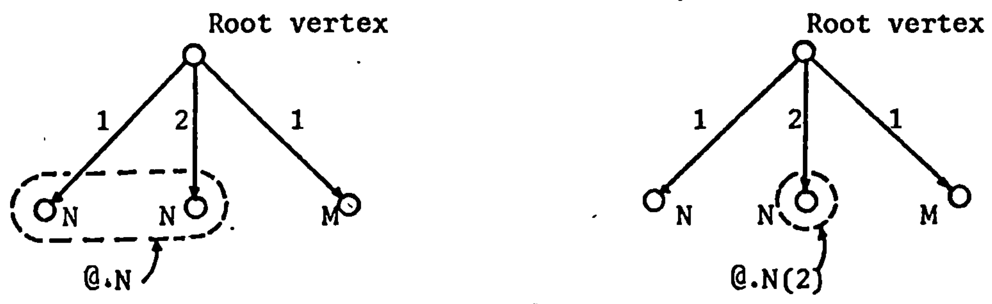


Figure E.3

name Vertex_Name_Identifier1 that are attached to all the vertices of the starting set.

Using a Selector_Index creates the reference

@ . Vertex_Name_Identifier1 (Integer_Expression1)

This produces the reference set { Xn1, Xn2,... XnN } where

Xni is an element of the reference set of
@.Vertex_Name_Identifier1, for all i from 1 to N.
For some Xrj that is an element of the reference set @,
Findex(Xrj,Xni) is equal to Integer_Expression1.

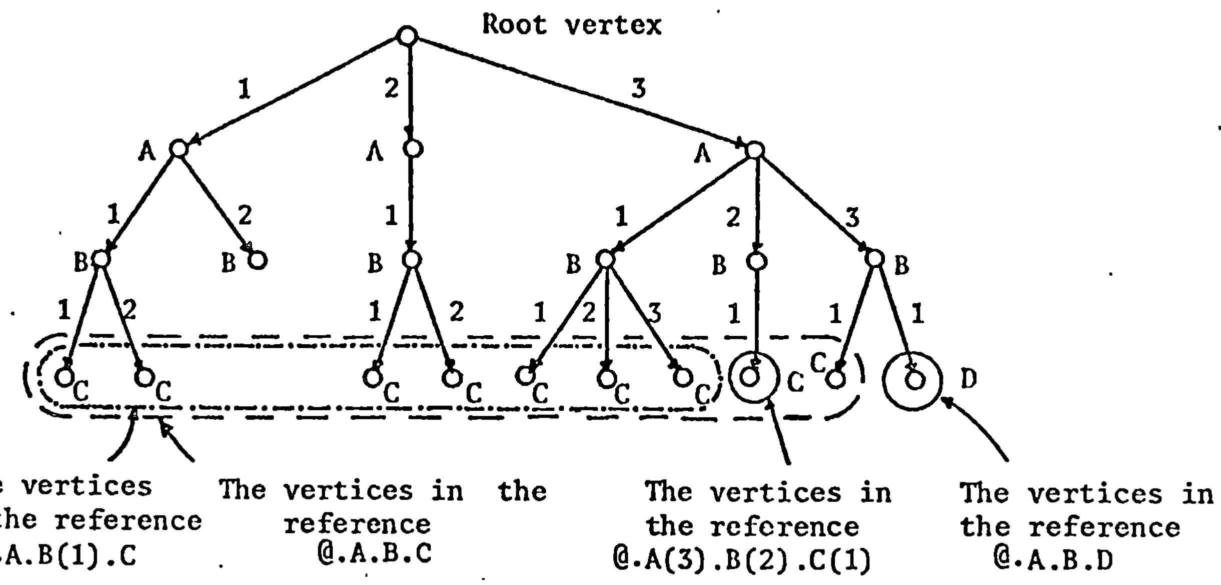
Informally, a Selector_Index will give a reference set which contains only vertices that have the indicated index value with respect to the vertices in the starting set to which they are attached.

As an example, the reference sets of the following two references are indicated in figure(E.3).

@.N
@.N(2)

(E.2.3.3) USING REFERENCE SET VARIABLES
....

A reference set variable may be used in a reference to supply its starting set. Given an assignment like



Root vertex
The vertices in the reference @.A.B(1).C
The vertices in the reference @.A.B.C
The vertices in the reference @.A(3).B(2).C(1)
The vertices in the reference @.A.B.D

Figure E.4

Vertex_Name_Set_Identifier1 := @

then the reference

Reference_Set_Variable_Identifier1 . Selector_Reference1

will be equivalent to the reference

@ . Selector_Reference1

As an example,

REF := @ ;

REF.C   is now equivalent to   @.C

(E.2.3.4) MORE THAN ONE SELECTOR REFERENCE

....

A Reference may have any number of Selector_References to it. A reference like

```
@ . Selector_Reference1 . Selector_Reference2 . ...
      Selector_ReferenceN . Selector_ReferenceM
```

where M = N+1, will produce a reference set. This will be equivalent to the reference set produced by the following reference

```
S . Selector_ReferenceM
```

where S is a reference set variable, and its contents is specified by the assignment

```
S := @ . Selector_Reference1 . Selector_Reference2 . ...
      Selector_ReferenceN
```

Some example references with more than one Selector_Reference are given in figure(3.4).

(E.2.3.5) BRACKETED REFERENCES

A reference expression may be bracketed. The starting set for all the references inside the brackets, that use @, is supplied by the reference that is placed in front of the brackets. If a bracketed reference is like

```
@. ( Reference_Expression1 )
```

then this will give the same reference set as the expression

```
Reference_Expression1
```

If the bracketed reference is

```
S . ( Reference_Expression1 )
```

where S is either a reference set variable or a reference, and Reference_Expression1 contains the factors

```
Reference1, Reference2, ... ReferenceN, ...
```

then the bracketed reference will generate the same reference set as the expression

```
Reference_Expression1'
```

where each reference ReferenceN that starts with a @ has this replaced with S.

Thus

```
@.A.B
```

has the root vertex as its starting set.

```
@.X.Y. ( @.A.B )
```

However here the reference @.A.B has the reference set of @.X.Y as its starting set. This reference is equivalent to the reference

```
@.X.Y.A.B
```

Another example is

```
@.X.Y. ( @.A.B + @.C.D * @.E.D )
```

Here each of the references @.A.B, @.C.D and @.E.D has the reference set of @.X.Y as its starting set. This reference produces the same reference set as the reference

```
@.X.Y.A.B + @.X.Y.C.D * @.X.Y.E.D
```

(E.2.3.6) CONDITIONAL REFERENCES

A conditional selector is a means of selecting vertices from a reference set which satisfy some given conditions. It is written according to the syntax

```
Conditional_Selector = "<", Boolean_Expression, ">" ;
Boolean_Expression = Expression ;
```

The simple conditional references

```
Reference1.< True >
Reference2.< False >
```

will generate either the reference set of Reference1 in the first case, or the empty reference set in the second example.

A general Conditional_Selector of the form

```
Reference1 . < Boolean_Expression1 >
```

will produce the reference set given by the expression

```
S1.< Boolean_Expression1 > +
S2.< Boolean_Expression1 > +
  ...
Sn.< Boolean_Expression1 >
```

where Si is a reference set variable that is equal to {Xi}, and the set {X1,X2,...Xn} is the reference set of the reference Reference1.

That is the boolean expression is evaluated for each of the vertices in the reference set of Reference1, and if it comes out true that vertex will be placed into the result reference set. The evaluation of the Boolean_Expression proceeds like any other expression, except that Reference1 provides the starting set for any reference that may appear in it.

Several special purpose functions are provided which are useful in this context. Their syntax is

```
Special_Function = Number_Function | Empty_Function |
                   All_Value_Function | Any_Value_Function |
                   Value_Function ;

Number_Function  =  "NUMBER", "(", Reference_Expression, ")" ;
Empty_Function   =  "EMPTY" , "(", Reference_Expression, ")" ;
Value_Function   =  "VALUE" , "(", Reference_Expression, ")" ;
Any_Value_Function = "ANY_VALUE",
        "(", Reference_Expression, Comparsion_Operator,
           Simple_Expression, ")" ;
     All_Value_Function = "ALL_VALUE",
        "(", Reference_Expression, Comparsion_Operator,
           Simple_Expression, ")" ;
```

A function like

```
     NUMBER ( Reference_Expression1 )
```

will return the integer number of vertices in the reference set of Reference_Expression1. A function like

```
     VALUE ( Reference_Expression1 )
```

assumes that there is one only vertex in the reference set of Reference_Expression1, and this vertex has a value. The function will return this value, the type of the result being the same as the type of the value. If the initial assumption is false, then this is treated as an error. A function like

```
     EMPTY ( Reference_Expression1 )
```

will return the same result as the equivalent expression

```
     ( NUMBER ( Reference_Expression1 ) = 0 )
```

A function of the kind

```
     ALL_VALUE ( Reference_Expression1 Comparsion_Operator1
                 Simple_Expression1 )
```

will return the same result as the equivalent expression

```
     ( VALUE ( S1 Comparsion_Operator1 Simple_Expression1 ) AND
       VALUE ( S2 Comparsion_Operator1 Simple_Expression1 ) AND
         ...
       VALUE ( SN Comparsion_Operator1 Simple_Expression1 )  )
```

where S1,S2,...SN are reference set variables such that

```
     S1 is equal to { X1 } ,
     S2 is equal to { X2 } ,
        ...
     SN is equal to { XN }
```

and the set { X1, X2,...XN } is the reference set of the reference Expression_Reference1. Thus this gives a true result if every vertex in the reference set satisfies the comparsion. It returns a false value if the reference set of Reference1 is empty. The last function is

```
     ANY_VALUE ( Reference_Expression1 Comparsion_Operator1
                 Simple_Expression1 )
```

and this returns a boolean type result equal to

```
     ( VALUE ( S1 Comparsion_Operator1 Simple_Expression1 ) OR
       VALUE ( S2 Comparsion_Operator1 Simple_Expression1 ) OR
          ...
       VALUE ( SN Comparsion_Operator1 Simple_Expression1 )  )
```

In other words, it returns a true result if any one of the vertices in the reference set of Expression_Reference1 satisfies the condition If the reference set is empty, it returns the false result.

(E.2.3.7) CONDITIONAL SELECTOR EXAMPLES

....

In the following some examples using the above syntax definitions are given.

A reference like

```
     Reference1. < NOT EMPTY ( Reference2 ) >
```

will produce a reference set of all vertices Xr which satisfy the conditions

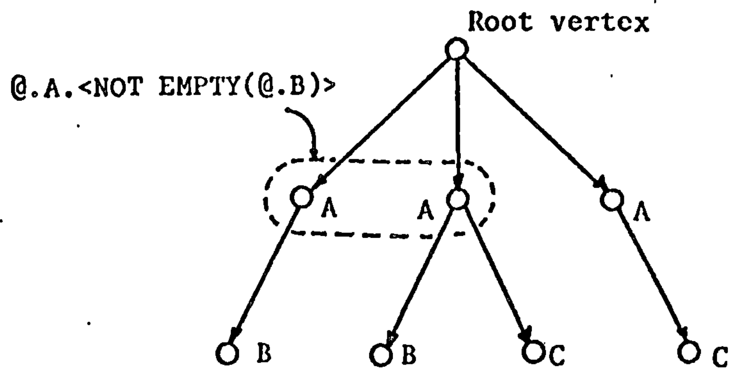


Figure E.5

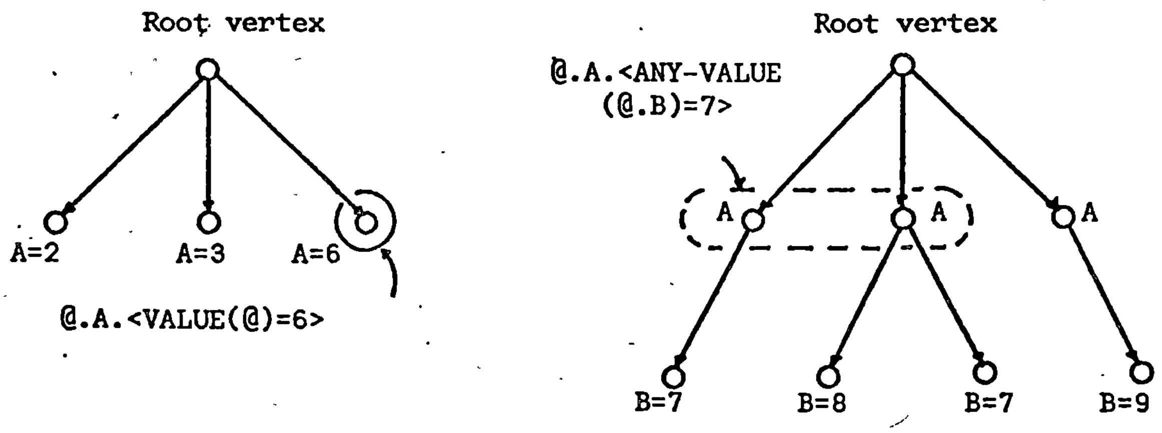


Figure E.6

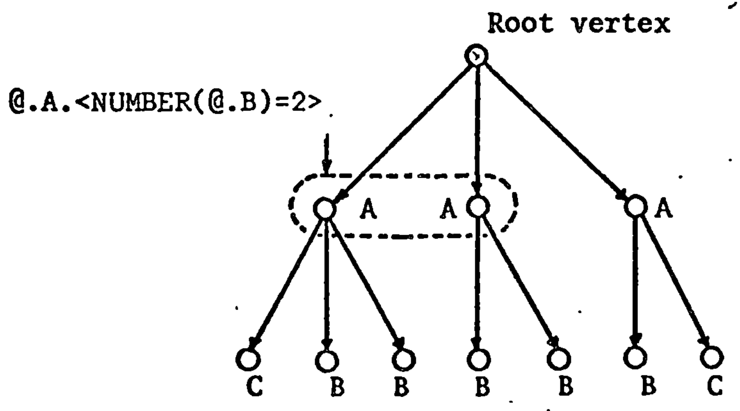


Figure E.7

Xr is in the reference set of Reference1 and
The reference set of Scr. (Reference2) is nonempty.
Sxr is a reference set variable and is equal to the reference set {Xr}.

A specific example is

```
@ . < NOT EMPTY ( @.A ) >
```

This will select the root vertex if it has a A vertex attached. If it does not, then it returns with an empty reference set. If the expression is

```
@ . A . < NOT EMPTY ( @.B ) >
```

then this will select all vertices A attached to vertices in the starting set such that each vertex A has one or more attached B vertices. Thus this expression selects the vertices as shown in figure(3.5).

An example reference using an arithmetical comparsion is

```
@ . A . < VALUE ( @ ) = 6 >
```

This assumes that all vertices A have a nonnull value function result, and will select all such vertices whose value is equal to 6. Another example is

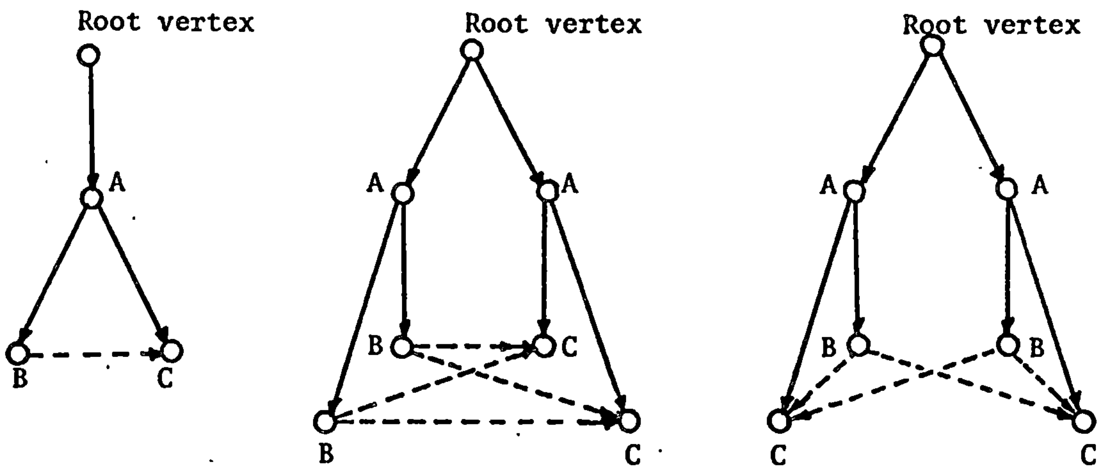
```
@.A. < ANY_VALUE ( @.B ) = 7 >
```

This will select all vertices A which have an attached vertex B whose value is 7. These two examples are depicted in figure(3.6).

Another example is the reference

```
@.A.< NUMBER (@.B) = 2 >
```

which will select all the A vertices with two B vertices attached. This is shown in figure(3.7).



The solid lines represent the orginal arcs, the dashed lines represent the new arcs added by the statement @.A.B -> @.A.C

Figure E.8

(E.2.4) ATTACH STATEMENTS

The attach operation will attach a vertex X1 to another vertex X2. Its syntax is

```
Attach_Statement = Attach_Operation, ";" ;
Attach_Operation =
   Attach_Reference, { "->", Attach_Reference }- ;
Attach_Reference = Reference_Expression | New_Operation |
                   Bracketed_Attach_Reference ;
Bracketed_Attach_Statement =
   "(", Attach_Operation, { ",", Attach_Operation }, ")" ;
Reference_Expression = Expression (* giving a SET type result *) ;
```

An attach statement like

```
Reference1 -> Reference2
```

will create directed arcs of the form (Xr1,Xr2), if Xr1 is a member of the reference set of Reference1 and Xr2 is a member of the reference set of Reference2, and the arc does not already exist.

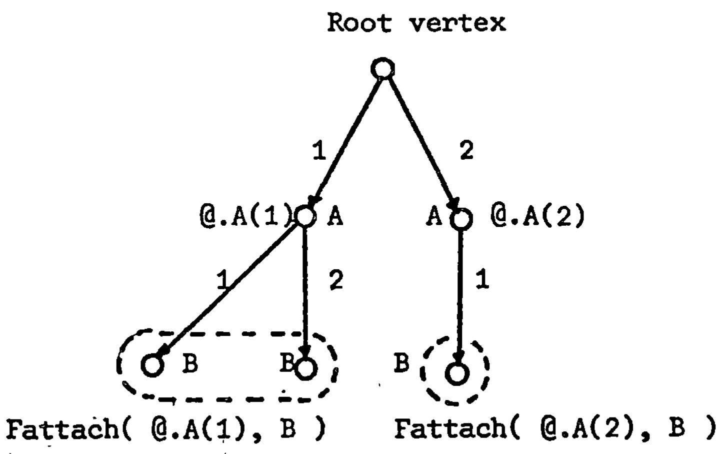


Figure E.9

As an example, the effects of the statement

```
@.A.B -> @.A.C
```

are shown in the graph of figure(E.8).

> NOTE. In order to show the effects of the attach statements with graphs, the following convention is adopted. If the graph is demonstrating an attach statement
>
> Reference1 -> Reference2
>
> which generates the new arcs (Xc1,Xr1),(Xc2,Xr2),.... then these arcs may be drawn with dashed lines in the graph. Similarly in a graph newly created vertices may be drawn with dashed circles.

(E.2.4.1) INDEX ORDERING

....

The vertices Xr2 will now have index values with respect to the vertices Xr1 that they have just been attached to. These are the indices used in Vertex_Selector_References. How these are ordered is described in this section.

Define the function Fname to be the set of vertices Xi from the set X, such that Fn(Xi) equals a given name N. This is represented by

Fname (X,N)

and is called the name set. It is a generalized version of an attached name set, where an attached name is the name set of a single vertex. Thus

Fattach (Xi,N) = Fname (Xi,N) , where Xi is a single vertex.

As an example, in the graph of figure(E.9), the attached name set of @.A(1) and of @.A(2) are indicated. The set union of these reference sets form the name set of @.A.

Just as an attached name set has an attached name list, then a name set has a name list. This name list is represented by

(Xf1, Xf2, ... XfN)

where Xf1..XfN are elements of the name set. In the following the rules used to order this is given.

Each of the vertices Xf1 is a member of Reference2. Therefore there will be a reference like

@.N1(I1).N2(I2)....Nm(Im)

which will reference each Xfi. Here Im is the index of Xfi with respect to the vertex to which it is attached, and Nm is the name of Xfi. Thus each Xfi will have associated with it one or more lists

(I1, I2, ... Im)

The ordering function is defined on these index lists. An index list (I1,I2,...Im) can be defined to be less than the index list (J1,J2,...Jn),

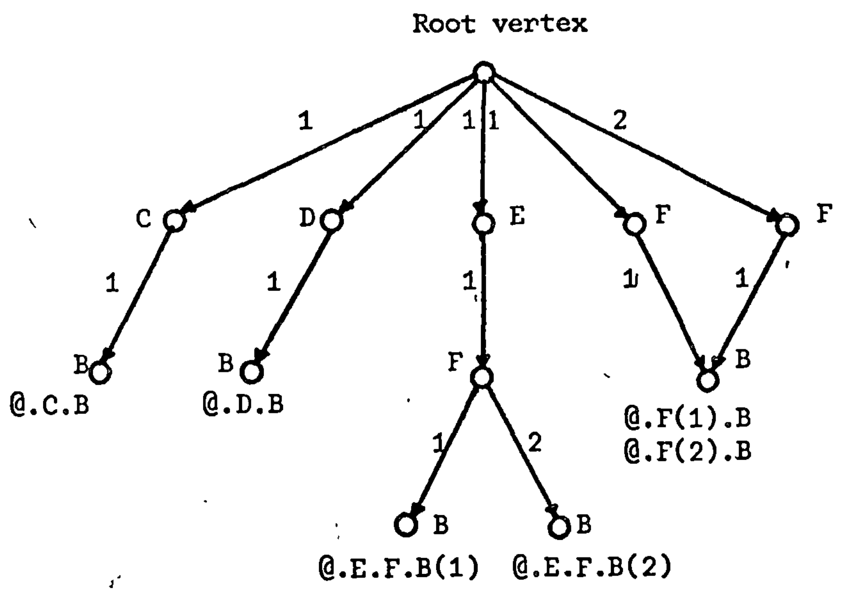


Figure E.10

If Ik = Jk for all k = 1 to p, p<n and p<m, and Ip+1<Jp+1 or
If Ik = Jk for all k = 1 to m and n>m.

If the index lists for two vertices are equal, then there are two possibilities. Either the vertices are identical, in which case the lowest index list is used to order this vertex with respect to others. Alternatively the two vertices may be different. In this case the order is undefined.

Using these ordering rules, the vertices of a reference when it is used in an attach statement can now be ordered.

Informally, the order of the vertices in one attached name set is given by its indices. If two attached names sets are combined together, then the vertices in one attached name set will come first. Which attached name set comes first is choosen on the basis of the index ordering of the parent vertices of the attached name sets. If there are more attached name sets, then each is ordered in a similar manner. Finally a vertex appearing in more than one attached name set is given the lowest ordering possible.

As an example, the graph of figure(E.10) provides the following index lists.

(1,1)	for the vertex referenced by @.C.B
(1,1)	for the vertex referenced by @.D.B
(1,1,1)	for the vertex referenced by @.E.F.B(1)
(1,1,2)	for the vertex referenced by @.E.F.B(2)
(1,1)	for the vertex referenced by @.F(1).B
(2,1)	for the same vertex as the above referenced by @.F(2).B

The index list of the reference @.F(2).B will be ignored since the same vertex is referenced by a smaller index list (1,1). The remaining index lists will be ordered like

```
(1,1) (1,1) (1,1) (1,1,1) (1,1,2)
```

where the order of the first three vertices will be undefined.

(E.2.4.2) ORDER OF VERTICES AFTER AN ATTACH

Using the ordering definition, an alternative definition of the actions of an attach statement can be given. Assume that the attached name set of a vertex Xr1 for the name N in Reference1 is Xa. Then after the execution of the statement

```
Reference1 -> Reference2
```

the attached name set of a vertex Xr1 for name N will be

```
Xa + Fname ( reference set of Reference2 , N )
```

The attached name list can be correspondingly defined as

```
( Xa1, Xa2, ... XaN, Xf1, Xf2, ... XfM )
```

where Xfi is an element of the name set of Reference2 and Xf1 is not a member of Xa. This list defines uniquely the index value of a newly attached vertex with respect to its parent vertex.

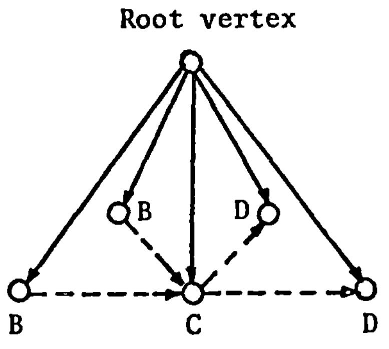


Figure E.11

(E.2.4.3) MULTIPLE ATTACH STATEMENTS
....

Given a statement like

```
    Reference_Expression1 -> Reference_Expression2 -> ...
          Reference_ExpressionN ;
```

for some N, then the action of

```
    Reference_Expression1 -> Reference_Expression2 -> ...
       Reference_ExpressionN -> Reference_ExpressionM ;
```

will be to create all the arcs of the form (Xrn,Xrm), if such an arc does not already exist, where Xrn is a member of the ReferenceN reference set, and Xrm is in the Reference_ExpressionM set. Thus the graph of figure(E.11) shows the result of the statement

```
    @.B -> @.C -> @.D
```

(E.2.4.4) NEW OPERATION
....

A new vertex can be created by a NEW statement. This has the syntax

```
    New_Operation =
       "NEW", "(", Vertex_Name, [ "=", Expression ], ")" ;
```

A statement like

```
    @ -> NEW ( Vertex_Name1 )
```

where @ represents in this case the root vertex, will create a new vertex, give it the indicated name, set its value function to null, and attach it to the root vertex.

If an attach statement is like

```
    Reference1 -> NEW ( Vertex_Name1 )
```

then this is equivalent to

```
    S1 -> Xnew1                    S1 is the set { Xr1 }
    S2 -> Xnew2                    S2 is the set { Xr2 }
       ...
    SN -> XnewN                    SN is the set { XrN }
```

In this, S1,S2,...SN are reference set variables whose values are given in the right hand side. The set {Xr1,Xr2,...Xrn} is the reference set of the reference Reference1, and Xnew1..XnewN are distinct new vertices, each having the name Vertex_Name1 and a null value.

If an Attach_Statement is like

```
    Reference1 -> NEW ( Vertex_Name1 ) -> Reference2 ;
```

then this would be equivalent to the actions

```
    Reference1 -> NEW ( Vertex_Name1 ) ;

    S1 -> Reference2               S1 is the set { Xnew1 }
    S2 -> Reference2               S2 is the set { Xnew2 }
       ...
    SN -> Reference2               SN is the set { XnewN }
```

where S1,S2,...SN are reference set variables. The set { Xnew1, Xnew2,...XnewN } contains all of the new vertices created by the attach statement

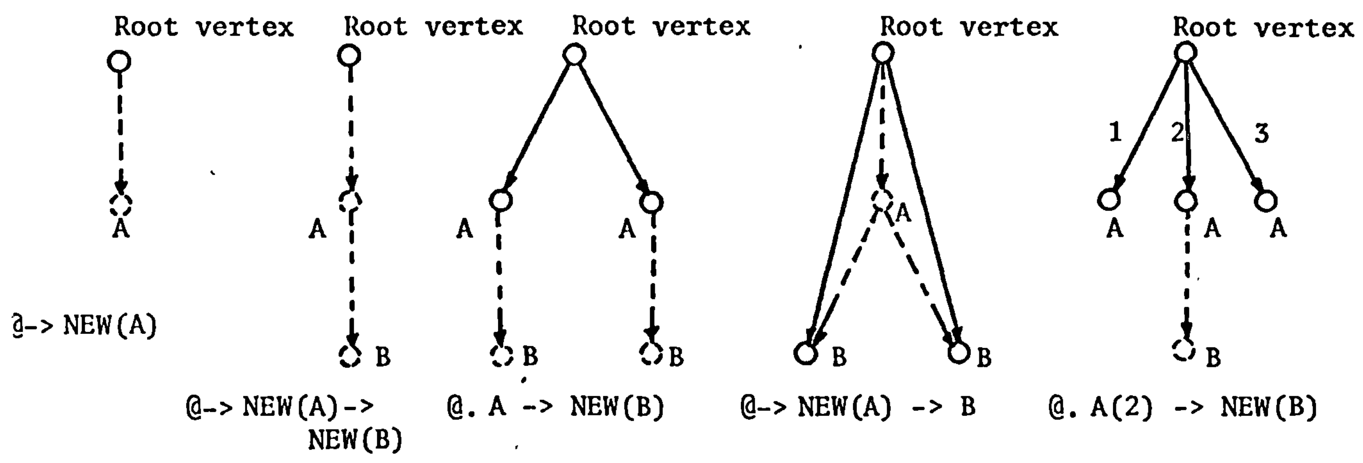


Figure EL12

```
    Reference1 -> NEW ( Vertex_Name1 ) ;
```

Another kind of NEW operation is

```
    NEW ( Vertex_Name1 = Expression1 )
```

which will also create a new vertex and attach it, except that the value of the vertex will not be null, it will be set to the indicated expression. This expression has to give a type of INTEGER, REAL, STRING or BOOLEAN. The SET type is not allowed.

An example is

```
    @ -> NEW ( A = 3 )
```

Once a vertex has been created, its name and value can not be changed. A vertex can not be destroyed. The operation of attaching the new vertex to the context base is also irreversible.

Finally, if an attach statement is like

```
    @ -> NEW ( Vertex_Name1 ) ;
```

and creates an arc (Xr1,Xr2), then

```
Findex (Xr1,Xr2) = Number of vertices in the set
                   Fattach (Xr1,Vertex_Name1)
```

In other words the vertices are numbered in the order in which they are created.

The graphs of figure(E.12) demonstrate some possible examples.

(E.2.4.5) BRACKETED ATTACH STATEMENTS

An attach statement of the form

```
Attach_Reference0 -> ( Attach_Operation1 ) ;
```

can also be represented as

```
Attach_Reference0 -> ( Attach_Reference1->... Attach_ReferenceN )
```

where the Attach_Operation has be expanded into its separate Attach_Reference parts. This statement is equivalent to the following

```
Attach_Reference0 -> Attach_Reference1 -> ... Attach_ReferenceN ;
```

A general attach statement of the form

```
Attach_Reference1 -> ( Attach_Operation1, .. Attach_OperationN ) ;
```

is equivalent in its actions to the separate attach statements

```
S := Attach_Reference1 ;
S -> ( Attach_Operation1 ) ;
   ...
S -> ( Attach_OperationN ) ;
```

where S is a reference set variable.

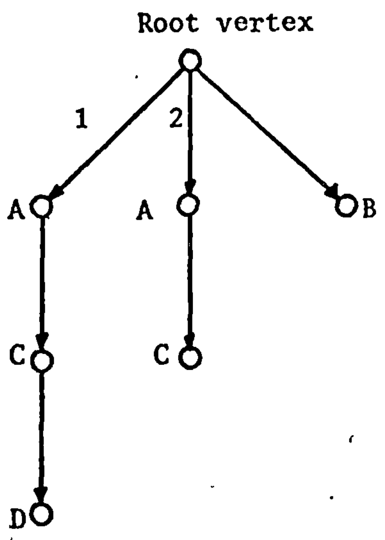


Figure E.13

(E.2.5) INITIAL CONSTRUCTION OF A GRAPH

The initial graph available, before any vertices have been created, has only one vertex. This is the root vertex, which is the vertex used in the default context of a reference or statement. Thus to create a graph like that in figure(E.13), requires the operations shown below

```
@ -> (  NEW ( A ) , NEW ( A ) , NEW ( B ) ) ;
@.A -> NEW ( C ) ;
@.A(1).C -> NEW ( D ) ;
```

(E.2.6) REPETITION CONSTRUCT

The action of a single statement may be repeated a number of times. This achieved by a For_Statement, defined by the syntax

```
For_Statement = For_Head, Statement_Block, "END", ";" ;
Statement_Block = { Statement } - ;
For_Head =
   "FOR", ( For_Number | For_Iteration | For_Each ), "DO" ;
For_Number = Integer_Expression ;
```

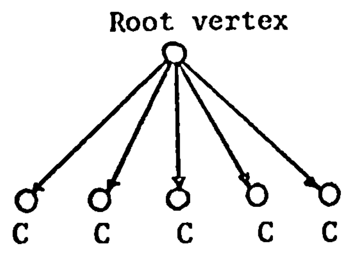


Figure E.14

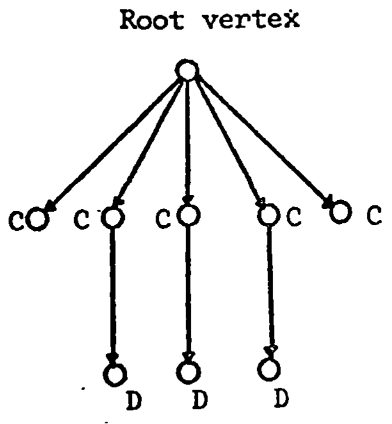


Figure E.15

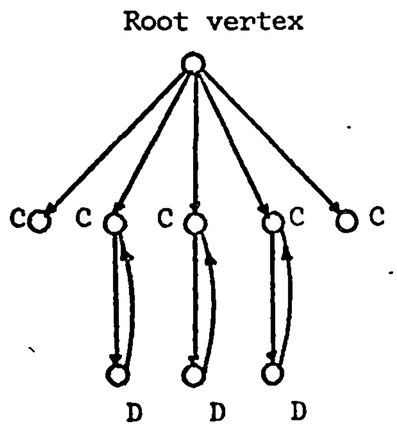


Figure E.16

```
For_Iteration = Variable_Identifier, ":=",
   Integer_Expression, "TO", Integer_Expression ;
For_Each =
   Reference_Set_Variable_Identifier, ":=",
   "EACH", "(", Reference_Expression , ")" ;
```

A For_Statement like

```
FOR Integer_Expression1 DO Statement_Block1 END ;
```

is equivalent to the reference

```
Statement_Block1 ; Statement_Block1 ; ... Statement_Block1 ;
```

where the number of Statement_Blocks is as given by the integer expression in the FOR statement. An example is

```
FOR 5 DO
   @ -> NEW ( C ) ;
END ;
```

Starting with an uninitialized graph containing only the root vertex, this For_Statement will create the graph of figure(E.14). A For_Statement like

```
FOR Variable_Identifier1 :=
   Integer_Expression1 TO Integer_Expression2 DO
   Statement_Block1 END ;
```

is equivalent to

```
Variable_Identifier1 := Integer_Expression1 ;
Statement_Block1 ;
Variable_Identifier1 := Integer_Expression1 + 1 ;
Statement_Block1 ;
   ...
Variable_Identifier1 := Integer_Expression2 ;
Statement_Block1 ;
```

Here the statement block is called once for each different value of the index variable. This takes on the values from Integer_Expression1 to Integer_Expression2 inclusive.

An example of this is

```
FOR I := 2 TO 4 DO
   @.C(I) -> NEW ( D ) ;
END ;
```

If this For_Statement starts with the graph of figure(E.14) then the graph of figure(E.15) will be constructed.

A For_Statement like

```
FOR Reference_Set_Variable_Identifier :=
   EACH ( Reference_Expression ) DO
   Statement_Block1 END ;
```

will be equivalent to the following statements

```
Statement_Block1 ;   Reference_Set_Variable_Identifier1 is { X1 }
Statement_Block1 ;   Reference_Set_Variable_Identifier1 is { X2 }
   ...                                ...
Statement_Block1 ;   Reference_Set_Variable_Identifier1 is { XN }
```

where the text in the right hand side is not part of the ISL but indicates what value the Reference_Set_Variable_Identifier1 has. The set { X1, X2,...XN } if equal to the reference set obtained from Reference1. In other words, this sets the Reference_Set_Variable_Identifier to each of the vertices in the reference, performing the Statement_Block once for each. An example is the statement

```
FOR S := EACH ( @.C.< NOT EMPTY ( @.D ) > ) DO
   S.D -> S ;
END ;
```

which, if it starts with the graph in figure(E.15), will create the graph of figure(E.16).

(E.2.7) IF STATEMENTS

A conditional statement may be used to govern the execution of a statement block. The syntax of an If_Statement is

```
    If_Statement = "IF", Conditional_Expression,
       "THEN", Statement_Block, [ "ELSE", Statement_Block ] ;
```

An if statement like

```
    IF Conditional_Expression1 THEN Statement_Block1 ;
```

will be equivalent to the following

```
    Statement_Block1
```

if the condition is true. If the condition is false then the If_Statement has no action. An if statement like

```
    IF Conditional_Expression1 THEN Statement_Block1
                                    ELSE Statement_Block2 END ;
```

is equivalent in its results to

```
    Statement_Block1 ;
```

if the condition is true. If the condition is false then the If_Statement is equivalent to

```
    Statement_Block2 ;
```

An example IF statement is

```
    IF I > 2 THEN
       @ -> NEW ( A ) ;
    END ;
```

If I is greater than 2, then this statement will create a new A vertex. Another example is

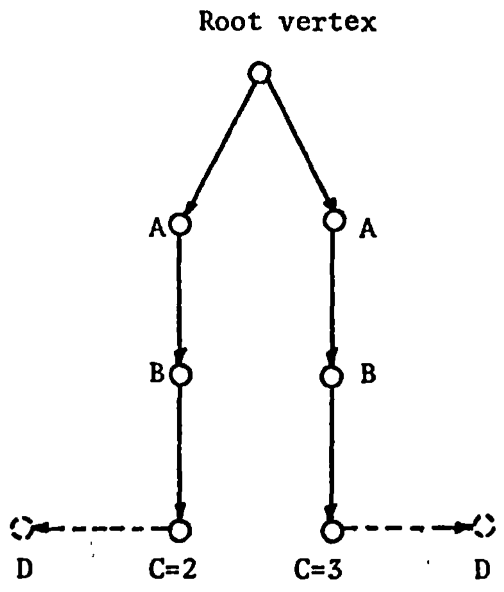


Figure E.17

```
IF NOT EMPTY ( @.A.B.< ANY_VALUE(@.C)=3 > ) THEN
   @.A.B.C -> NEW (  D ) ;
END ;
```

which will create a new D Vertex for the C vertices in the reference A.B.C only if there exists at least one C vertex with a value of 3. The action of this is visible in the graph of figure(E.17).

(E.3) DECLARATIONS

In the preceding sections the basic graph creation and access statements have been defined. These statements have used various kinds identifiers. In the following the syntax of the declarations that are used to define these names is given.

(E.3.1) CONSTANT IDENTIFIERS

The constant definition part defines identifiers that represent constant values. There after these identifiers may be used in the specifications in place of an actual constant value. The constants may be real, integer, string or boolean.

The syntax of a constant definition part is

```
    Constant_Definition_Part =
       "CONST", One_Constant_Definition,
       { ",", One_Constant_Definition }, ";" ;
    One_Constant_Definition = Identifier, "=", Constant_Value ;
    Constant_Value = [ Unary_Sign ]  Unsigned_Integer   |
                     [ Unary_Sign ] Unsigned_Real |
                     String |
                     True | False ;
```

An example constant definition part is

```
    CONST A=3.4, B=2, C='string', D=True ;
```

(EL3.2) VERTEX IDENTIFIERS

The vertex definition defines identifiers that may be used as the names of vertices in NEW operations. As well the ISL may provide some predefined vertex names, depending upon the requirements of the resource allocator application. Only names defined by the user or predefined names may be used to create and to refer to vertices. The syntax is

```
    Vertex_Definition_Part =
       "VERTEX", Identifier, { ",", Identifier }, ";" ;
```

and an example is

```
    VERTEX  A , B , C ;
```

(EL3.3) VARIABLE IDENTIFIERS

The variables are defined in a Variable_Declaration_Part with the syntax

```
Variable_Declaration_Part =
   "VAR", { Var_Declaration_List, ";" }- ;
Var_Declaration_List =
   Identifier, { ",", Identifier }, ":", Var_Type, ;
Var_Type = "INTEGER" | "STRING" | "REAL" | "BOOLEAN" | "SET" ;
```

An example is

```
VAR
   A, B : INTEGER ;
   S    : SET ;
```

which will define two integer variables of names A and B, and define a reference set variable of name S.

(E.3.4) PROCEDURE DEFINITIONS

A procedure defines a Statement_Block and gives it a name. Thereafter this statement block may be invoked by using this name. The syntax of a procedure definition is

```
Procedure_Definition =
   "PROCEDURE", Procedure_Name_Identifier,
   [ Formal_Parameter_List ], Local_Definition_Part,
   "BEGIN", Statement_Block, "END", ";" ;

Local_Definition_Part = [ Constant_Definition_Part ],
                        [ Vertex_Definition_Part   ],
                        [ Variable_Definition_Part ] ;

Formal_Parameter_List = "(", One_Formal_Parameter,
   { "," , One_Formal_Parameter }, ")" ;
One_Formal_Parameter = Identifier_List, ":", Var_Type ;
Identifier_List = Identifier, { "," , Identifier } ;
```

The procedure may be called by using a procedure call statement. The syntax of this is

```
Procedure_Call =
   Procedure_Name_Identifier, [ Actual_Parameter_List ], ";" ;

Actual_Parameter_List = "(", Actual, { ",", Actual }, ")" ;
Actual = Expression ;
```

(EL3.4.1) PARAMETER LISTS
....

A procedure defined without a formal parameter list, as in

```
PROCEDURE Identifier1 ;
BEGIN Statement_Block1 END ;
```

can be called with a procedure call without any actual parameters. Thus

```
Identifier1 ;
```

A procedure defined with a formal parameter list containing a One_Formal_Parameter like

```
Identifier1, Identifier2, ... IdentifierN : Var_Type1
```

is equivalent to a procedure defined with a formal parameter list like

```
Identifier1 : Var_Type1 ; Identifier2 : Var_Type1 ;
   ... IdentifierN : Var_Type1 ;
```

A procedure defined with a nonempty formal parameter list such as

```
PROCEDURE Identifier1
   ( Identifier1 : Var_Type1 ;
     Identifier2 : Var_Type2 ; ...
     IdentifierN : Var_TypeN ) ;
   Statement_Block1
END ;
```

is called by a Procedure_Call of the form

```
Identifier1 ( Actual1, Actual2, ... ActualN ) ;
```

where there are the same number of Actual parameters as there are formal parameter identifiers. Furthermore the types of the corresponding actual and formal parameters must agree.

(E.3.4.2) PROCEDURE SEMANTICS

Given a procedure of the form

```
PROCEDURE Identifier1 Formal_Parameter_List1 ;
   Local_Definition_Part1
BEGIN
   Statement_Block1
END ;
```

then a procedure call to this procedure like

```
Identifier1 Actual_Parameter_List1 ;
```

will have the same actions as an equivalent group of statements constructed by modifying the statements of Statement_Block1. If there is a formal parameter IdentifierF in the procedure declaration, then there will also be a corresponding actual parameter ActualP. The equivalent statements are constructed by replacing every mention of IdentifierF by ActualP.

All the identifiers defined inside the procedure and thus used in the above equivalent statements will need to be defined in equivalent definitions. Thus every identifier in the Local_Definition_Part of the procedure will be defined with the same type in the equivalent definitions.

(E.3.4.3) EXAMPLES

....

An example procedure definition is

```
PROCEDURE P ( S :  SET ) ;
BEGIN
   S := @.A.<@.C=6> ;
END ;
```

and this may be called by

```
P ( S ) ;
```

This will return a reference set variable which refers to all vertices in the reference

```
@.A.<@.C=6>
```

Another example is

```
PROCEDURE P ( VALUE : STRING ; S :  SET ) ;
BEGIN
   S -> NEW ( C = VALUE ) ;
END ;
```

If this is called with the reference

```
P ( 'string' , @.A ) ;
```

then this reference is equivalent to

```
@.A -> NEW ( C = 'string' ) ;
```

Procedures may be called recursively. Thus a procedure may be defined as

```
PROCEDURE TREE ( S :  SET ; LEVEL : INTEGER ) ;
SET L ;
BEGIN
```

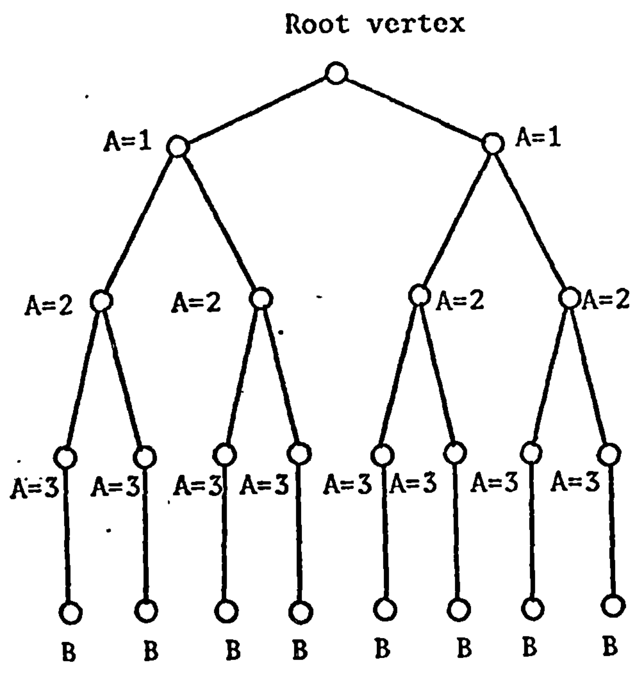


Figure E.18

```
    IF LEVEL <= MAX_LEVEL THEN
       L := NEW ( A = LEVEL ) ;
       S -> L ;
       TREE ( L , LEVEL+1 ) ;
       L := NEW ( A = LEVEL ) ;
       S -> L ;
       TREE ( L , LEVEL+ 1 ) ;
    END ;
 END ;
```

In this the L set reference set variable is used as a temporary reference to the new A vertex. Each A vertex is created, attached to the reference set S, and then used in a further recursive call to the TREE procedure. If Max_Level is equal to 3 and the procedure is called as in the following,

```
TREE ( @ , 1 ) ;
```

will be equivalent to the Attach_Statements

```
@ ->
( NEW(A=1) ->
   ( NEW(A=2) ->
      ( NEW(A=3) , NEW(A=3) ),
      NEW(A=2) ->
      ( NEW(A=3) , NEW(A=3) )
   ),
   NEW(A=1) ->
   ( NEW(A=2) ->
      ( NEW(A=3) , NEW(A=3) ),
      NEW(A=2) ->
      ( NEW(A=3) , NEW(A=3) )
   )
) ;
```

The actions of the statements

```
TREE ( @ , 1 ) ;
@.A.A.A -> NEW ( B ) ;
```

are shown in figure(E.18).

(E.4) BRINGING THE DECLARATIONS AND STATEMENTS TOGETHER

The part of the ISL that creates graphs is contained in one specification block. This contains the references and identifier definitions which have been discussed above. Its syntax is

```
Graph_Specification_Block =
   "GRAPH",
      [ Constant_Definition_Part ] ,
      [ Vertex_Definition_Part ] ,
      [ Variable_Definition_Part ] ,
      { Procedure_Definition },
   "BEGIN"
      Statement_Block,
   "END", ";" ;
```

A example graph definition is

```
GRAPH
   CONST MAX_LEVEL = 3 ;
   VERTEX A,B ;
   PROCEDURE TREE ... as in the above definition ... ;
   BEGIN
      TREE ( @ , 1 ) ;
      @.A.A.A -> NEW ( B ) ;
END ;
```

and this is a complete legal definition to obtain the graph structure of the figure(EL.18).

(EL4.1) SCOPE OF IDENTIFIERS

The various identifiers defined in the specifications have defined scopes over which they may be used. An identifier may be defined globally with respect to the graph specification block. Such identifiers are those defined as constants, vertex names, procedure names and variables. Alternatively identifiers may be defined locally with respect to a procedure. These are the formal parameter identifiers, and the identifiers in the local definition part.

A global identifier may be only defined once. Once defined it retains this definition and may be used through out the graph specification block. A global identifier can not be redefined as a local identifier inside a procedure.

A local identifier may not be redefined with the same procedure. The definition of a local identifier is local to the procedure only, and different procedures using the same local identifiers have independent definitions for the identifier. The scope of a local identifier is the whole of the procedure block.

APPENDIX (F)

(F.1) SHORT COMPLETE SPECIFICATION PROGRAM

In the following a complete example will be developed for a problem similar to the instrument monitoring problem introduced in the introduction. Firstly the computer architecture is described

```
ALLOCATOR
   GRAPH
      VERTEX
         INPUT_OUTPUT , TTY_ATTACHED ;
      VAR
         J : INTEGER ;
         PSR : SET ;

   PROCEDURE STANDARD_MEMORY (
      C : SET ; START_VALUE , SIZE_VALUE : INTEGER ) ;
   VAR MEM : SET ;
   BEGIN
      MEM := NEW ( ADDRESS ) ->
      (  NEW ( START = START_VALUE ) ,
         NEW ( MEMORY ) ->
         (  NEW ( SIZE = SIZE_VALUE ) ,
            NEW ( ACCESS = 0.45
         )
      ) ;
      C -> MEM ;
   END ;

   PROCEDURE ONE_PROCESSOR ( C , PSR : SET ) ;
   BEGIN
      PSR := NEW ( PROCESSOR ) ;
      C -> PSR ->
      (  NEW ( CYCLE = 2.5 ) , NEW ( NAME = 'BRANDX' ) ) ;
   END ;
```

```
PROCEDURE MAP ( PSR : SET ) ;
VAR ME, P : SET ;
    I : INTEGER ;
BEGIN
   ME -> NEW ( MEMORY_ACCESS ) ;
   PSR -> ME ;
   I := 1 ;
   FOR P := EACH ( PSR ) DO
      ME -> NEW ( ADDRESS ) ->
      (  NEW ( START = 8192 * I ) , P.ADDRESS.MEMORY ) ;
      I := I + 1 ;
   END ;
END ;

BEGIN
   FOR 10 DO
      ONE_PROCESSOR ( @ , PSR ) ;
      STANDARD_MEMORY ( PSR , 0 , 8192 ) ;
   END ;
   MAP ( @.PROCESSOR ) ;

   @ -> NEW ( INPUT_OUTPUT ) ;
   FOR J := 11 TO 20 DO
      @.INPUT_OUTPUT -> NEW ( READ_WRITE_PORT = J ) ;
   END ;

   @.PROCESSOR ->
   (  NEW ( INTERRUPT = 0 ) , @.INPUT_OUTPUT ) ;

   FOR J := 1 TO 2 DO
      @.PROCESSOR(J) -> NEW ( TTY_ATTACHED ) ->
         NEW ( PORT ) -> NEW ( READ_WRITE_PORT = 4 ) ;
   END ;

   @ -> NEW ( TTY_PROCESSOR ) ->
     -> @.PROCESSOR. < NOT EMPTY ( @.TTY_ATTACHED ) > ;
    END ;
```

This defines a computer system with ten processors. Each accesses a local memory of 8096 bytes and has indirect access via a common bus to all the other memories of the system. Each processor also has an interrupt at address 0. Each processor can access the same group of input/output ports which are numbered 11 to 20. As well processors 1 and 2 have a TTY port attached,. indicated by the TTY_ATTACHED vertex. To allow direct reference to these two processors, their vertices are attached to the TTY_PROCESSOR vertex.

The instrument monitoring program takes the form

```
  PROGRAM INSTRUMENT_MONITOR ;
     PROCESS INSTRUMENT_1 ; ...
     PROCESS INSTRUMENT_2 ; ...
     . . .
     PROCESS INSTRUMENT_10 ; ...
     MAIN PROCESS
        ( which accesses the TTY ports )
     END ;
  END ;(* this text is separate from the allocation specification *)
```

Here it is assumed for the sake of the example that the main process is the only process that accesses the TTY ports. The user programmer will now need to provide the constraint specification to insure that the main process is assigned to a processor that accesses this TTY port. Thus an object specification is required to indicate this,

```
  OBJECT
     DEFINITION
        MAIN_PROCESS : PROCESS ;
     END ;
     SPECIFICATION
        MAIN_PROCESS := [ INSTRUMENT.MAIN ] ;
     END ;
  END ; (* this is part of the allocation specification *)
```

This object specification is used in a constraint block like

```
        CONSTRAINT
           ASSIGN ( MAIN_PROCESS ) -> [ TTY_PROCESSOR ] ;
        END ;
     END (* of complete allocation specification *) ;
```

The main process will be assigned to either one of the first two processors. The other processes and all the memories, which have not been mentioned in any directives, will be assigned by the allocator to achieve the maximum throughput.

REFERENCES

[1] Am2900 Bipolar Microprocessor family
Proceedings Micro 8, 8th Workshop on microprocessors, Page 75.

[2] Preliminary Ada Reference manual, Rationale for the
Design of the ADA programming language.
ACM Sigplan Notices, Vol 14, No 6(June 79).

[3] L.H.Anderson, R.M.Larsen
Distributed intelligence microcomputer systems for
industrial control
Microprocessor InfoTech State of the Art Report, No 35(1977).

[4] G.A.Anderson, E.D.Jensen
Computer interconnection structures, taxonomy, characteristics
and examples.
Computer surveys, Vol 7, No 4(Dec 77), Page 197.

[5] B.Appelbe, M.Kroening
Concurrent programming on microprocessors.
Sigsmall newsletter, Vol 15, No 2(1979).

[6] J.Armstrong
Fault diagnosis in a boolean N cube of microprocessors.
IEEE transactions on computers(Aug 81), Page 587.

[7] D.Aspinall
Comparsion of microprocessors:
Instruction processor level,
Processor memory switch level.
Microprocessor InfoTech State of the Art Report, Vol 2,
No 35(1977).

[8] F.Baskett, A.J.Smith
Interference in multiprocessor systems with interleaved
memories.
Communications of the ACM, Vol 19, No 6(June 76), Page 327.

[9] G.J.Burnett, E.G.Coffman
Analysis of interleaved memory system using blockage buffers.
Communications of the ACM, Vol 18, No 12(1975), Page 91.

[10] D.W.Bustard
Pascal Plus Users Manual
Queens University of Belfast (Aug 78).

[11] A.Celentano, et al .
Seperate compilation and partial specification in Pascal.
IEEE Software Engineering, Vol 6, No 4(July 80).

[12] O.Cert
Parallelism, control and synchronization expressions
in a single assignment language.
Sigplan Notices, Vol 13, No 1(Jan 78).

[13] E.G.Coffman
Operating system theory (1973).

[14] E.L.Daglees
A multimicroprocessor : CYBA-M*.
IFIP (1977), Page 843.

[15] O.J.Dahl, E.W.Dijkstra, C.A.R.Hoare
Structured programming (1973).

[16] J.B.Dennis
Modularity
Lecture notes in computer science, No 30(1975).

[17] A.M.Despain, D.A.Patterson
X Tree, a tree structured multiprocessor computer
architecture.
5th Annual symposium on computer architecture (1978), Page 144.

[18] E.W.Dijkstra
A Discipline of programming (1976).

[19] D.J.Farber
Software considerations in distributed architectures.
Computer, Vol 7, No 3(Mar 74), Page 31.

[20] E.T.Fathi, M.Krieger
Multiple microprocessor systems: What, Why, and When.
Computer, Vol 16, No 3(Mar 83), Page 23.

[21] R.A.Finkel, M.H.Solomon
Processor interconnection strategies.
IEEE Transactions on Computers, Vol C29, No 5(May 80), Page 362.

[22] M.J.Flynn
Some computer organizations and their effectiveness.
IEEE Transactions on Computers, Vol C21, No 9(Sept 72), Page 948.

[23] E.C.Freuder
Synthesizing constraint expressions.
Communications of the ACM, Vol 21, No 11(Nov 78), Page 958.

[24] S.H.Fuller, et al
Multimicroprocessors: an overview and working example.
Proceedings of the IEEE, Vol 66, No 2(Feb 78), Page 216.

[25] E.F.Gehringer, et al
Cm* test bed.
Computer (Oct 82), Page 40.

[26] E.F.Gehringer, R.J.Chansler
StarOS user and system structure manual.
Department of computer science, Carnegie-Mellon university, Pittsburg, Pennsylvania (1981).

[27] A.M.Geoffrion
Integer programming by implicit enumeration and Balas method.
SIAM Review, Vol 9, No 2(April 67), Page 178.

[28] M.Georgeff
Strategic search.

Australian computer science communications, Vol 2, No 1(Jan 80).

[29] R.Gleaves
Modula 2 Users Manual (Nov 82).

[30] A.Gottlieb, J.T.Schwartz
Networks and algorithms for very large scale parallel computers.
Computer, Vol 15, No 1(Jan 82), Page 27.

[31] A.N.Habermann
Path expressions
Carneige Mellon Tech Report (1975).

[32] A.N.Habermann, Campbell
The specification of process synchronization by path expressions.
Lecture notes in computer science, Vol 16(1974), Page 89.

[33] K.Haessig,C.Jenny
Partitioning and allocating computational objects in distributed computer systems.
IFIP 80(1980), Page 593.

[34] P.B.Hansen
Concurrent Pascal Report (June 75).

[35] P.B.Hansen
Distributed processes, a concurrent programming concept.
Communications of the ACM, Vol 21, No 11(Nov 78), Page 934.

[36] P.B.Hansen
The programming language Concurrent Pascal
IEEE transactions on software engineering, Vol 1, No 2(June 75).

[37] P.B.Hansen
A multiprocess program.
IEEE Computer science and applications conference, Chicago, Illinois (Nov 77).

[38] P.B.Hansen
Operating system principles (1973).

[39] A.C.Hartmann
A concurrent Pascal compiler for mini computers.
Lecture notes in computer science, No 50(1977).

[40] L.S.Haynes, R.L.Lau, D.P.Siewiorek, W.Mizell
A survey of highly parallel computing
Computer, Vol 15, No 1(Jan 82), Page 9.

[41] C.A.R.Hoare
Monitors, an operating system concept.
Communications of the ACM, Vol 17, No 10(Oct 74), Page 549.

[42] C.A.R.Hoare,R.H.Perrott
Operating systems techniques (1972).

[43] C.A.R.Hoare
Communicating sequential processes.
Communications of the ACM, Vol 21, No 8(Aug 78), Page 666.

[44] C.Hoogendoorn
A general model for memory interference in multiprocessors.
IEEE Transactions on computers, Vol c-26, No 10(Oct 77),
Page 998.

[45] J.G.Hunt
Interrupts.
Software Practice and Experience, Vol 10(1980), Page 523.

[46] A.K.Jones, R.Chansler, I.Durham, P.Feiler, K.Schwans
Software management on Cm*- a distributed multiprocessor.
AFIPS conference proceedings, Vol 46(1977), Page 657.

[47] A.K.Jones, P.Schwarz
Experience using multiprocessor systems: a status report.
ACM Computing Surveys, Vol 12, No 2(June 80).

[48] A.Kaufmann
Graphs, dynamic programming and finite games (1967).

[49] J.L.Keedy
On structuring operating systems with monitors.
Australian computer journal, Vol 10, No 1(1978).

[50] J.L.Keedy
The Monads operating system
Proceedings of the 8th Australian computer conference
in Canberra.

[51] J.L.Keedy
The influence of the information hiding principle on the
Monads operating system.
Proceedings of the Australian University Computer Science
Seminar, University of New South Wales (1978).

[52] P.B.Kieburtz,J.L.Hennesy
Tomal, a high level language for micro processor
Control application..
Sigplan Notices, Vol 11, No 4(April 76), Page 127.

[53] W.A.Kornfeld
Combinatorially implosive algorithms.
Communications of the ACM, Vol 25, No 10(Oct 82).

[54] B.Kumar, E.S.Davidson
Performance evaluation of highly concurrent computers by
deterministic simulation.
Communications of the ACM, Vol 21, No 11(Nov 78), Page 904.

[55] H.T.Kung
Why systolic architectures.
Computer, Vol 15, No 1(Jan 82), Page 37.

[56] B.W.Lampson, J.J.Horning, R.L.London, J.G.Mitchell
Report on the programming language Euclid.
Acm Sigplan Notices, Vol 12, No 2(Feb 77).

[57] E.J.Lau,D.Ferrari
Program restructuring in a multilevel virtual memory.
IEEE Transactions on Software Engineering, Vol SE-9,
No 1(Jan 83).

[58] W.Y.P.Lim
HIDSL a structure description language.
Communications of the ACM, Vol 25, No 11(Nov 82).

[59] G.J.Lipovski
Hardware description languages,
Computer, Vol 10, No 6(June 77), Page 14.

[60] A.M.Lister
Fundamentals of operating systems (1975).

[61] M.D.Maples, E.R.Fisher
Real time micro computer applications using LLL Basic.
Computer, Vol 10, No 9(Sept 77), Page 15.

[62] T.A.Marsland,M.Campbell
Parallel search of strongly ordered game trees.
ACM Computing Surveys, Vol 14, No 4(Dec 82).

[63] T.J.Miller, R.H.Campbell
A Path Pascal Language
Department of Computer Science, University of Illinois at
Champaing-Urbana, Urbana-Illinois 61801, No 217-333-0215
(April 78).

[64] J.Montuelle, J.Mossiere, J.L.Cheval, F.Cristian, S Krakowiak
An experiment in modular program design,
IFIP (1977), Page 23.

[65] K.T.Narayana, V.R.Prasad, M.Joseph
Some aspects of concurrent programming in Concurrent Pascal
Software Practice and Experience, Vol 9, No 9(1979), Page 749.

[66] J.K.Ousterhout, D.A.Scelza, S.S.Pradeep
MEDUSA, an experiment in distributed operating system

structure.
Communications of the ACM, Vol 23, No 2(1980), Page 92.

[67] D.L.Parnas
A technique for software module specification
Communications of the ACM, Vol 15, No 5(1972), Page 330.

[68] D.L.Parnas
Information distribution aspects of design methodology
IFIP (1971).

[69] D.L.Parnas
On the criterion to be used in the decomposition of systems into modules.
Communications of the Acm, Vol 15, No 12(1972), Page 1053.

[70] J.H.Patel
Processor memory interconnections for multimicroprocessor, performance.
IEEE transactions on computers, C30, No10(Oct 81), Page 771.

[71] B.Peuto
Z8000 Architecture, Cpu and memory management unit.
Computer, Vol 12, No 2(Feb 79), Page 10.

[72] J.L.Potter
Image processing on a massively parallel processor.
Computer, Vol 16, No 1(Jan 83), Page 62.

[73] E.M.Reingold,J.Nievergelt
Combinatorical algorithms (1977).

[74] E.S.Roberts et al
ADA task control.
Software Practice and Experience (Oct 81), Page 1019.

[75] M.Satyanarayanan
Multiprocessors: a comparative study.

[76] F.B.Schneider, A.J.Bernstein
Scheduling in Concurrent Pascal.
Operating System review, Vol 12, No 2(Apr 78).

[77] R.S.Scowen
An introduction and handbook for the standard syntactic metalanguage.
NPL Report DITC 19/83(Feb 83).

[78] R.J.Shan,S.H.Fuller,D.P.Siewiorek
Cm*, a modular multimicroprocessor.
AFIPS Conference Proceedings, Vol 46(1977), Page 637.

[79] H.J.Siegel
A model of SIMD machines, and a comparsion of various interconnection networks.
Proceedings of the IEEE transactions on computers, No 12, Vol C-28(Dec 79), Page 907.

[80] H.J.Siegel
Partionable SIMD/MIMD system for image processing and pattern recognition
IEEE transactions on computers, Vol C30, No 12(Dec 81), Page 934.

[81] D.P.Siewiorek, D.E.Thomas, D.L.Scharfetter
The use of LSI modules in computer structures, trends and limitations.
Computer, Vol 11, No 7(July 78), Page 16.

[82] A.Silberschatz, R.Kieburtz, A.Bernstein
Extending Concurrent Pascal to allow dynamic resource management.
IEEE transactions on software engineering, Vol SE-3, No 3(May 77).

[83] A.J.Smith
Multiprocessor memory organization and memory interference
Communications of the ACM, Vol 20, No 10(Oct 77), Page 754.

[84] L.Snyder
Introduction to the configurable highly parallel computer.
Computer, Vol 15, No 1(Jan 82), Page 47.

[85] J.H.Stewart
LOGAL: A computer hardware description language
for logic design and synthesis of computers.
Computer, Vol 10, No 6(June 77), Page 18.

[86] E.Stritter
Motorolla 68000 architecture
Computer, Vol 12, No 2(Feb 79), Page 43.

[87] S.Y.H.SU
A survey of computer hardware description languages in the
U.S.A.
Computer, Vol 7, No 12(Dec 74), Page 45.

[88] P.R.Torrigiani, M.W.Shields, P.E.Lauer
Cosy, a system specification language based upon paths
and processes
Acta Informatica, Vol 12, No 2(1979), Page 109.

[89] N.I.Vilenkin
Combinatorics (1971).

[90] I.C.Wand, J.Holden
Experience with the programming language Modula
IFAC/IFIP real time programming workshop (1977).

[91] A.J.Weissberger
Application ideas for microprocessors.
Instrument control system, Vol 48, No 10(Oct 75), Page 19.

[92] J.Welsh, D.W.Bustard
Pascal plus
Software Practice and Experience, Vol 9, No 11(Nov 79), Page 947.

[93] N.Wirth
Modula, a language for modular Multiprogramming
Software, Practice and Experience, No 7(1977), Page 3.

[94] N.Wirth
Toward a discipline of real time programming.
Communications of the ACM, Vol 20, No 8(Aug 77), Page 577.

[95] D.Wright
Microcomputers, Fundamentals and applications,
"Microprocessor survey" (1974).

[96] S.J.Young
An introduction to ADA (1983).

[97] S.Zeigler
Intel 432 microcomputer supports ADA language.
Computer(June 81), Page 47.