AN ARCHITECTURAL STUDY ON MASSIVE MULTIPROCESSOR SYSTEMS

by

Reiji Aibara

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Circuits and Electrical System Engineering)
in Hiroshima University

January 1986

Abstract

The computer architecture has been explored for higher performance, higher facilitate and/or more reliable systems at lower costs (sometimes at any cost). Parallel processing with multiprocessors has been employed by many researchers as a suitable technology for the improvements, and has been realized in experimental or commercial machines consisting of up to 10^2 processors. In particular, a lot of proposals for new supercomputer architectures aimed at increasing machine performance by an order of magnitude have come out in the past several years. Decreasing costs and increasing density of CPU and memory chips due to the recent advanced VLSI technology have made such computer architectures feasible even if it is a **Massive Multiprocessor System** (in short, MMS) configuring more than 10^3 processing elements. However, there remain a lot of problems to be solved toward realization of the MMS's efficiently performing a job.

The goal of this dissertation is to provide a design methodology of such MMS's based on the architecture aimed at increasing their performance. Though several levels of the architectures are investigated, we mainly focus on the PMS (Processor-Memory-Switch) level because systems based on the architecture allow the design flexibility of their parallelism, and have great possibility of a realization at high cost-performance.

On the basis of several experiments using multiprocessor UNIP with 32 processors, the dissertation describes a massive

multiprocessor simulator for performance evaluation and interconnection networks for MMS's based on a new device technology, i.e., 3-dimensional integrated circuits.

First, Chapter 1 surveys studies on computer architectures toward higher performance of computing systems in various levels.

In Chapter 2, multiprocessor approaches are presented. Basic parallel processing schemes and typical multiprocessor configurations are summarized, and then, a fabrication of experimental multiprocessor system UNIP is described. After several experiments using UNIP are demonstrated, essential and crucial issues of multiprocessors derived from that experience are summarized.

In Chapter 3, a modeling of MMS programs for performance evaluation using the parallel programming scheme is proposed. The model which is largely intuitive, is applicable to a simulater for the performance evaluation of MMS's in which the interprocessor communication cost can be measured. After a description language of the programming scheme is described, the simulator implemented on the UNIX system and simulation analysis on the experimental results are demonstrated.

In Chapter 4, a new type of common memory (in short, 3-D CM) based on a technology of 3-dimensional integrated circuits is proposed and its fundamental properties are described. A communication module for connecting processors using 3-D CM and processor interconnection networks for MMS's, consisting of the modules are demonstrated. A brief analysis of the network performance is also described.

Finally, in Chapter 5, we discuss and summarize further problems to realize high-performance MMS's.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In the quest for higher levels of computational performance, systems have been planned to have throughputs in excess of one billion instructions per second (BIPS). Over the several years we have seen myriad proposals for supercomputers, especially, new architectures aimed at increasing machine performance by an order of magnitude. As a result, it has been possible to achieve such high levels of performance only for very specialized systems such as signal processors. Such systems are optimized to execute a few well-formed algorithms, so that it is difficult to effective implement other application problems.

Recently, it has become clear that there are many applications (e.g., image recognition, computer tomography, data base management, simulation for weather forecast, etc.) which require high levels of performance. The variety of the applications requires general purpose flexibility for systems.

On the other hand, the advent of large and very large scale semiconductor integrated techniques has reduced the cost of digital circuits. In particular, significant reduction of the cost has been done for memory and CPU chips. Hence, an inexpensive realization of a system having a wide spectrum of applications is becoming possible.

The dissertation addresses design methodologies to develop systems satisfying the following two requirements:

(i) High performance, e.g., exceeding one BIPS.

(ii) General purpose computational flexibility.

There exist two main approaches to the development of the supercomputers:

(1) The use of extremely high-speed single processor based on conventional von Neumann architectures.
(2) The use of new architectures, e.g., parallel processing.

The first approach has been due to the significant improvement of the device technology in the recent years. However, since a signal transfer speed in a circuit cannot exceed the velocity of the light in principle, there is an apparent limitation of the improvement. It is difficult to develop a device which is hundred times faster than the currently existing chips.

This dissertation adopted the second approach. It has been employed by several computer researchers and designers, and a various kind of advances in computer architecture has been dene. In [MYERS82], Myers defines the term **computer architecture** as "the distribution of functions across a proposed level or boundary within a system, and the precise definition of the boundary". According to this definition, a research of the computer architecture can be divided into a lot of levels. In particular, parallel processing techniques for high performance computation are investigated by the most of researchers at several architecture levels, which are classified into the following three typical levels [BELL71]:

(a) RT (Register-Transfer) level,

(b) ISP (Instruction-Set-Processor) level, and

(c) PMS (Processor-Memory-Switch) level.

The major distinction among RT, ISP, and PMS levels is the granularity of their parallelism. RT, ISP, and PMS level architectures are suitable for sub-instruction, instruction, and process (or task) level parallel processing, respectively.

Parallel processors based on the RT level architectures execute an instruction in parallel. An example of this type of architectures is a vector processor which divides an instruction into several pipelining stages or a vectorized datum into elements and executes them simultaneously. Therefore the division of functions and scheduling of their execution must be decided at the designing time. It is impossible for users to change them after the design has completed. A typical example of the parallel processing systems based on the ISP level architectures is a data flow machine. The system has several execution units which perform an instruction when all required data (or operands) of the instruction meet at the units. The RT and ISP architectures allow high granularity of parallel processing; however, they have little flexibility in their execution.

In contrast, systems based on the PMS level architectures such as multiprocessor systems, execute a process (or a task) in parallel. Modules composing the systems are autonomous processing elements or processors, which can be employed under centralized or distributed control. It is true the PMS architectures allows the systems to apply a wide spectrum of applications, but it was said that a drawback in them was that the

modules become large and complex. However, decrease in costs and increase in the number of CPU's and memories compressible in a chip due to the recent advance of the VLSI technology have made such architectures feasible even if it is a **Massive Multiprocessor System** (in short, MMS) connecting more than 10^3 processing elements [SPECIAL82] [POTTER85] [CHRIST84]. Thus, we focus on the PMS architecture level in the dissertation, because systems based on the architecture are flexible and feasible. Table 1.1 summarizes the architecture levels with respect to parallel processing stated above.

Table 1.1 Architecture levels with respect to parallel processing.

Architecture Level	RT	ISP	PMS
Parallelism Level	Sub-Instruction	Instruction	Task or Job
Autonomous Processing Elements	No	No	Yes
Examples	vector processor processor array	data flow machine	systolic array Transputer, GF-11

Many authors have proposed parallel processing systems, based on the PMS architectures, which have a variety of design policies. There are four main factors which a designer should

consider and decide on to design PMS architectures:

(1) Processing element flexibility,

(2) Control strategy,

(3) Connection topology, and

(4) Partitioning and scheduling a job.

Processing element flexibility

As mentioned above, a processing element in the PMS architecture is assumed to be an autonomous module which may be small and simple, or large and complex. The autonomous module, or a computer, can be decided to be flexible for changing its functions: hardwired, firmware, and software. When the hardwired method is selected, we can get a fast and compact module at the expense of the flexibility. In the case of the software method, its condition is the contrary. Systolic array [KUNG82a] and wavefront array [KUNG84] are originally based on hardwired or firmware processing elements. On the other hand, Transputer [MAY84] designed as a language-oriented processor adopts the software method.

Control strategy

A job is distributed and performed on many processing elements communicating with other elements. The communication and synchronization are realized by certain control functions. These functions can be managed by a centralized controller or by each of the individual processing elements. Another way to manage them is a hierarchical control strategy, i.e., the elements are divided into several clusters, each of which is

managed by its centralized controller. The centralized control in GF-11 [BEETEM85], the distributed control in Transputer, and the hierarchical control in Cm* [SWAN77] are typical examples of the three different control strategies mentioned above.

Connection topology

Processing elements are connected with each other by point-to-point links or interconnection networks. The former is called as a static network topology, and the latter as a dynamic network topology. In the dynamic topology, switches in the networks can be dynamically set or not dynamically set. Therefore, connection topologies can be classified into two categories: using point-to-point links and interconnection networks. Topology selecting strategies can be grouped into three categories: static, reconfigurable, and dynamic. Systolic array, wavefront array, Transputer, etc. belong to the point-to-point and static class, and GF-11 belongs to the interconnection network and reconfigurable class. It is now difficult to find a realistic system which can change its topology dynamically.

Partitioning and scheduling a job

From the view point of the software design, program partitioning and scheduling are the most important factors influencing system performance. For high performance, the maximum parallelism with the lowest possible overhead is desired. A job can be partitioned into a various size, from a few instructions to a large task. Parallelism of the job can be

detected by users during algorithm design, by a certain compiler during the compile time, or by a particular software or hardware during the run time. Moreover, scheduling in a multiprocessor is a function that allocates tasks (or processes) to processors, and manages their execution. In the system design, we can consider two kinds of criteria for the scheduling function: static or dynamic, and centralized or decentralized. The static method decides a scheduling before the execution, and the dynamic one determines a scheduling at the run time. A scheduler in a system manages all schedulings in the case of centralized method, and several schedulers manage their local schedulings in the case of decentralized one.

As discussed above, there exist several issues to be considered when a multiprocessor, i.e., a system based on the PMS architecture, is designed. It is difficult to obtain a complete solution of the issues because of their dependency upon applications performed by the system, even if a target is not massive. In particular, the overhead for task and/or resource allocation, and for task or process scheduling increases with increasing number of the processors. The overhead is a crucial problem of massive multiprocessors in both the designing time and the running time. Although a lot of researchers have been investigated the allocation or the scheduling problem [CHOU82] [IRANI82] [STONE77] [MA82] [MA84], there has been no algorithm solving it in feasible time. Since the main objectve of the dissertation is to obtain a methodology which can realize to design MMS's in feasible time, we will try to discuss the methodology under the following assumptions:

(1) ideal (or almost ideal) interprocessor communication hardware and

(2) restricted communication flow.

Those assumptions make it possible to extremely reduce the overhead at the design because the communication cost can be estimated as constant value. Moreover, to evaluate the system performance from the description like a programming language, the strategy of the dissertation is to use a simulator for accurate and quick evaluation. There already exist a few programming language supporting multiprocessors. However, we adopt another description language simplified for performance evaluation, because they are not enough to describe the MMS's. On the other hand, a wide bandwidth interconnection network is also proposed for the first assumption, and makes it possible to realize a target system evaluated by the simulator.

1.1 Summary of Results

The goal of this research is to provide design methodology of MMS's aimed at increasing performance by an order of magnitude. Since the objective applications of the methodology spread a broad range, it assumes that the target systems, i.e., MMS's, are designed for general purpose or a wide range of applications. This dissertation especially focuses on the PMS architecture level which is suitable for VLSI realization of MMS's. A simulator as a design support tool for MMS's and an interconnection network for a realization of MMS's are presented. The dissertation demonstrates the following results through arguments and experiments:

- Experimental results of parallel processing with task level control is obtained from experiments using multiprocessor UNIP with 32 processors. The experiments are done in the fundamental parallel processing in multiprocessors, i.e., pipelined processing and parallel processing. The essential problems of highly parallel processing in multiprocessors are derived from the experiments.

- A model based on the program scheme, and a simulator for performance prediction and evaluation of parallel programs on MMS's are proposed. The simulator implemented on the UNIX operating system is used for the analysis on the experiments. As a result, we found that the simulator is used for the design of MMS's.

- A processor interconnection network based on a new technology of 3-dimensional very-large-scale-integration is proposed to decrease communication time between processors. The proposal of the interconnection technology allows the designer to use the simulator without feedback retry.

1.2 Outline of the Dissertation

In this dissertation, based on experiments using multiprocessor UNIP with 32 processors, a massive multiprocessor simulator for performance evaluation and an interconnection networks for MMS's based on the 3-dimensional integrated circuit technology are discussed.

In Chapter 2, multiprocessor approaches are presented. Basic parallel processing schemes and typical multiprocessor configurations are summarized, and then, a fabrication of experimental multiprocessor system UNIP is described. After several experiments using UNIP are demonstrated, essential and important problems of multiprocessors derived from that experience are summarized.

In Chapter 3, a modeling of MMS programs for performance evaluation using the parallel programming scheme is proposed. The model which is largely intuitive, is applicable to a simulator for the performance evaluation of the MMS's in which the interprocessor communication cost can be measured. The

simulator is implemented on the UNIX system. After a description language of the programming scheme is described, the simulator specification and simulation analysis on the experimental results are also demonstrated.

In Chapter 4, a new type of common memory (in short, 3-D CM) based on a new technology of 3-dimensional integrated circuits is proposed and its fundamental properties are described. A communication module for connecting processors using 3-D CM and processor interconnection networks for MMS's consisting of the modules are demonstrated. A brief analysis of the network performance is also described.

Finally, in Chapter 5, we conclude the dissertation and summarize future problems for a realization of high-performance MMS's.

Chapter 2

Multiprocessor Approach

The discussion and the experiments in this chapter give the basis of the design and realization of MMS's based on the PMS level architectures. Consideration of essential multiprocessor issues leads us to crucial problems of MMS's. A fabrication of a multiprocessor makes difficult points of the MMS's realization clear. The experiments going on in this chapter suggest that a precise estimation of program execution in MMS's is possible before the systems are realized.

First, fundamental properties of multiprocessor systems aimed at high levels of performance are discussed in this chapter. After advantages and disadvantages of the multiprocessor approaches are presented, we consider essential issues, especially, their control schemes, and performance measures of the systems. Next, an experimental multiprocessor UNIP with 32 microprocessors is proposed and experiments using UNIP are demonstrated. Finally, we point out crucial problems of multiprocessors toward MMS's which have high levels of computational performance and the flexibility for their applications.

2.1 Overview

With the advent of VLSI technology which made it possible to manufacture high-performance microprocessors and other circuits at low cost, multiprocessors with many processors have been highlighted in recent years. The following major motivations (or advantages) for building multiprocessor systems:

(1) to increase performance,
(2) to increase reliability, and
(3) to meet distributed application requirements.

On the other hand, several disadvantages of multiprocessor can be pointed out as follows:

(1) The software is complex, difficult to design, expensive to produce, and difficult to test.
(2) Information of the hardware is required for efficient software implementation.
(3) All hardware resources are rarely used at a time.

Gajiski [GAJISKI85] has pointed out the essential issues in multiprocessor systems: hierarchical control of computation, program partitioning, scheduling, synchronization, and memory access. These issues are closely related with multiprocessor architectures. Hierarchical control of computation is discussed in the next section. Program partitioning, scheduling, and synchronization are related to Chapter 3 in this dissertation; however, since we assume that task allocation

should be given for inspections at the design of MMS's, these issues are not discussed in detail. The last issue of memory access is discussed in Chapter 4.

2.2 Parallel Processing Control Scheme

A program executed in a multiprocessor is represented by a control graph [GAJISKI85], in which nodes represent one or more transformations or movements of data, and arcs represent the order in which nodes are executed. The program execution is indicated by the flow of data or control tokens on the graph. From the standpoint of sequencing, there are two models. A serial model of computation corresponds to sequential language execution, where only one token exists in the graph. The token flows without splitting into two or more. In contrast, a parallel model of computation has splitting nodes and merging nodes. A token in the graph flows with splitting and/or merging.

The latter model explicitly represents a parallel processing. Though the former model looks valid only for sequential processing, it can provide parallel processing for several data.

A simple example of the parallel processing in the serial model is pipelined processing, that is, several data flow on the sequentially control graph and parallel processing is performed between processing data. On the other hand, a typical example of the parallel processing in the parallel model is parallel processing in a narrow sense, that is, several processing elements are executed simultaneously. These two processing control schemes are so simple and fundamental that they are suitable for implementation in MMS's consisting of a large number of processing elements.

Parallelism of the pipelined processing and the parallel processing can be implemented at several levels: instruction level, process level, task level, etc. There is a trade-off between parallelism granularity and communication overhead. The designer should decide a level of parallelism in consideration of program algorithm to be solved and a capacity of the interconnection networks.

2.3 Performance Measure

A performance measure of multiprocessors (or, parallel processors) is needed to evaluate a system whether it is sufficient to perform a specified job. In case of single processors, the measure is often represented as throughput (e.g., MIPS: Million Instructions Per Second, FLOPS: FLoating Point Operations Per Second, etc.) which is the capability of performing instructions or operations in one unit time. The throughput makes it possible to compare performance with other systems, and to estimate execution time for a specified job. Since a single processor executes instructions sequentially, the throughput represents accurate performance of the processor. (Note that MIPS or FLOPS does not always reflect the accurate performance when each execution time for an instruction or an operation is not constant but dependent on its variety.)

The throughput, however, is not sufficient to measure performance of multiprocessors because of their parallelism. Moreover, a performance measure which represents multiprocessor architecture as a complex of hardware and software systems is needed, because the throughput only represents the ability of hardware systems.

Several measures of multiprocessors (or, parallel processors) have been proposed as follows:

(1) speed-up ratio [STONE73],

(2) throughput (improved ratio) [ENSLOW74],

(3) parallelism [FENG72],

(4) throughput and parallelism [HOCKNEY81] etc.

The speed-up ratio represents how many times a multiprocessor can execute programs faster than a single processor can. The speed-up ratio is defined as,

$$\text{speed-up ratio} = \frac{\text{execution time in a single processor}}{\text{execution time in a multiprocessor}} .$$

Since the measure directly indicates an effectiveness of multiple processing, it can be intuitively acceptable. However, since it is a relative value, it is difficult to compare a multiprocessor system with another system based on the different processors by the speed-up ratio.

The throughput represents, like definition for a single processor, the capability of performing total instructions or operations on a multiprocessor at unit time. By this measure, we can easily compare performances between systems based on different processors. Moreover, the throughput improved ratio, which is defined as the ratio of the multiprocessor throughput to the single processor throughput, is often referred as a measure of parallel processing effectiveness.

On the other hand, the parallelism, e.g., the number of bits or words executing at a time, is proposed as a measure of multiprocessor performance, especially of architectural factor. The measure is based on the idea that the architectural performance of multiprocessors should be represented by the degree of parallelism but not by their throughput capabilities, which of course depend on their clock rates. The measure may

also refer to the number of words or data performed at a time.

Hockney et al. have proposed two measures, r_∞ and $n_{\frac{1}{2}}$, for the linear approximation, i.e., execution time is approximated by vector length of input data. They imply throughput capabilities and parallelism of multiprocessor systems. It assumes that the system is designed for vectorized data input. When the system solves a problem of input data length n, the system performance r_n is defined as

$$r_n = n/t_n \ ,$$

where t_n is the processing time for the problem.

t_n is obtained from the following linear approximation:

$$t_n = (n+n_{\frac{1}{2}})/r_\infty \ ,$$

where r_∞ is the maximum performance and

$n_{\frac{1}{2}}$ is the length of input data for which the system performance is the half of the maximum performance.

r_∞, which is the limit of r_n as n approaches infinity, is consider to represent the system throughput. $n_{\frac{1}{2}}$ is regarded as a parallelism measure. The linear approximation of the last proposal cannot always apply to any multiprocessor systems based on task level parallelism. However, the measure can represent both the throughput and the parallelism.

Measures (1)-(4) can be selected by the requirements or the design policy of the target system. Throughout the dissertation, we use the following two measures, from which all the measures listed above can be derived:

interval time : t_{int} and

response time : t_{res} .

The interval time t_{int} is the average processing time per datum and the response time t_{res} is the processing time for the first datum. For example, r_∞ and $n_{\frac{1}{2}}$ are derived as follows:

$r_\infty = 1/t_{int}$ and

$n_{\frac{1}{2}} = (r_\infty/r_1)-1 = r_\infty \cdot t_{res}-1$,

where r_1 is the system performance for a datum defined above. The speed-up ratio is also derived as follows:

speed-up ratio = $t_{ref}/(t_{int} \cdot n + t_{res})$,

where n is the number of input data and

t_{ref} is the execution time for the same job in a reference single processor.

It could definitely be said that the execution time in a reference single processor is required. Other measures described above are also derived in a similar way.

2.4 Experimental Multiprocessor UNIP

We have developed a multiprocessor UNIP with 32 processors for experimental applications [AE82]. UNIP served as a pilot machine for acquiring fundamental data of execution and communication time in task level parallelism. In this section, the hardware and system software of multiprocessor UNIP are presented.

2.4.1 Design Policy

Multiprocessor UNIP with 32 microprocessors has been designed as an experimental machine for the purpose of gaining fundamental data, e.g., execution and communication time for parallel processing, and making underlying problems visible toward high-performance MMS's. We fabricated a realistic machine with more than ten processors even if an each processor is very small, and facilitated execution of pipelined processing and/or parallel processing. The design policy of the multiprocessor is listed as follows:

(1) Simple hardware,
(2) Attached processor,
(3) Homogeneous-processor organization,
(4) Single bus structure, and
(5) Parallel and/or pipelined processing configuration.

Simple hardware is required for a guarantee of execution

stability when several asynchronous processors run individually. Each processor and its communication mechanism with other processors are designed as simply as possible. The simplicity is also suitable for the VLSI realization.

Attached processor, which needs a host computer to complete a job, plays a role of a back-end processor of the host. Since the host handles its peripheral devices, e.g., disk storage, and user interface facility, the multiprocessor can dedicate itself to its own tasks. UNIP communicates with a host computer through a full-duplex parallel port.

Homogeneous-processor organization and single bus structure are available to simplify the hardware and to repair or exchange a faulty processor. They also lead to an ease of fabrication and the system extensibility.

UNIP has two kinds of communication mechanism between processors, i.e., bus transfer mechanism connecting all processors and parallel port connecting adjacent processors. The mechanism is used for pipelined and/or parallel processing. In the pipelined processing, main data stream uses the parallel port. In the parallel processing, data are distributed and acquired through the bus.

2.4.2 Hardware Overview

UNIP contains 32-working processors called slaves, and one supervisor called master. All processors are connected with a shared bus where the master processor performs memory access to slaves. In addition, two adjacent processors are combined to

each other with a parallel port. Fig. 2.1 shows the configuration of UNIP. Each processor of master or slave is assembled on a single board, and has Zilog Z-80A CPU, 16 KBytes random access memory, input/output interfaces (a pair of bit-parallel ports for slaves and two pairs for master), bus controller, etc. All slaves are completely homogeneous except for hardware switch to identify slaves.

As one of the characteristics of UNIP, each processor has a communication port to both of adjacent processors. Since the port can be combined with any device or processor as well as adjacent processor, the connection configuration of processors is not fixed. In reality, several multiprocessors, i.e., subset of UNIP, with a few slaves which have different connection pattern are also produced in our laboratory.

There exist many multiprocessor systems with single bus structure and a variety of access methods through the bus. To avoid the occurrence of conflict on the bus, several arbitration methods are proposed. A typical solution is an additional hardware arbiter, e.g., a ring arbiter, race arbiter, etc. When two or more processors is required to use a single bus at the same time, the bus arbiter accepts only one request and permits its accepted processor to access the bus. UNIP, however, has no additional hardware for bus arbitration because of its hardware simplicity. The access right through the bus is given only to the master processor of UNIP for the reason that the master dedicates itself to all communications through the bus. Therefore, the master plays a role of an interprocessor,

or a communication processor, transferring data between host and slaves, and between slaves. Fig 2.2 shows memory map of relation between the master and slaves. From the master processor, all slaves' memory modules look like one module by interleaving. The master has two memory access modes: **individual mode** and **broadcasting mode.** In the former mode, the master can individually read and write a slave's memory. For the selection of a specified slave memory, the address bus width is extended to 24 bits, where the extended 8 bits can select a slave from the maximum 256 slaves. In the access mode, the slave to which the master accesses halts during a period of real memory access (about 500 nsec per access). In the latter access mode, the master can write data in interleaved memory of all slaves simultaneously, where the extended address is ignored and all slaves halt. The memory access is restricted to only write operation in this case.

Interruption is required for task level synchronization between the master and slaves. Two directions of the interruption are considered: the master interrupts slaves for an initiation or termination of slave tasks, and slaves interrupt the master for a communication request with another slave or with the master. Since UNIP has a hardware interruption mechanism only from the master to slaves because of its simplicity, the master must observe status of all slaves instead of the interruption. The interruption is provided with two modes such as the memory access. One is individual interruption and the other is broadcasting. The former interrupts the slave selected by extended address, and the latter causes the

interruption in all slaves at a time. Since NMI (Non Maskable Interrupt) of Z-80A is used for the interruption, slaves can never avoid it.

2.4.3 System Software

UNIP provides basic communication and processor control facilities by system software including software development tools. The software is divided into two functional parts: runtime support and development assistance. These functions are as follows:

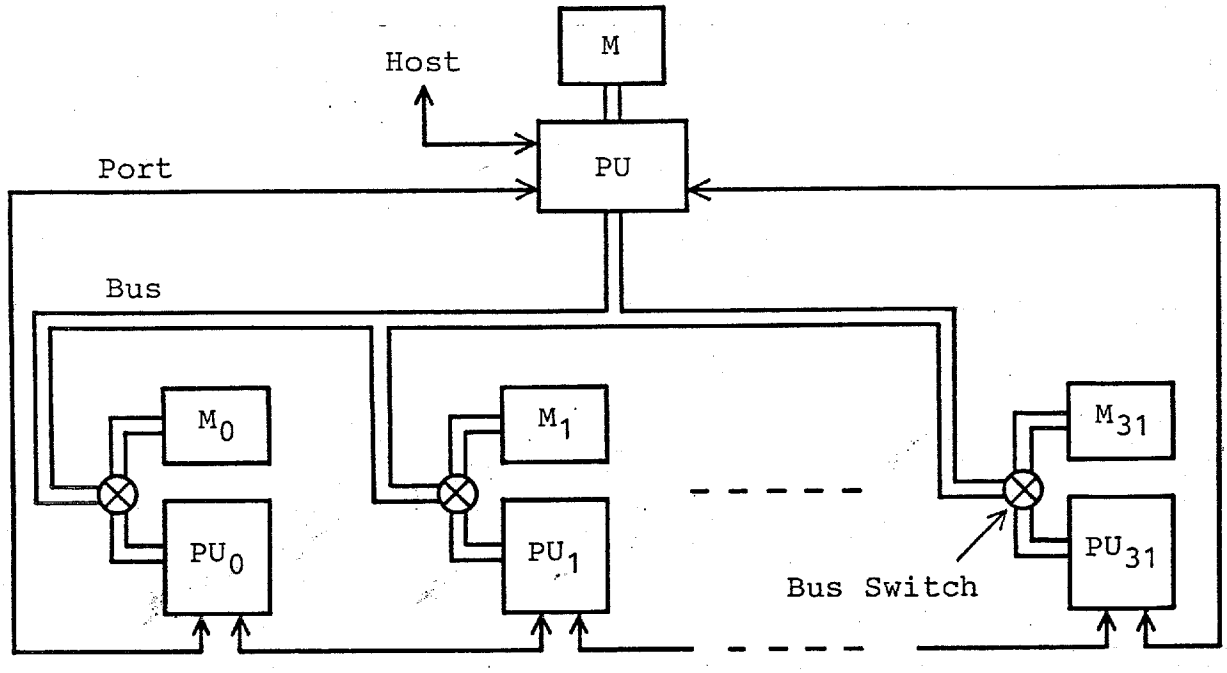
Runtime support

- data transfer through the bus
- data transfer using the parallel port
- slave control functions, e.g., reset, interruption, initiation of slave program, etc. (master only)
- communication with a host computer (master only)

Development assistance

- memory read and write in an arbitrary processor from the host
- initiation of programs in an arbitrary processor from the host
- single step execution at machine language level for debugging
- development support of multiple languages, i.e., assembly language, C language and Forth language

Since the runtime support part must be installed in small memory space of the processors, the functions are minimized for compaction within 2K Bytes of the code size.



PU : Master Processor M : Master Memory
PU_i : Slave Processor M_i : Slave Memory

($0 \leq i \leq 31$)

Fig. 2.1 Configuration of multiprocessor UNIP.

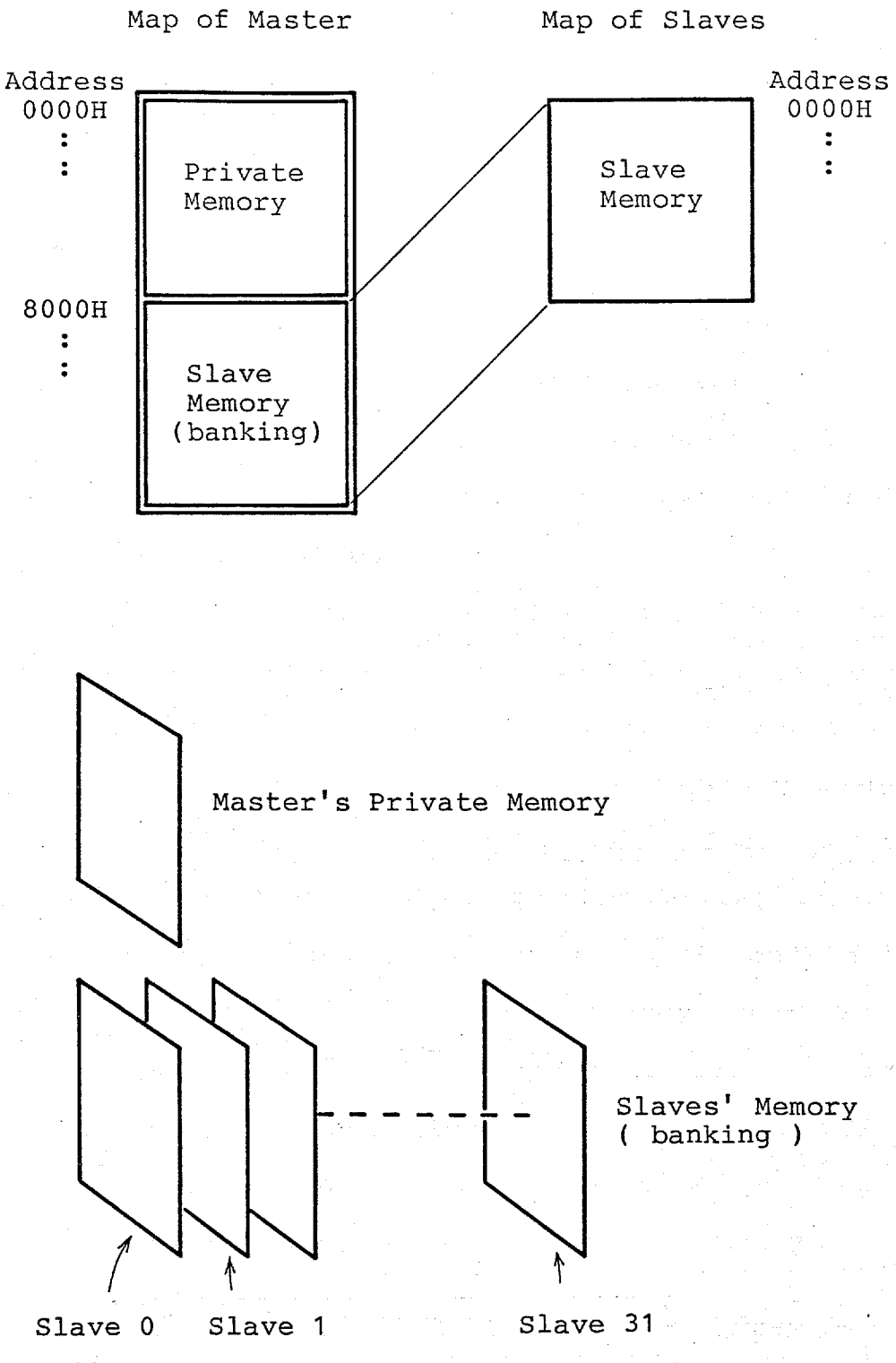


Fig. 2.2 Illustration of memory map in UNIP.

2.5 Case Studies

2.5.1 Sorting and Searching

In database management systems, the execution time for data sorting and searching occupies the most part of the whole execution time. An interactive database system requires its processor to shorten the processing time dramatically, when the large amount of data is treated. We have implemented a sorting and searching modules on UNIP and obtained their experimental results [AIBARA85].

The sorting and searching algorithms that we use have been originally proposed by Tanaka et al. [TANAKA80] called **pipelined heap sorting** and **pipelined searching**, respectively. The original algorithms have the same data structure, as shown in Fig. 2.3 , i.e., binary tree structure. Since a processor corresponds to a level of the data structure in implementation on a multiprocessor, the level i processor must contain 2^i data. However, it is difficult for the processor which keeps only 8-KByte memory space (after that, memory has increased up to 16 KBytes) to implement the algorithms. As a result, we modified the algorithms for the data structure, as shown in Fig. 2.4 . The structure saturates increase in data at level s in order to limit at most 2^s data stored in a processor. The modified algorithms require the number of processors, p:

$$p = \lceil (n+1)/k \rceil - 1 + \log_2 k \quad (n \geq 2k) \quad \text{or}$$
$$\lceil \log_2 (n+1) \rceil \quad (n < 2k),$$

where n is the number of processing data (sorted data in the sorting, stored and searched data in the searching), and

k is the maximum number of stored data in a processor (i.e., $k=2^s$).

The sorting and searching algorithms consist of two phases, respectively. In the sorting, one is **input phase**, when data from the host come into the multiprocessor comparing the data with another data and they are stored in each processors, and the other is **output phase**, when stored data comparing with another data go out to the host. In the searching, one is **storing phase**, when data already sorted come into the multiprocessor and they are stored in the specified sequence, and the other is **searching phase**, when data to be searched come from the host and they are compared with the stored data. The identifier of the stored data is taken out in the search phase, when the datum of the same value is found. Each phase is executed in pipelined processing. Fig. 2.5 shows a time chart of the input phase in the sorting and the both phases in the searching, and Fig. 2.6 shows a time chart of the output phase in the sorting, where Figs. 2.5 and 2.6 indicate the case that each processor cannot execute both internal processing (e.g., comparing) and input/output handling concurrently.

Figs. 2.7 and 2.8, and Tables 2.1 and 2.2 show experimental results of total execution time for sorting and searching. Moreover, Table 2.3 shows the average rate of data transfer between UNIP and the host computer in sorting and searching.

Table 2.4 shows each part of sorting and searching time represented by percentages. The specification of the implementation is listed as follows:

- One datum consists of
  key : 32 bits (4 bytes) and
  identifier : 16 bits (2 bytes).
- The maximum number of stored data in the system is 11775.
- The maximum number of stored data in a processor is 512.
- The maximum number of working processors is 31.
- The programs are written by assembly language.

2.5.2 Two-Dimensional Bit-Pattern Matching

An experimental 2-dimensional bit-pattern matching system has implemented in UNIP. The system is divided into two parts: pattern analysis for data compression (GA), and data searching (REC). GA and REC can be executed in parallel processing and in pipelined processing, respectively, when several bit-pattern data to be matched come from the host. UNIP slave processors are divided into two parts corresponding to GA and REC. The GA part is used as parallel processor and the REC part is used as pipelined processor. Fig. 2.9 shows the experimental results of execution time, where the number of GA part processors added to REC processors is constant (that is 29). Table 2.5 shows each part of 2-dimensional bit-pattern matching execution time, which has been estimated from the program list. The processing system is divided into several

parts as shown in the Table:

Host -> Master	: The master receives bit-pattern data to be matched from the host.
Master -> GA	: The master sends the data to GA processor waiting for the input.
GA	: A GA processor analyzes a bit-pattern and compress it into a string data.
GA -> Master	: The master acquires results from GA's which have completed the processing.
Bank1	: A slave receives the string data from master and sends it to a REC processor.
REC	: A REC compares the input string data with the reference data stored in the processor. The REC finding a matched data sends its identifier as output.
Bank2	: A slave sends results of REC's to the master.
Master -> Host	: The master sends the results obtained from the Bank2 to the host.

Note that, the master, the Bank1 and the Bank2 play the role of data buffering.

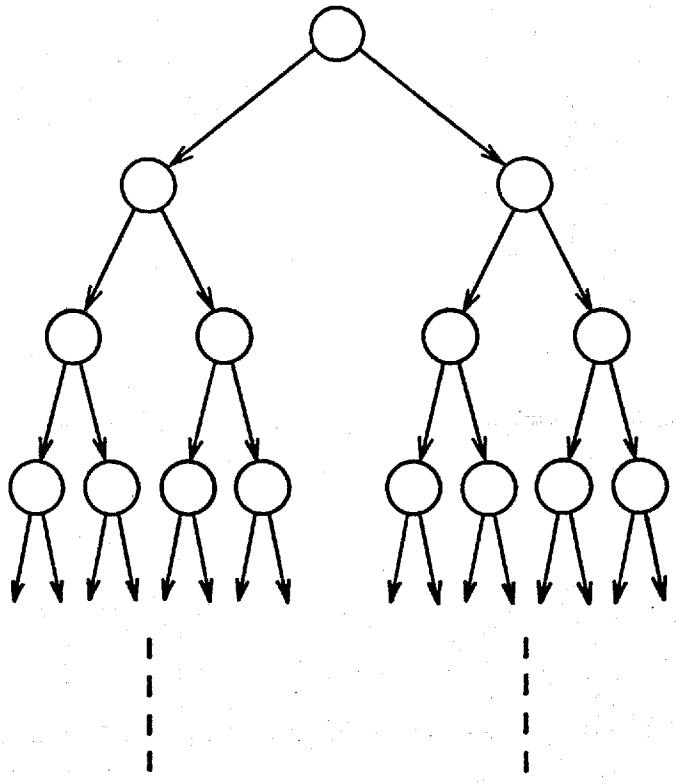


Fig. 2.3 Data structure for sorting and searching.

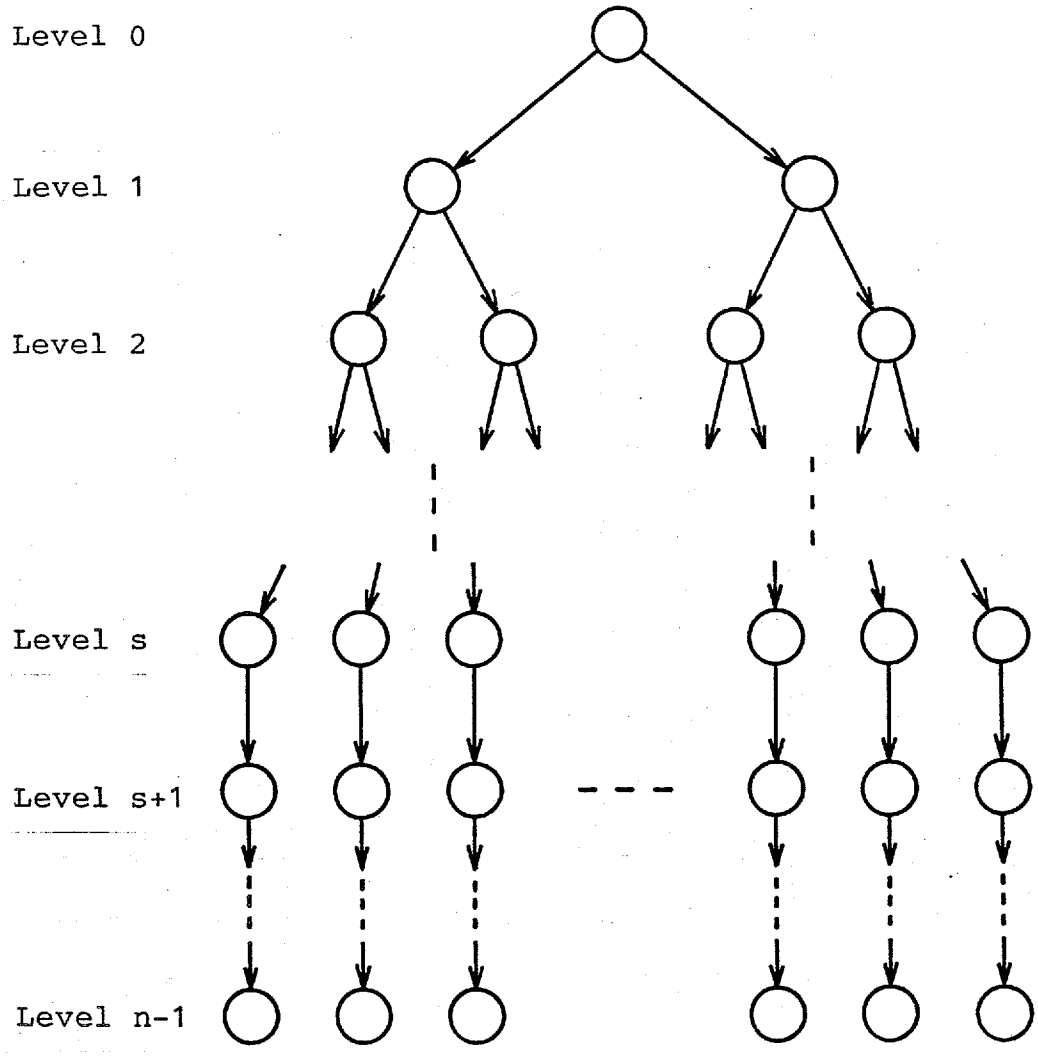
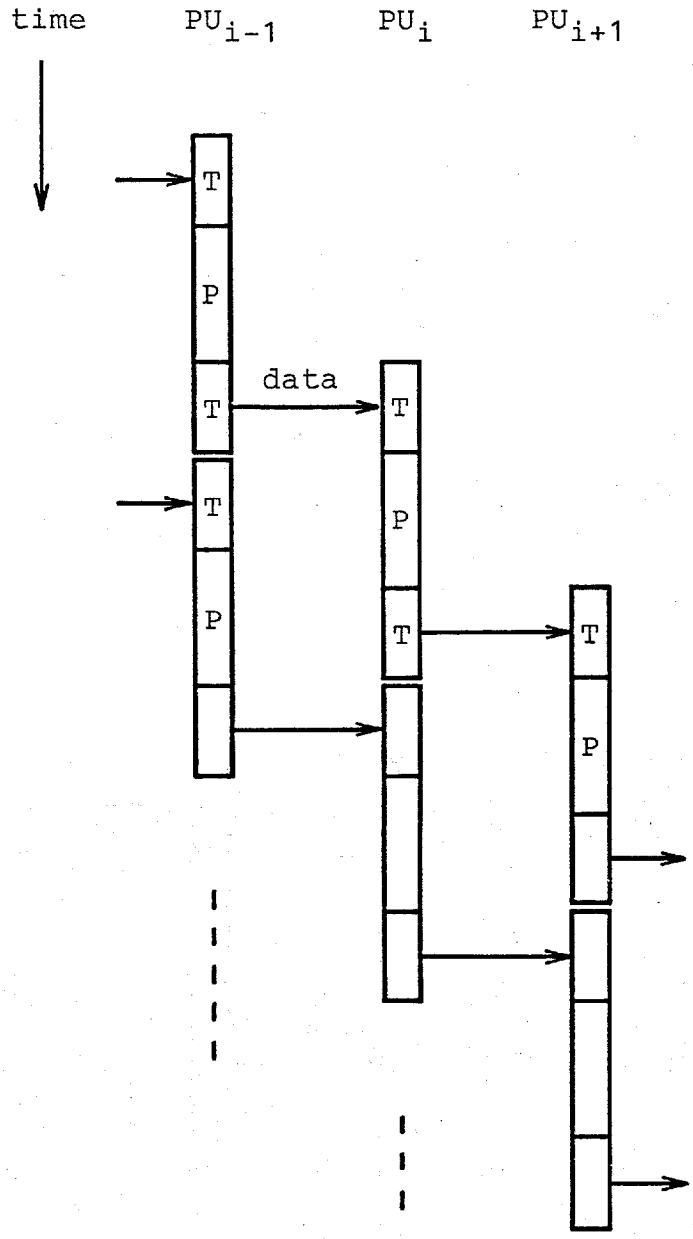
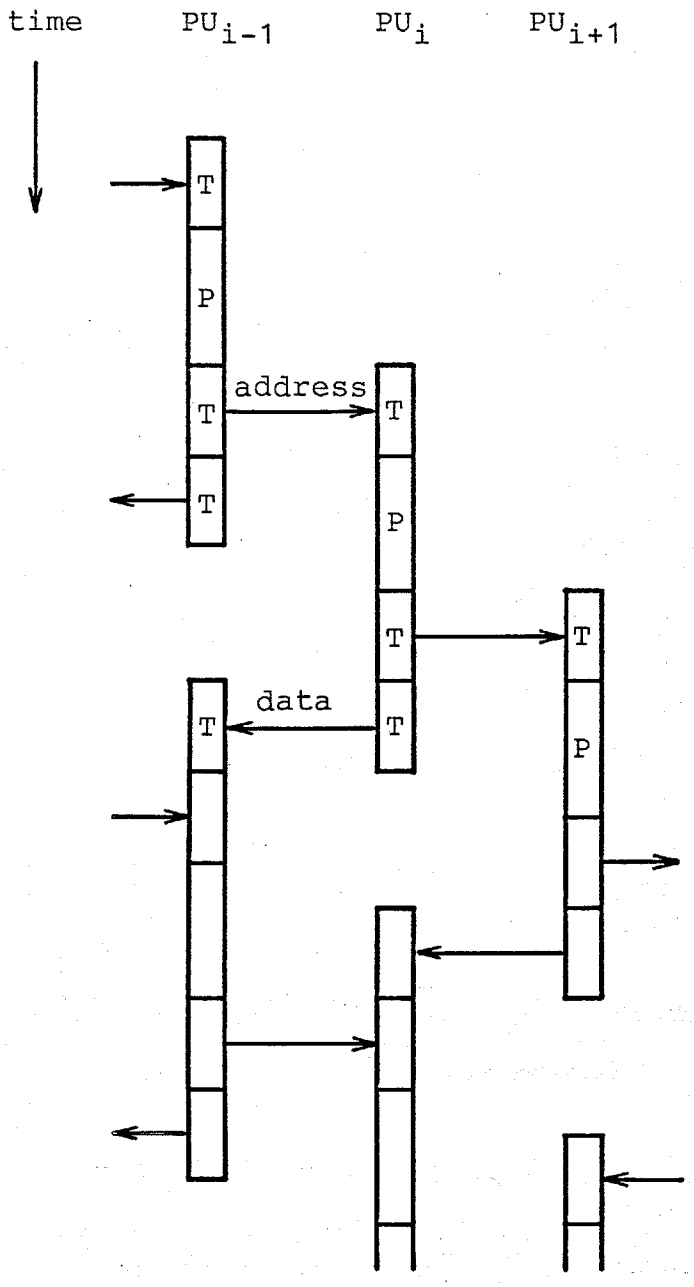


Fig. 2.4 Modified data structure for sorting and searching.



P : Processing Time T : Transfer Time

Fig. 2.5 Time chart of pipelined processing (I).



P : Processing Time T : Transfer Time

Fig. 2.6 Time chart of pipelined processing (II).

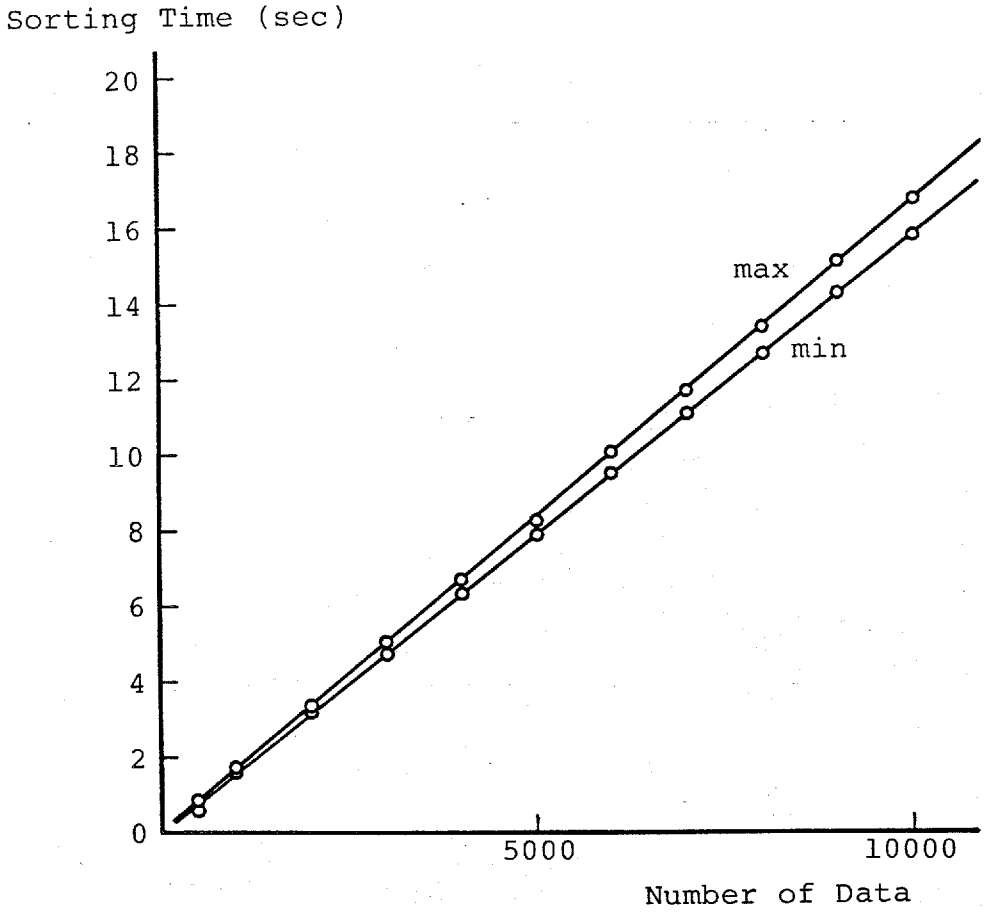


Fig. 2.7 Sorting execution time.

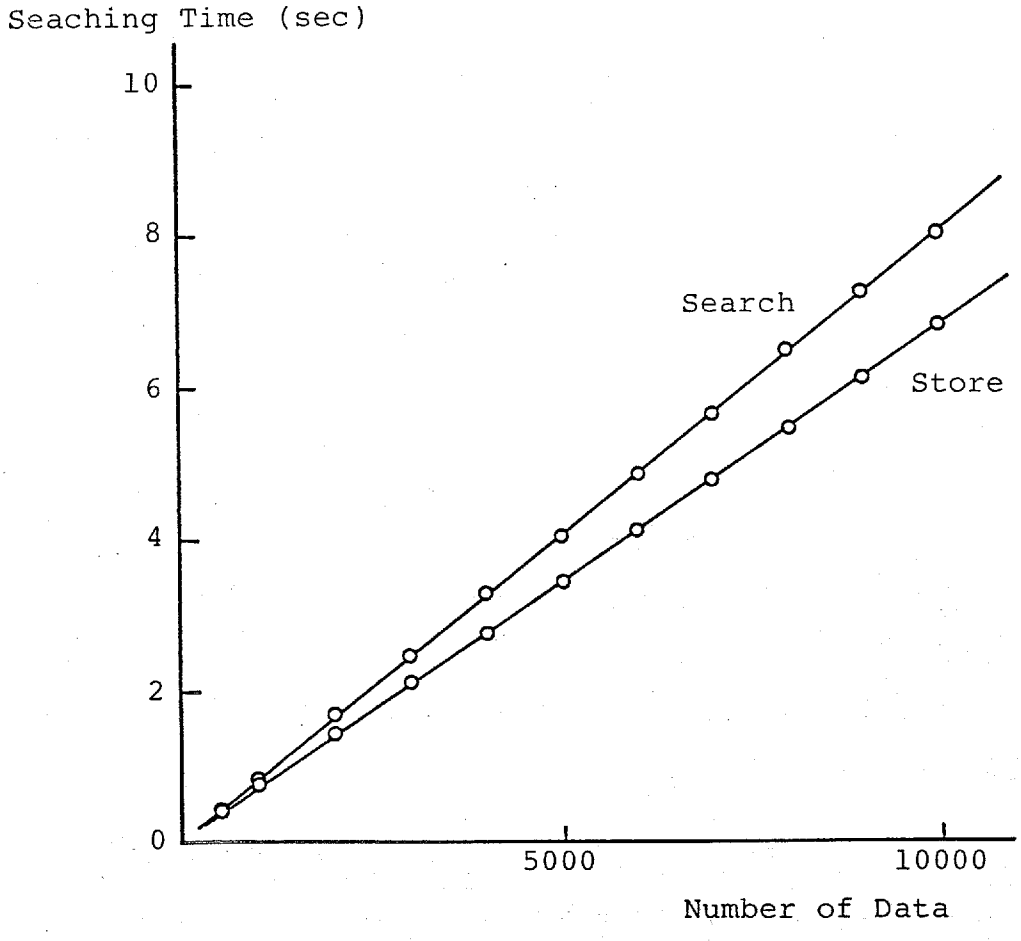


Fig. 2.8 Searching execution time.

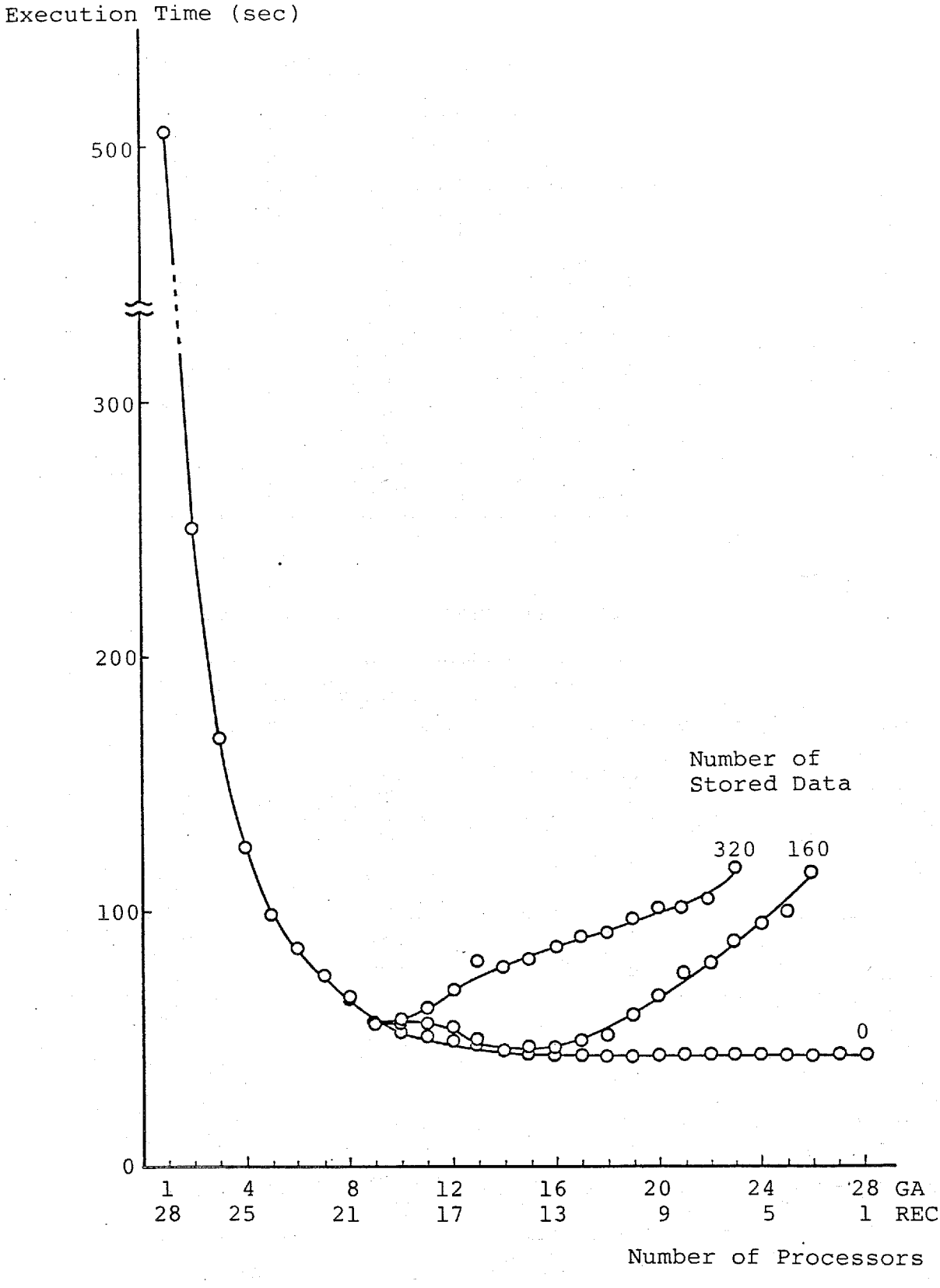


Fig. 2.9 Two-dimensional bit-pattern matching execution time.

Table 2.1 Sorting execution time.

Number of Data	Sorting Time (sec)	
	min.	max.
100	0.19	0.20
500	0.57	0.86
1000	1.61	1.69
2000	3.18	3.36
3000	4.75	5.03
4000	6.34	6.71
5000	7.91	8.25
6000	9.49	10.05
7000	11.07	11.72
8000	12.65	13.40
9000	14.24	15.10
10000	15.82	16.78

Table 2.2 Searching execution time.

Number of Data	Searching Time (sec)	
	Store	Search
100	0.11	0.08
500	0.37	0.40
1000	0.71	0.79
2000	1.37	1.57
3000	2.04	2.36
4000	2.70	3.14
5000	3.37	3.92
6000	4.03	4.71
7000	4.70	5.49
8000	5.36	6.28
9000	6.03	7.06
10000	6.69	7.84

Table 2.3 Average data transfer speed in sorting and searching.

Phase		Ave. Data Transfer (Kbyte/sec)
Sort	Data Input	8.9
	Data Output	6.3
Search	Data Store	9.0
	Data Search	7.7

Table 2.4 Each part of sorting and searching time.

Phase		Essential Processing (%)	Data Transfer (%)	Other Processing (%)
Sort	Data Input	13.9	62.5	23.6
	Data Output	10.5	84.4	5.1
Search	Data Search	12.5	86.2	1.3

Table 2.5 Each part of 2-dimensional bit-pattern matching execution time.

Processing Part	Time (msec/data)
Host -> Master	21.1
Master -> GA	2.97
Proc. GA	489
GA -> Master	0.72
Master -> Bank1	0.72
Bank1 -> REC	3.20
Proc. REC	300+827/m
REC -> REC	3.20
REC -> Bank2	3.20
Bank2 -> Master	0.09
Master -> Host	0.64

Note that, m is the number of REC processors.

2.6 Summary

We have obtained fundamental data with respect to the parallel and pipelined processing and given essential issues of multiprocessing through the experiments using UNIP. Major problems of designing and realizing such a multiprocessor as an MMS with more than 10^3 processors are presented as follows:

(1) Does the algorithm contain sufficient parallelism for MMS's?
(2) Is it possible to estimate the total execution time including the communication time?
(3) Is it possible to expand the communication bandwidth of interconnection networks?

Though solution of the optimum parallelism of algorithm is not a goal of this dissertation, we believe that there exists sufficient parallelism in the application fields of database management, pattern recognition, etc., as mentioned in the previous section.

Approaches to a solution of the problem (2) can be classified into two categories:

(a) to obtain an (approximately) optimum allocation of tasks for a given configuration of multiprocessor and
(b) to estimate or evaluate the performance for a given task allocation and a given multiprocessor configuration.

For the approach (a), the solution using the minimum cut algorithm in graph theory [STONE77] and the heuristic algorithm [MA82] have been proposed. These algorithms, however, can

solve the problem restricted to a few processors, and cannot apply to a large number of processors such as MMS's. For the approach (b), the queuing theory and simulation methods such as the Monte Carlo simulation can apply to the problem. In this approach, the simulation is repeated until the evaluation results satisfy the designated requirements. This dissertation adopts the simulation for performance evaluation. The simulation model and a simulator realized on a UNIX system will be described in the next chapter.

A solution of problem (3) is the interconnection network with sufficiently broad bandwidth. However, there exists a trade-off between network cost (i.e., hardware complexity) and network performance (i.e., throughput and delay). A multi-port memory with multiple-read single-write based on a new device technology of 3-dimensional integrated circuits (i.e., multi-layered VLSI) is proposed in Chapter 4. Moreover, MMS's interconnection networks using the memory are considered.

Chapter 3

Performance Evaluation

This chapter mainly discusses a simulator as a development support tool for the performance evaluation of Massive Multi-processor Systems (MMS's) aimed at high levels of performance. To clearly represent MMS's functioning and behaviors, the Parallel Flow Graph (PFG) which consists of nodes representing small tasks and arcs indicating inter-task communications is introduced for their design, performance evaluation and simulation. For the performance evaluation of MMS's, a simulator which can simulate more than 10^3 nodes has been implemented regarding PFG as a simulation model.

3.1 Modeling

When studying the performance of asynchronous concurrent systems, various techniques are available to model system's behavior and workloads [SPECIAL83] [SPECIAL84] [HIDELBERGER82] [DUBOIS84]. A model for predicting the total execution time of a logical scheme must satisfy at least the following criteria:

(1) Describes a logical scheme of MMS and its data flow.
(2) Provides the capability to evaluate a large system.
(3) Provides a complete, unambiguous, and machine-processable form.

Considering the first criterion, a graph model is preferable to others, Queuing Model, Markovian Model etc., and have been developed by many authors. Therefore, we introduce a graph model, which is similar to Flow Graph [WESSELKAMPER82] [GAJISKI82] [KODRES78] [OLDEHOEFT83] [MEKLEY80] [JAJODIA83] and Timed Petri Net [PETERSON81] [RAMAMOORTHY80] as a candidate which satisfies the above criterion.

In order to model and to evaluate clearly the data flow on the logical scheme of MMS, we begin with Parallel Flow Graph (in short, PFG), i.e., an expansion of Flow Graph. Other graphical schemes, process flow graphs, data flow graphs and data dependence graph, are alike except that PFG is provided for the performance analysis of logical schemes. PFG is a finite digraph which has nodes, directed edges, and dot markings for representation of processes, communication links and communication media such as messages, respectively. We assume that those processes logically communicate with each other only through channels, and that the execution itself is sequentially performed within a process.

A. Node Primitives

According to input/output behavior, we introduce five node primitives: process nodes, fork nodes, join nodes, select nodes and merge nodes [AIBARA86].

Process Node (PN)

A process node has a single incoming edge for a single input port and a single outgoing edge for a single output port. It

receives one message as an input, and sends one message to the output port. (See Fig. 3.1(a).) Throughout this chapter, we denote that a **message** is the media of communications.

Fork Node (FN)

A fork node has a single incoming edge for a single input port and n outgoing edges for n output ports. As is shown in Fig. 3.1(b), it receives a message as an input, and sends n messages to the n output ports.

Join Node (JN)

A join node has n incoming edges for n input ports and a single outgoing edge for a single output port. As is shown in Fig. 3.1(c), it receives n messages as n inputs from each direction, and sends a message to the output port.

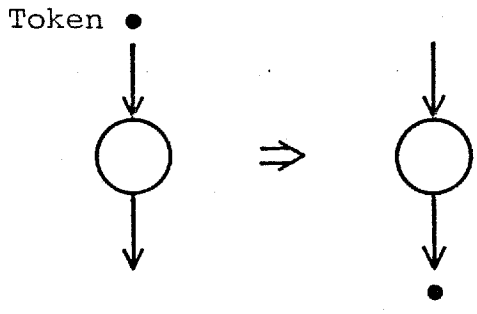
Select Node (SN)

A select node has a single incoming edge for a single input port and n outgoing edges for a single output port. As is shown in Fig. 3.1(d), it receives a message as an input, and sends a message as an output to an appropriate direction selected from the n directions.

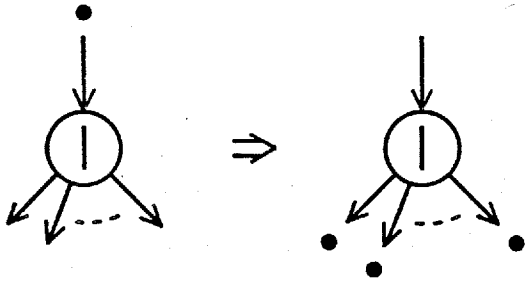
Merge Node (MN)

A merge node has n incoming edges for a single input port and a single outgoing edge for a single output port. As is shown in Fig. 3.1(e), it receives a message as an input from an appropriate direction selected from the n directions, and sends a message to the output port.

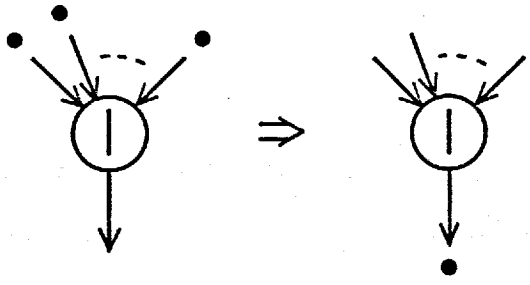
n means the number of incoming or outgoing incident edges. If n is equal to 1, a node primitive can be regarded as a process node.



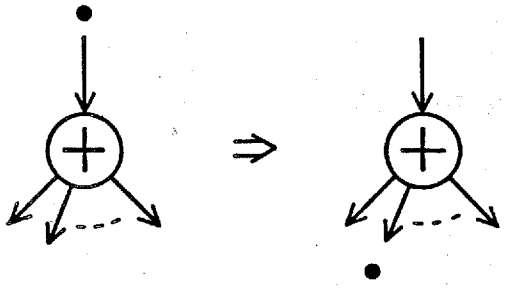
(a) Process Node



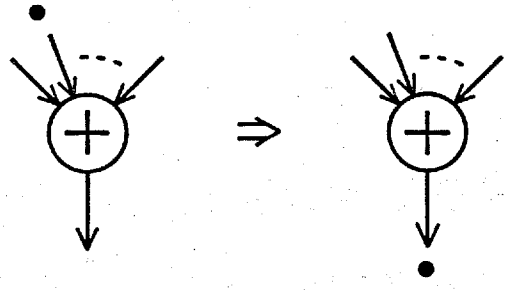
(b) Fork Node



(c) Join Node



(d) Select Node



(e) Merge Node

Fig. 3.1 Node primitives.

B. Node State and Time Factors

Each node primitive has a unique state at a time. There are six states as follows:

1. input ready,
2. input,
3. process ready,
4. process,
5. output ready, and
6. output.

We assume the state transition to be shown as in Fig. 3.2. When a message flows into a certain node, the node changes its state with time delay. The time delay of a node is defined as the time elapsed from when a node state is input ready to when it becomes input ready again. The time delay can be divided into three factors: the processing time P, the communication time C, and the waiting time W. Hence,

Total Time Delay = P+C+W.

Considering the meaning of input and output behavior of a node, a communication time C can be divided into the input time Cin and the output time Cout:

C = Cin+Cout.

Thus, a circular state transition requires the following time values.

Phase 1: from input ready to process ready: Cin

Phase 2: from process ready to output ready: P

Phase 3: from output ready to input ready: Cout+W

The value W, i.e., a waiting time, is the time that the node spends to wait for the successor's input ready state. The behavior "waiting" is caused by following synchronizations. (Note that these synchronizations are mentioned only for the process node primitive. In another case, some extensions are needed.)

Synchronization 1:

A node whose state is input ready changes its state to input, if and only if its predecessor's state is output ready.

Synchronization 2:

A node whose state is output ready changes its state to output, if and only if its successor's state is input ready.

In order to represent the time domain behavior, we label the processing time on a node and the communication time on a directed edge (See Fig. 3.3), because we can obtain those times as static time factors.

Considering the dynamic time-domain behavior, the waiting time must be included. However, we cannot clarify it except the restricted case [AE84a], since the waiting time is a dynamic factor which is affected by other nodes (which are adjacent to others).

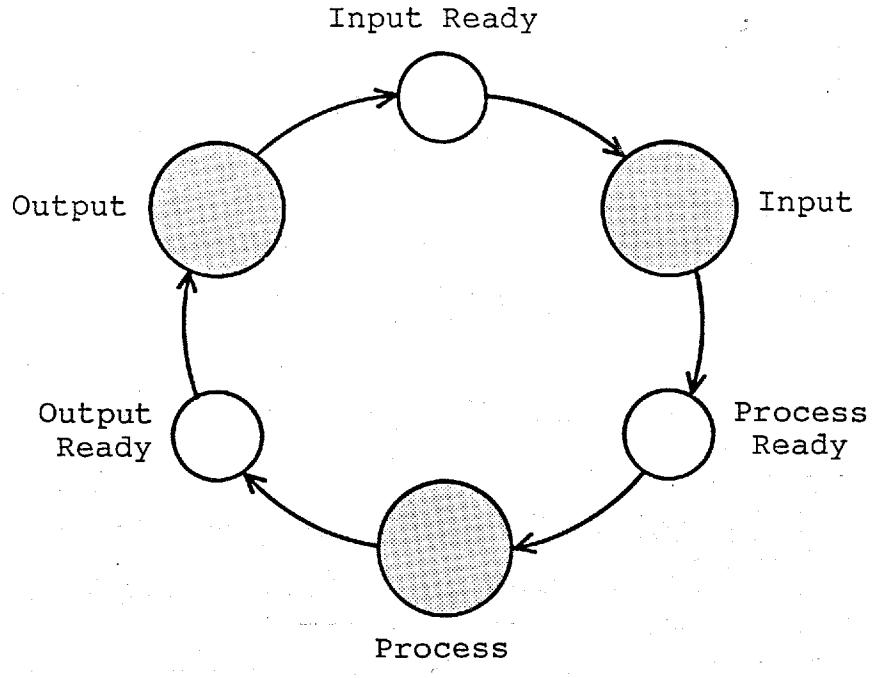


Fig. 3.2 Node state transition.

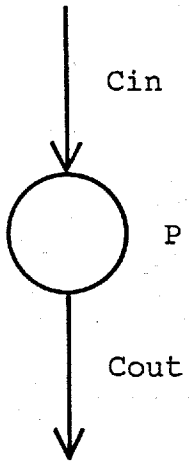


Fig. 3.3 Labeling on process node.

3.2 Applicable Structure and Description

3.2.1 Representation of Graph Constructions

In this section we present basic properties of PFG which can be denoted by the symbolic form [AE84a]. In contrast to the reduction rules in the definition of PFG, there exist the construction rules, which are called "sequential construction", "parallel construction" and "selective construction", and which perform precisely reverse operations to reduction rules. These constructions are denoted by expressions with a substitution form as follows. To simplify the notation, a module is denoted by a single symbol which is unique within a PFG as an identifier.

1. Sequential Construction:

M:=a•b denotes the sequential construction of module "M" into module "a" followed by module "b" as shown in Fig. 3.4(a). Module "M" is called a "sequential module".

2. parallel construction:

M:=a|b denotes the parallel construction of module "M" into module "a" and "b" connected by the fork/join nodes as shown in Fig. 3.4(b). Module "M" is called a "parallel module".

3. selective construction:

M:=a+b denotes selective construction of module "M" into module "a" and "b" connected by the select/merge nodes as shown in Fig. 3.4(c). Module "M" is called a "selective module".

In the case of Fig. 3.5, the construction of the PFG begins with module "A" at the highest level. Next we apply the above constructions to obtain the scheme represented by Fig. 3.5 as follows:

$$A:=B\cdot C$$

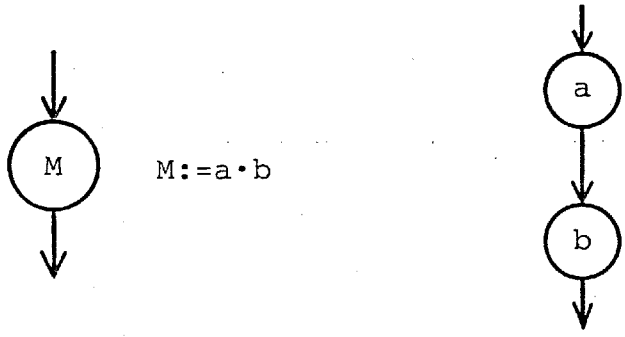
$$B:=D+E$$

$$D:=F|G\ .$$

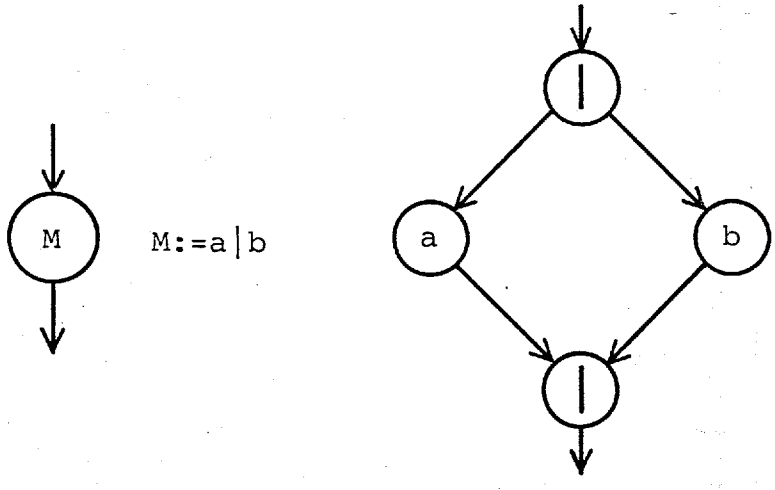
After the above substitutions of expressions, we obtain the symbolic form

$$A=((F|G)+E)\cdot C$$

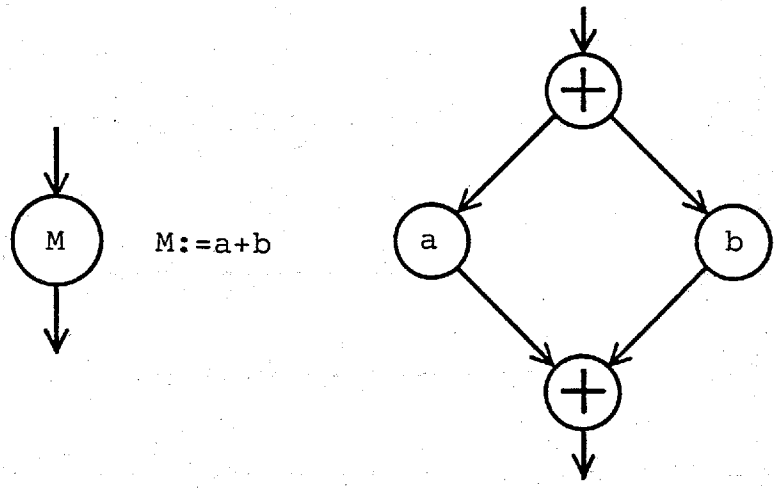
which represents the given logical scheme. Such an expression is equivalent to PFG, and is also another representation of the logical scheme. This is an important property because it provides machine processable forms for a design support system. The following section will show you that it is very usable for the simulator implementation.



(a) Sequential construction



(b) Parallel construction



(c) Selective construction

Fig. 3.4 Three construction rules.

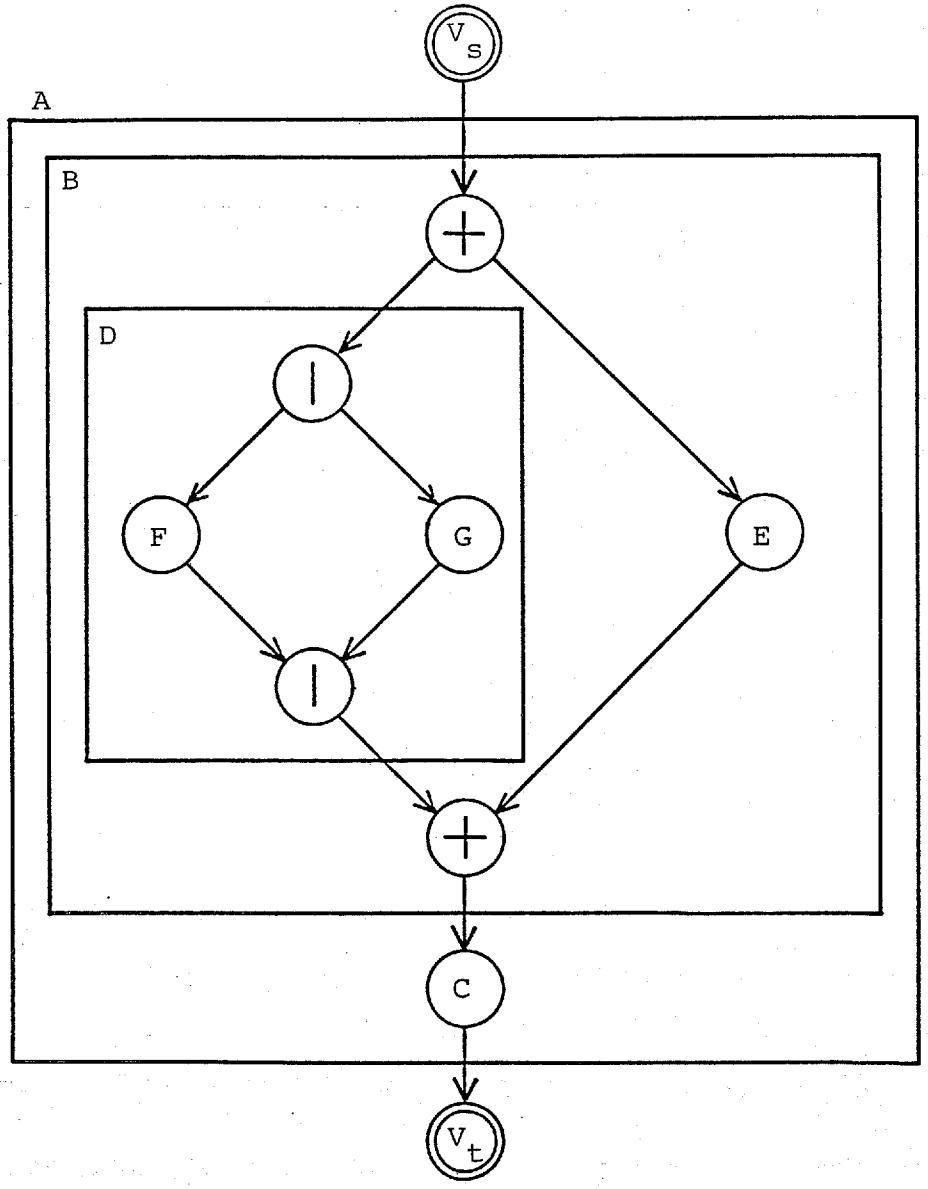


Fig. 3.5 Example of Parallel Flow Graph.

3.2.2 Behavior of Modules

The message flow on the three kinds of modules at any level are specified as follows:

Sequential Module

Fig. 3.6(a) essentially represents a sequence of two process nodes, i.e., a sequential module where the input message is given in turn. An input message flows into the node A and the node B sequentially with some time delayed. Fig. 3.6(b) shows the time chart of Fig. 3.6(a). In the condition that there are "no wait" caused by the synchronization, the time delay is equal to C1+P1+C2+P2+C3. In a general case, however, the waiting time W must be included so that total time delay is represented as follows:

Total Time Delay = C1+P1+C2+P2+C3+W .

That waiting time within a process node is caused by its successor who is not input ready when he is output ready. Such waiting time can be hardly clarified deterministically because it is hard to clarify when the successor will be input ready while its successor's status is affected by the successor's successor recursively. Therefore, we can label the static time factor, the processing time and the communication time, on node primitives while we cannot label the delay time including the waiting time on modules in general.

Parallel Module

Fig. 3.7 illustrates a parallel module which contains two modules A and B represented by squares, the fork node and the join node. An input message i is copied into two messages with same identifier i at the fork node. Two messages flow along the path F-A-J and the path F-B-J, respectively, and at the join node, those are reduced into one. As is easily understood, the total delay time of an input message in this module is obtained as the maximum delay time of two paths. Considering the join node's behavior, it is needed that two modules, A and B, must have the FIFO queue for the input/output, in order to join two messages with same identifier. All node primitives, the sequential module and the parallel module have essentially the FIFO queue. In the case of selective module, more detailed description is needed.

Note that, from now on, we use the rectangle or the square for the representation of modules in figures.

Selective Module

Fig. 3.8 represents a selective module which contains two modules A and B represented by squares, the select node and the merge node. The path of an input message i's flow, is switched by the select node according to conditions of the message's contents or the status of module A and B. A selective module indicates a non-deterministic behavior in the meaning of that the direction of a message flow is not decidable. Therefore, the total delay time of an input message in this module should be obtained as the best case or the worst case. Meanwhile, in order to keep FIFO manner, we must describe the merge node's behavior in detail. In a selective module, it

easily occurs that two consecutive message i and message i+1 arrive at a merge node in the reverse order, for the preceding message can take more delay time than the following message depending on the selected path. In order to avoid this case, we assume that the merge node must select its input so that message i+1 follows message i for any i ($1 \leqq i \leqq n$).

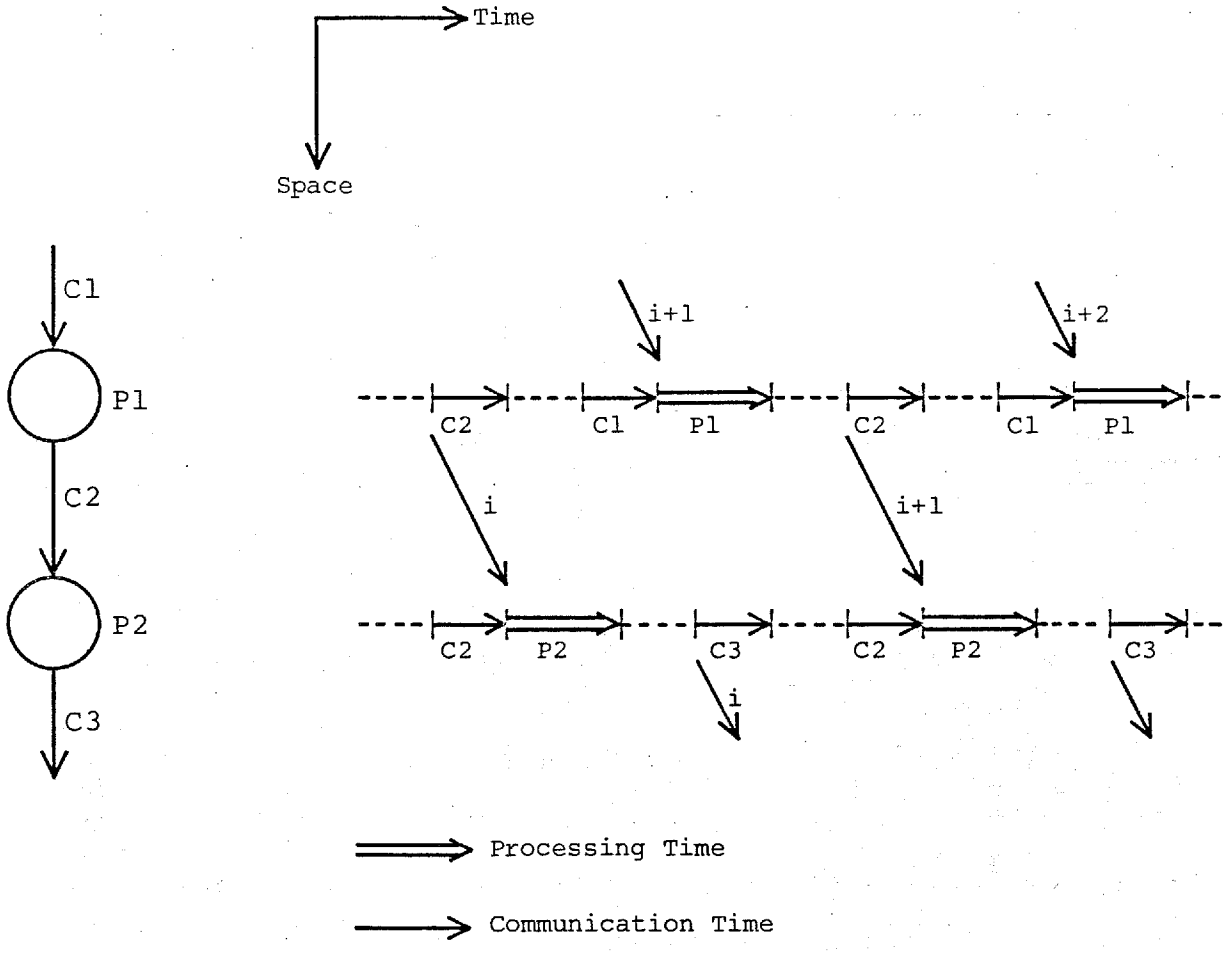


Fig. 3.6 Sequential model and its time-domain behavior.

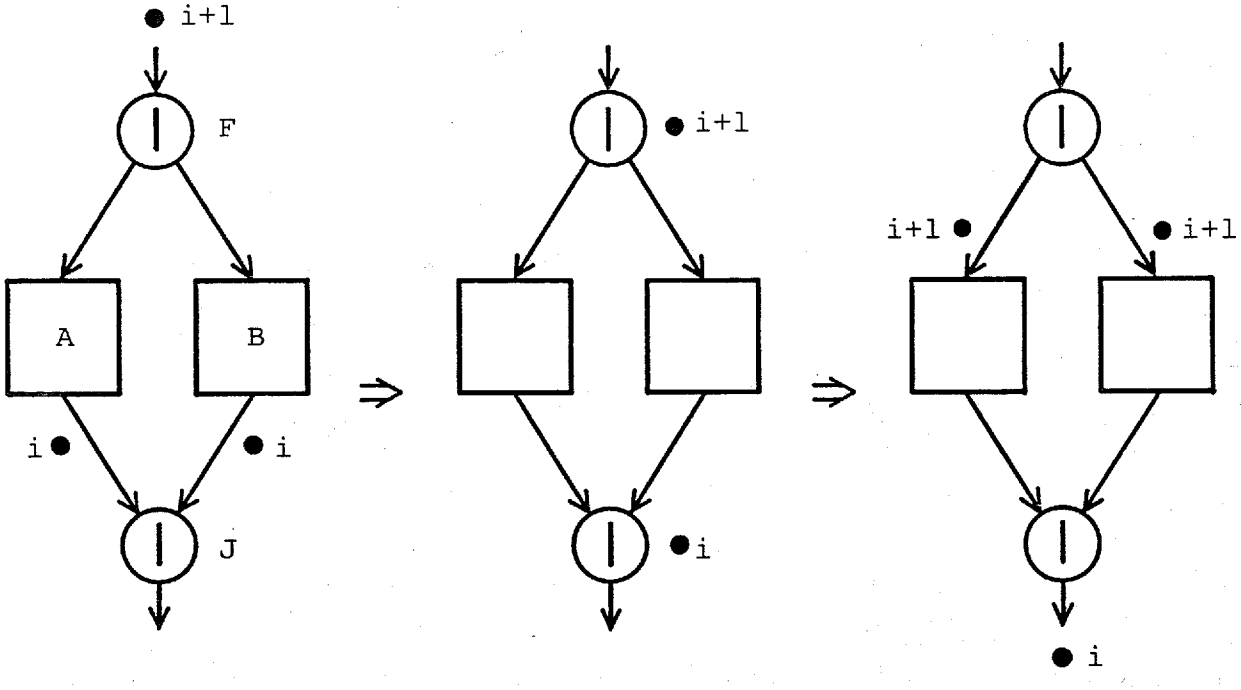


Fig. 3.7 Behavior of parallel module.

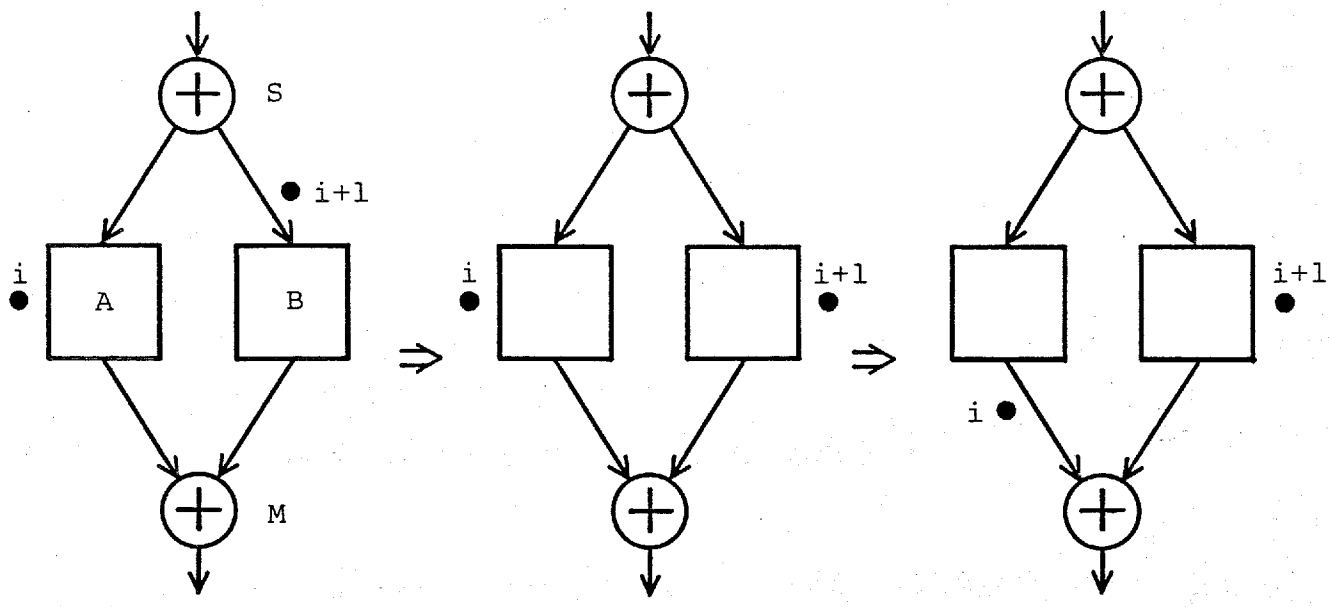


Fig. 3.8 Behavior of selective module.

3.3 Simulator

3.3.1 Requirements for Simulator

Before going into the main argument of a simulator, we mention the user environment of its usage and the requirements for its performance capability.

A. Environment

We assume that the user of this simulator operates it interactively, because the performance evaluation is required whenever the logical scheme is changed. For a simulator, we define

Inputs: Workload,
Outputs: Performance, and
Parameters: Software/Hardware Configuration.

A simulator is regarded as the subsystem of MMS's design support system. A total system configuration is not mentioned in detail because we concentrates on the performance evaluation in this chapter. If we are forced to mention MMS support system configuration, it consists of two major system, the editor and the simulator, which constructs and analyzes MMS model, using an interactive graphic terminal. For example, Fig. 3.9 illustrates the configuration of such interactive design system.

B. Requirements for Simulator

There are two major capabilities which are required for a

simulator. These are as follows;

(1) Performance Measurement and
(2) Behavior Inspection.

The outline will be discussed below.

Performance Measurement:

The first capability should be recognized as an extension to cover the performance evaluation for the dynamic workload. Any performance measure is derived from the time when a message comes into a module and when it leaves the module. The simulator can obtain the total time delay at a module, i.e., P+C+W, for any message. Therefore, any performance measure can be derived by the integration of them. The simulator can explore the following:

(1) The performance evaluation of workload-sensitive measures such as the response time, the input queue length and so on.
(2) The performance evaluation of the interactive scheme.
(3) The performance evaluation under the condition that there are hardware restriction between the logical communication link and its realization.

Behavior Inspection:

In order to quickly inspect the logical scheme to find bottlenecks or other performance problems, such as detection of a deadlock, it is needed for the simulator to visualize the message

flow on the scheme. On the real-time simulation, the message flow is visualized in the indefinite scaled time snapshots. The ratio of the simulated time to the real-time is controlled by a user. To change that ratio, the user invokes a simulator command and uses the keyboard to define the new ratio. The simulator can be terminated or interrupted at any time by calling a menu to the display. Otherwise, it terminates when there are no pending messages on the scheme. The inspection is performed for a module at any level.

If the logical scheme has a structured property, it is defined as the Parallel Flow Graph (in short, PFG), and both the performance measurement and the behavior inspection can be performed at each module on each hierarchical level [AE85b]. Fig. 3.10 illustrates also such concept, in which the inspection point or the measurement point is placed at the node of syntax tree. For the performance measurement, the simulator provides the result of measurement with the module designated by the measurement point. For the behavior inspection, the simulator visualizes the behavior of modules which are immediately subordinate to the module designated by the inspection point. Using this capability, we can improve the performance for each module. It is needed for a user to designate the module in order to set up the inspection or measurement point.

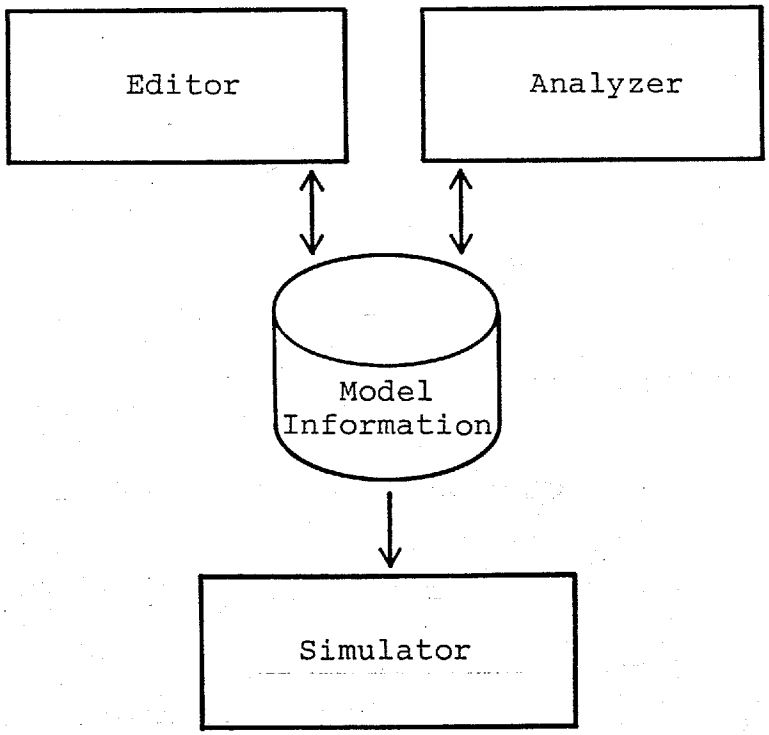


Fig. 3.9 Example of total system configuration.

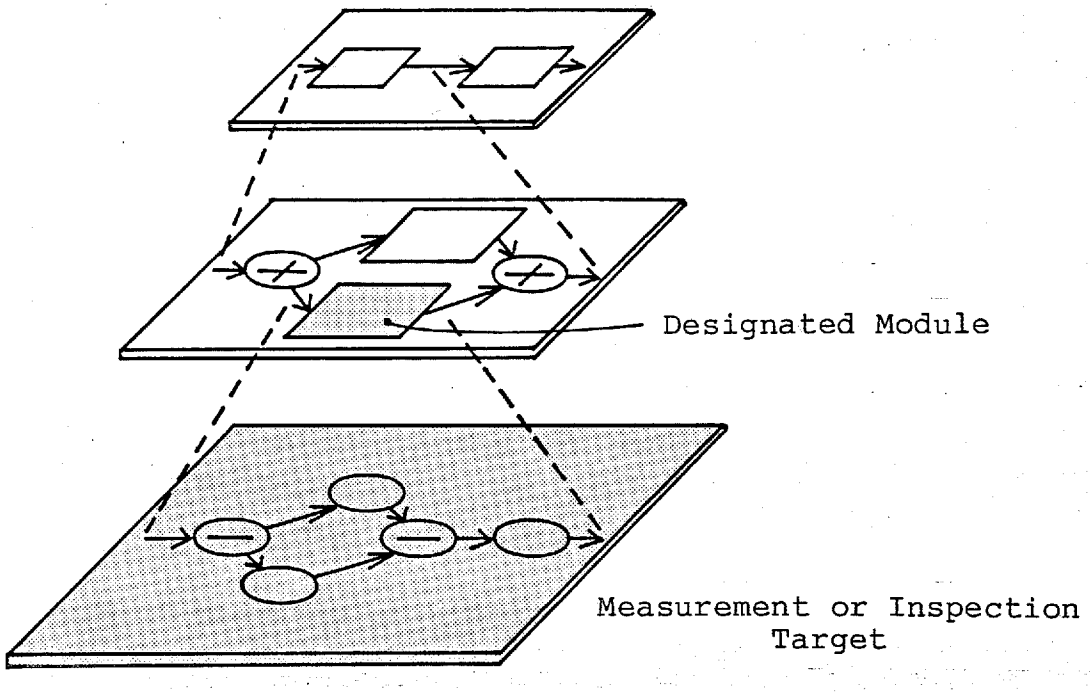


Fig. 3.10 Hierarchical measurement or inspection.

3.3.2 Simulator

To meet the above requirements, the following specification of the simulator is adopted.

A. Specification of Simulator

We specify the input, the parameters and the output of the simulator. For the parameters, we integrate them to additional input of the simulator. Consequently, the input of the simulator is as follows.

Input:

(1) Workload.

(2) Hardware/Software Configuration.

  (a) Logical Scheme of PFG

  (b) Time Factors

  (c) Hardware Information

  (d) Scheduling of Loop Node

(3) Simulator Control.

  (a) Break Point

  (b) Scaled Time Ratio

  (c) Measurement (or Inspection) Point

The output of the simulator is classified into two categories according to the simulation mode as follows.

Output:

Performance Measurement Mode:

(1) Input Interval and Output Interval of the designated module

for any message.

(2) The number of input and output messages of the designated module at any time.

(3) Input Queue Length at any time.

(4) Response Time of the designated module for any message.

Behavior Inspection mode:

(1) Status of the subordinate modules from the inspection point at any time.

(2) The message flow from the designated inspection point on the scaled real-time.

B. Implementation of Simulator

In the input of simulator, the workload, the logical scheme, the time factors and the designation of measurement (or inspection) points are realized by "script" which is coded like a program format. Other inputs are realized by the simulator inquiry. We will explain the input and output of the simulator briefly in an orderly manner.

Workload:

Since the workload is represented by an input sequence $(x_1, x_2, \ldots, x_n)$, we describe it as follows:

%{ Tin(1),Tin(2),....,Tin(n-1) }% ,

where n is the number of inputs. If n=1, the workload description is

%{ }% .

The symbol "%{" and "}%" indicate the begin and the end of character sequence, respectively. Tin(i) $1 \leq i \leq n-1$, which is a non-negative integer, represents the input interval between x_i and x_{i+1} for PFG model. This description can be simplified if consecutive input intervals take the same value. For example, the workload, which has 5 input data and the input interval T for all, is described as follows:

%{ 4(T) }% .

Logical Scheme of PFG:

To clarify the description, an example is given in Fig. 3.11. The example is described by the following symbolic form. Note that this symbolic form does not permit more than one construction for each line to provide the time factor description for control nodes such as fork, join, select, merge and loop nodes. Thus, M1=(E1|E2|E3|E4)*4 is not acceptable in the example.

```
@MAIN
MAIN=M1-M2
M1=A*4
M2=B1+B2
A=E1|E2|E3|E4
```

The symbol "@" identifies the most abstract module MAIN. Furthermore, we can simplify the notation for a number of equivalent modules, i.e., those modules are copied from an original module. For example, if E1=E2=E3=E4(=E), the notation A=E1|E2|E3|E4 is simplified to A=4|E. In the same

way, other cases are described as follows:

A=E1-E2-E3-E4 is simplified to A=4-E and

A=E1+E2+E3+E4 is simplified to A=4+E.

The detailed syntax of the script is described in the Appendix A.1. In the implementation, a sequential scheme is represented as "M1-M2" instead of "M1•M2" by reason that the symbol "•" is not acceptable in our machine.

Time Factors:

To simulate a PFG scheme, the processing time and communication time are provided for node primitives. In the case of the above example, they are described as follows:

Process node E1 is described as

#E1(C3,P3,C4),

where E1 is the identifier of the process node, and C3, P3, and C4 are the communication time for an input, the processing time, and the communication time for an output, respectively.

Fork/join nodes are described as

#A(C2,P2|P4,C5),

where A is the identifier of the parallel module, and C2, P2, P4, and C5 are the communication time for an input of the fork node, the processing time of the fork node, the processing time of the join node, and the communication time for an output of the join node, respectively.

Select/merge nodes are described as

#M2(C6,P6|P8,C9),

where M2 is the identifier of the selective module, and C6, P6, P8, and C9 are the communication time for an input of the select node, the processing time of the select node, the processing time of the merge node, and the communication time for an output of the merge node, respectively.

Loop node is described as

#M1(C1,P1,C6),

where M1 is the identifier of the iterative module, and C1, P1, and C6 are the communication time for an input of the loop node, the processing time of the loop node, and the communication time for an output of the loop node, respectively.

In the implementation, those time factors are represented as non-negative integers.

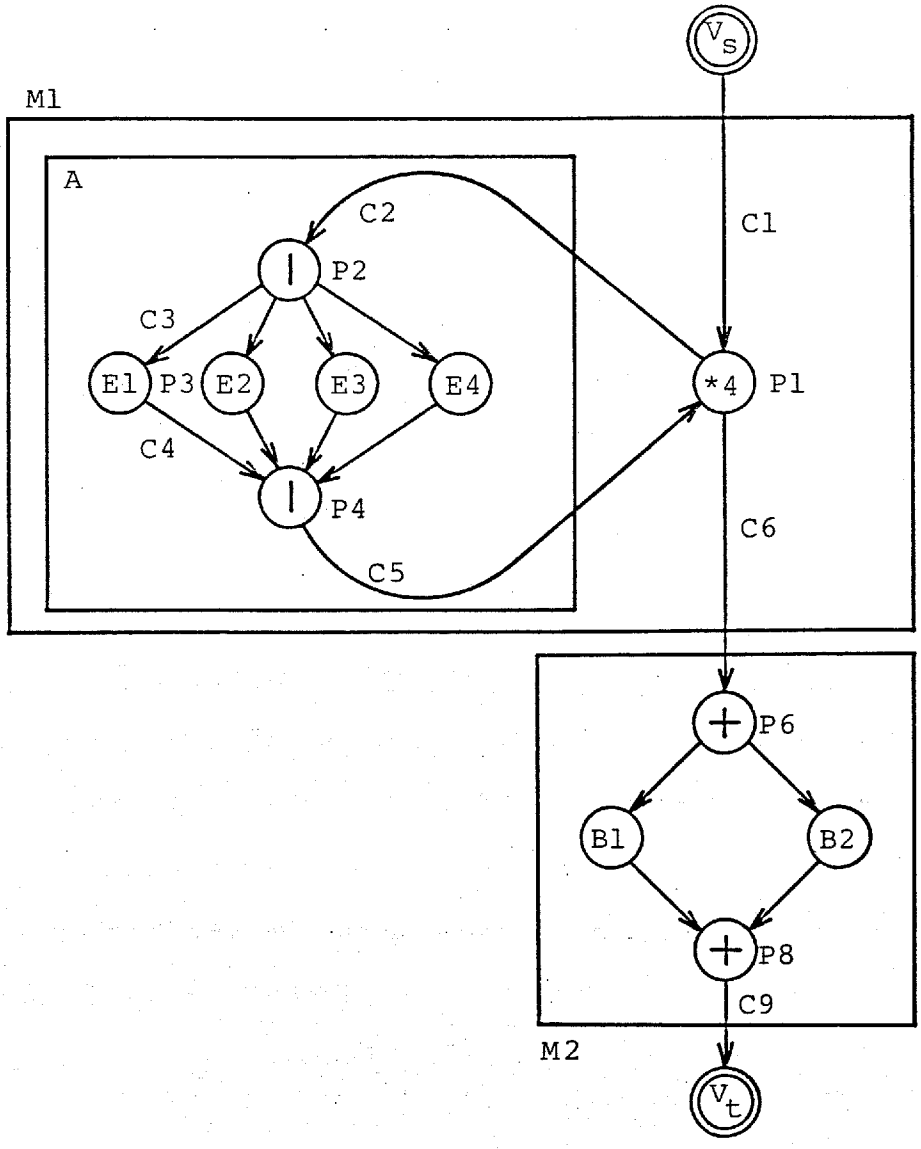


Fig. 3.11 Example of logical scheme including a loop.

Designation of the Inspection or Measurement Point:

A user can designate an arbitrary module as the objective module to be inspected or measured. Thus, the simulation result is presented for the designated module. For example, if the user intends to measure the module A in Fig. 3.11, he can designate A in the following manner:

observer { /MAIN/M1/A } .

If he intends to get the total performance, the description is as follows:

observer { /MAIN } .

Therefore, module designation is performed by "path expression" on the the module hierarchy of a logical scheme.

Other Inputs:

Inputs of hardware information, scheduling of loop nodes, the break point and the scaled time ratio are realized by inquiry of the simulator. The simulator provide some questions to the user to obtain the above inputs. For the hardware information, following items are considered.

(1) Capability of Processor,

(2) Capability of Communication Link, and

(3) Interconnection Network Configuration.

(Point-to-Point, Bus, etc.)

For the scheduling at a loop node, following two cases are considered as a convenient and temporary method:

(1) No scheduling.

(2) Restrict the message incoming of the iterative module according to the message outgoing.

Break points are classified into two categories which are the "space" break point and the "time" break point. In the space break point, if a user designates a process node and its status, simulator breaks his behavior when the designated process node becomes the specified status. In the time break point, the user sets a logical time. The simulator breaks his behavior at the specified logical time.

The scaled time ratio has been implemented. In addition to original capability, it is able to instruct "step by step" behavior. That means that logical clock is stopped until a user invokes a simulator command.

Outputs:

The outputs have been already realized except the behavior inspection. At the present stage, the graphical display is not realized owing to the hardware environment. Instead of the graphical display, as a very simple and handy method, we have realized "status snap" in which the events caused by a message flow are reported as statements.

Fig. 3.12 illustrates the simulator configuration. As is mentioned above, Script includes the description of the workload, the time factors and the measurement (or inspection) point.

The simulator can work even on a small UNIX machine (Radio Shack TRS-80 Model 16 with XENIX^{TM}), where about maximum 600 primitive nodes can be realized. About 50,000 primitive nodes can be realized on VAX 11/750.

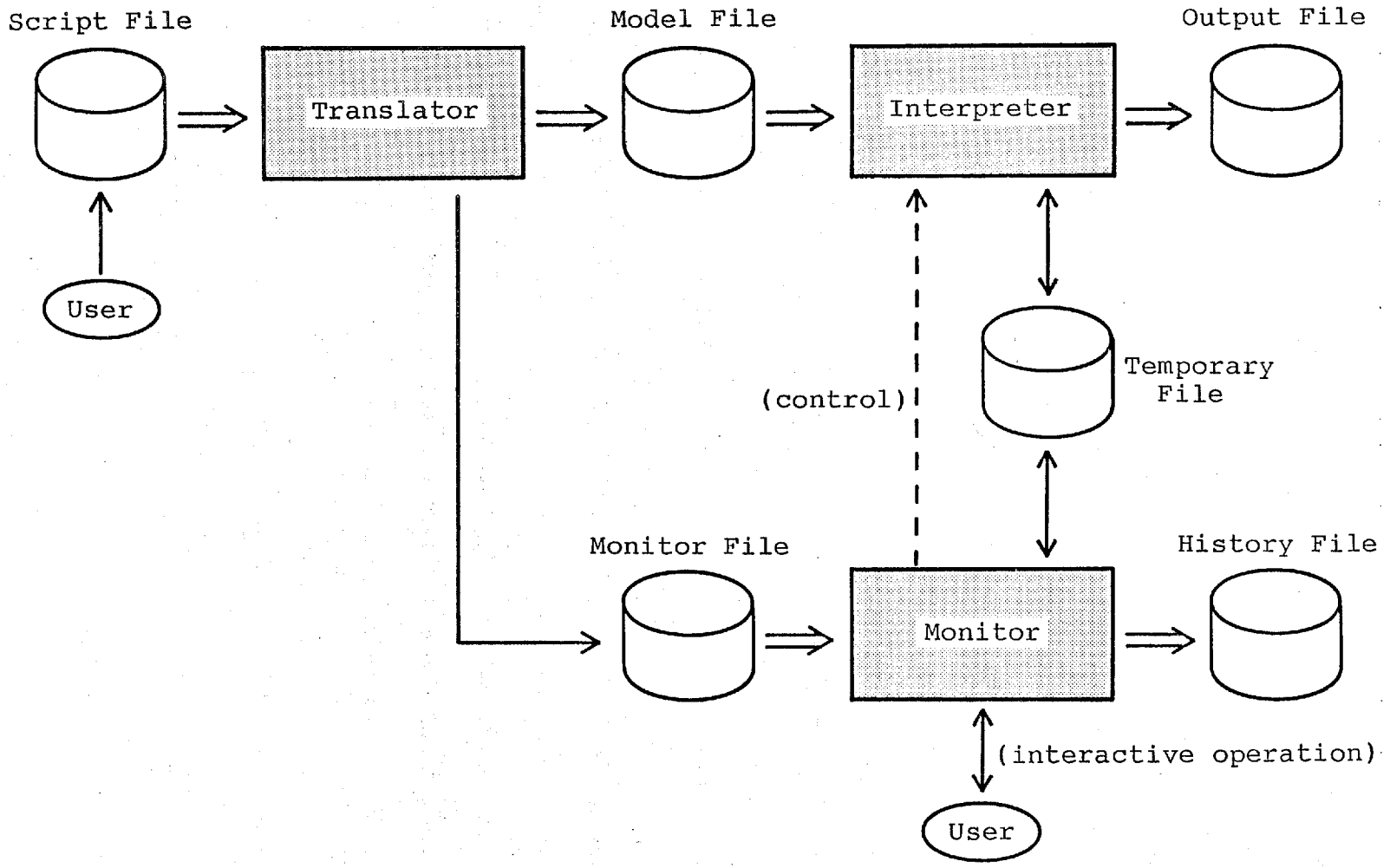


Fig. 3.12 Simulator configuration.

3.4 Case Studies

To demonstrate the use of simulator, we provide an example of the script in the following. The section enclosed in the marks "/*" and "*/" declares a comment in which any description is ignored. In this example, the horizontal broken lines with PFG's names (Thus, these are comments.) are added to indicate the degree of PFG construction. PFG construction starts with a PFG G_0 which consists of one process node. G_1 is constructed from G_0 by applying the sequential construction: MAIN=A-B-C-D. In the same manner, $G_2 \ldots G_5$ are constructed in order. Therefore, those have the relation such that G_0 G_1 G_2 G_3 G_4 G_5.

Since this script represents the PFG G_5 about the description of the logical scheme and time factors, in the case of G_i evaluation($0 \leqq i \leqq 4$), the description between the broken line with G_5 and one with G_i must be omitted. In this case, the workload is described as 50 inputs which have the same input interval 75 respectively. The simulation result of this example is shown in Fig. 3.13. Fig. 3.13 illustrates the response time improvement associated with G_0,G_1,G_2,G_3,G_4 and G_5.

The simulator is still been developing for the case of general logical scheme.

An Example of Script:

```
    @MAIN          #MAIN(50,2000,2)
/* G0  - - - - - - - - - - - - - - - - - - - - */
    MAIN=A-B-C-D   #A(50,1000,30) #B(30,800,20)
```

```
                    #C(20,300,20)   #D(20,60,2)
/* G1  - - - - - - - - - - - - - -  */
     A=(5-I)-G      #I(50,200,50)   #G(50,50,50)
     B=(2-F)-E      #F(30,400,30)   #E(30,100,20)
/* G2  - - - - - - - - - - - - - -  */
     F=5|J          #F(30,5|5,30)   #J(10,100,10)
     C=10+N         #C(20,20|5,20)  #N(20,300,20)
     I=10|M         #I(50,10|5,50)  #M(5,30,5)
/* G3  - - - - - - - - - - - - - -  */
     G=4+S          #G(50,2|1,50)   #S(50,50,50)
     E=X-Y-Z        #X(30,10,30)    #Y(30,80,30)
     J=P-Q-R        #Z(30,10,30)    #P(10,40,20)
                    #Q(20,40,20)    #R(10,30,10)
/* G4  - - - - - - - - - - - - - -  */
     Y=4|W          #Y(30,10|5,30)  #W(8,20,8)
     D=2+O          #D(20,4|2,2)    #O(20,60,2)
/* G5  - - - - - - - - - - - - - -  */


/* Workload */
     %{ 49(75) }%


/* Measurement Point */
     observer { /MAIN }
                                End of Script
```

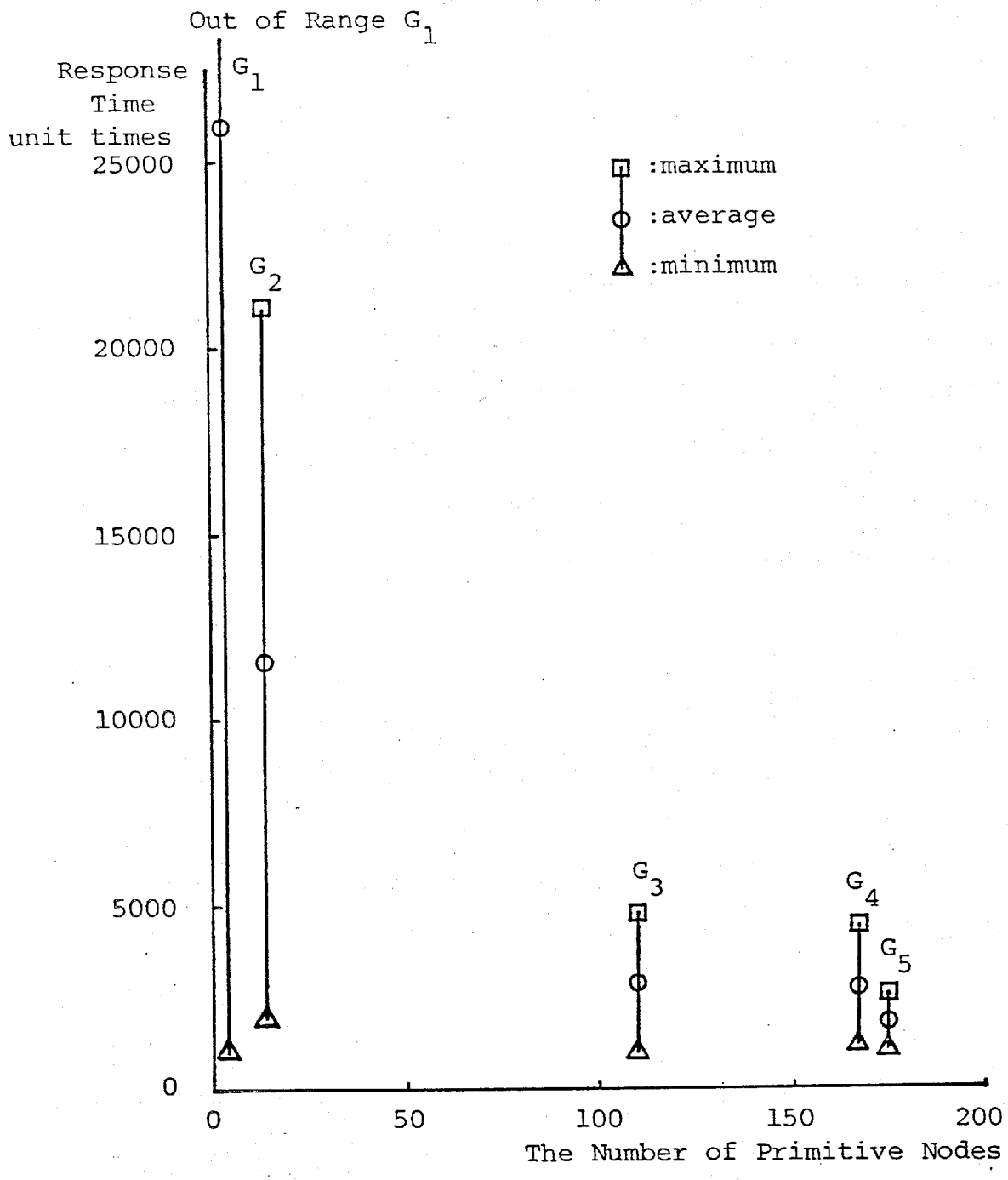


Fig. 3.13 Example of simulation results.

Furthermore, we demonstrate the simulation results of 2-dimensional bit-pattern matching presented in subsection 2.5.2. The job can be divided into parallel part and pipelined part, sequentially, where the sum of the parallel processors and the pipelined processors is fixed to 29. The simulation input data is based on the execution time derived from the individual part measurements of the realized program (as shown in Table 2.5). Fig. 3.14 shows the simulation results comparing with the experimental results, and Table 3.1 shows the detailed results. The simulation consuming time for the experiment is about 70 seconds per one plotted point where Radio Shack TRS-80 model 16B is used, and about 25 seconds per one plotted point where DEC VAX-11/750 is used. The script of the simulation is described as follows.

Script of 2-Dimensional Bit-Pattern Matching:

```
      $n=$1            /* the number of GA */
      $m=29-$n         /* the number of REC */

/* Scheme */
      @main
      main=ga-rec
      ga=($n)+gasub
      rec=bank1-($m)-recsub-bank2-mast2

/* Time factor */
      $port_A = 2114
```

```
    $bus_A   = 297
    $bus_B   = 72

    $mast1   = 1
    $mast2   = 1
    $ga_p    = 48885

    #ga($port_A,$mast1|$mast2,bus_B)
    #gabus($bus_A,$ga_p,$bus_B)

    $portB1 = 513
    $busB   = 72
    $portB2 = 320
    $busC   = 9
    $portC  = 64

    $a=5910*14/$m+300

    #bank1($busB,1,$portB2)
    #recsub($portB2,$a,$portB2)
    #bank2($portB2,1,$busC)
    #mast2($busC,1,$portC)

/* Input sequence */
    %{ 102(0) }%
/* Measurement point */
    observer { /main }
```

End of Script

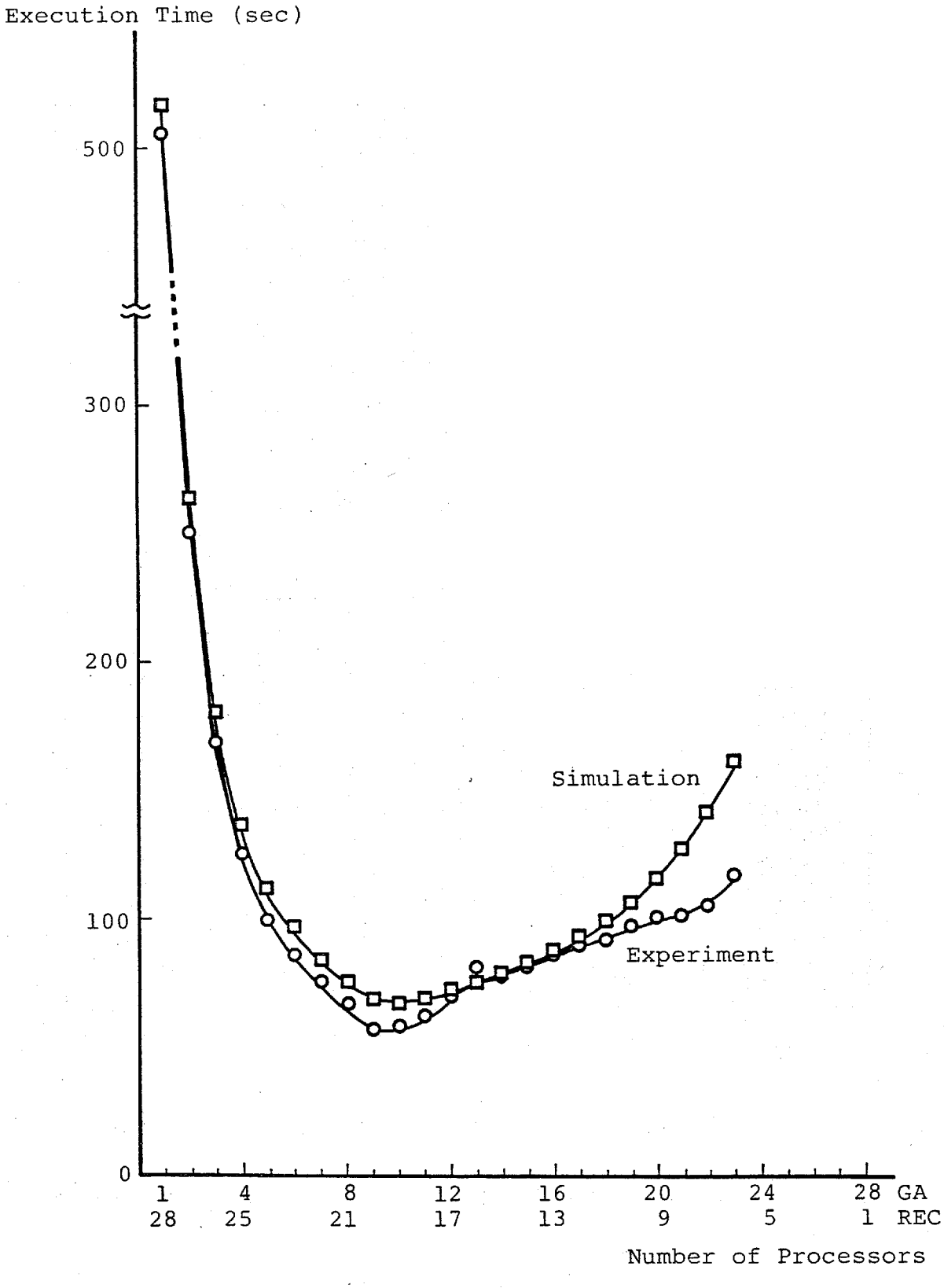


Fig. 3.14 Simulation results compared with experimental results.

— Two-dimensional bit-pattern matching. —

Table 3.1 Simulation results of 2-dimensional bit-pattern matching.

Number of GA	Number of REC	Simulation Results (sec)		
		Total	t_{int}	t_{res}
1	28	51.76	0.4879	26.639
2	27	26.63	0.2419	13.971
3	26	18.25	0.1601	9.758
4	25	13.90	0.1212	7.661
5	24	11.43	0.0970	6.411
6	23	9.86	0.0789	5.584
7	22	8.57	0.0685	5.002
8	21	7.68	0.0597	4.572
9	20	7.04	0.0538	4.245
10	19	6.86	0.0513	4.163
11	18	7.10	0.0529	4.281
12	17	7.37	0.0546	4.413
13	16	7.68	0.0566	4.562
14	15	8.02	0.0589	4.731
15	14	8.41	0.0615	4.926
16	13	8.88	0.0646	5.151
17	12	9.41	0.0681	5.416
18	11	10.04	0.0723	5.729
19	10	10.81	0.0774	6.107
20	9	11.74	0.0836	6.569
21	8	12.90	0.0914	7.149
22	7	14.40	0.1014	7.897
23	6	16.41	0.1149	8.895
24	5	19.21	0.1337	10.296
25	4	23.43	0.1620	12.399
26	3	30.45	0.2092	15.910
27	2	44.51	0.3038	22.936
28	1	86.71	0.5876	44.029

3.5 Summary

This chapter has mainly demonstrated the simulator as an MMS's design support tool which can estimate the systems performance. Performance of MMS's was evaluated by two measures, i.e., response time and interval time, from which throughput, parallelism etc. can be derived. We can find that it was sufficient to evaluate the restricted processing scheme constructed by pipelined and/or parallel processing in MMS's. Moreover, the simulation experiments compared with the multiprocessor execution indicated that the simulator is high-speed and is suitable to the systems.

The simulator can accept the restricted scheme, i.e, extended series-parallel flow because of the restriction of its translator. The interpreter, however, is possible to apply an arbitrary scheme. Therefore, the simulator can accept another scheme by exchange of the translator for another one. For example, we have implemented a new translator for 2-demensional array simulation. Appendix A.2 demonstrates a syntax of script for 2-dimensional array scheme.

Chapter 4

Interconnection Network Based on Three-Dimensional Integrated Circuits

A realization of MMS's based on PMS level architectures is presented in this chapter. The most crucial issue of PMS architecture realizations is the Switch, that is, interconnection between processors or between processors and memory modules. Though researchers have proposed various techniques, there are few proposals suitable for realization of MMS's with many processing elements. In this chapter, we propose a common memory with very low access conflicts, based on a new device technology, that is, 3-dimensional integrated circuits [AE85a]. Moreover, an interconnection network incorporating the common memory as its elements is presented for a realization of MMS's.

Though the crossbar is the best known scheme for connecting n processors to n memory modules (or processors), its cost, i.e., $O(n^2)$ becomes too expensive for a large number of n. Therefore, a lot of modifications have been discussed, one kind of which is the multistage network with the cost of O(n log n). Though the multistage network is also a good scheme because of the full access property [SULLIVAN77], the bandwidth (or the throughput) is not so high as the crossbar switch. In order to improve this, Dias et al. [DIAS81] have proposed the buffered delta network, which performance may become comparable to that of the crossbar switch as to the bandwidth. The buffered delta network, however, suffers from additional delay arising from a

number of buffering besides to the delay of O(log n) which exists originally in the multistage network.

4.1 Overview

A typical multiprocessor configuration is shown as in Fig. 4.1, which represents the memory modules explicitly. In this configuration the crossbar switch is the best scheme, disregarding the cost of $O(n^2)$, where n is the number of ports. Therefore, it has been actually used for cases of relatively small n (e.g., n=16 at C.mmp [WULF81]). For the massive multiprocessor case (e.g., $n=10^3$), however, its cost is considered too expensive to be realized.

In order to decrease the network cost, the packet switching interconnection network with multistages (shown as in Fig. 4.2) is becoming popular [FENG72], because it has the reasonable cost, i.e., O(n log n) and the full access property [SULLIVAN77] that all ports can access distinct destinations simultaneously. In this case the memory does not appear explicitly and the packet flows unidirectionally.

The bandwidth (or the throughput) of the multistage network, however, is lower than the crossbar switch, and, to improve this, Dias et al. [DIAS81] have proposed the buffered multistage interconnection network (actually, the buffered delta network) which performance may become comparable to that of the crossbar switch as to the bandwidth. The buffered delta network, however, includes the delay increasing with the buffer size besides to the delay of O(log n) which is essential in the

multistage network.

In this chapter, we discuss an alternative way of improving the throughput, where the common memory or the multiport memory (we often refer to it as n-port memory) plays an important role. The multiport memory is, in a sense, logically equivalent to the memory with the crossbar switch. An n-port memory is shown as in Fig. 4.3. Suppose that the size of memory is large enough to divide the area into sub-areas used for processor-to-processor communication as in Fig. 4.4, where a_i and P_i mean the address and the processor, respectively.

The features of the idealized n-port memory are listed as follows:

(i) random access is allowed, and

(ii) reader and writer processes can be executed concurrently, because they can read and write all addresses simultaneously except that the same address cannot be accessed at a same time by multiple writer processes.

Obviously, the idealized n-port memory is also expensive, rather more than the crossbar switch with the memory. Therefore, we introduce the way of reducing the complexity with the multistage structure, where the module on each stage has a restricted size (e.g., 2x2-port memory). Moreover, such a module is assumed to be realizable by the three-dimensional VLSI technology [AE84b][AE85c]. We call this network the **COMBINET** (**CO**mmon-**M**emory-Based **I**nterconnection **NET**work).

The features of COMBINET are summarized as,

(1) the throughput is essentially the same as the buffered multistage network,

(2) the delay of each stage is kept constant for any buffer size as long as possible to improve the throughput, and

(3) the multiple channels on a path and the broadcast are also easily realizable.

Features (2) and (3) come from the property of the common memory, which is assumed to be realizable by the multi-layered 3-D VLSI technology.

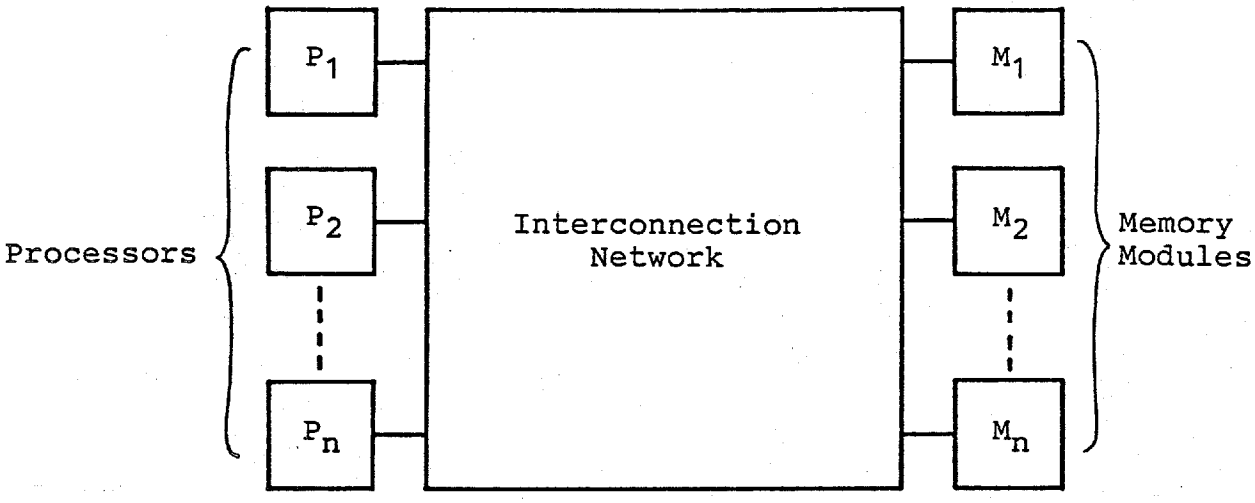


Fig. 4.1 Typical multiprocessor configuration.

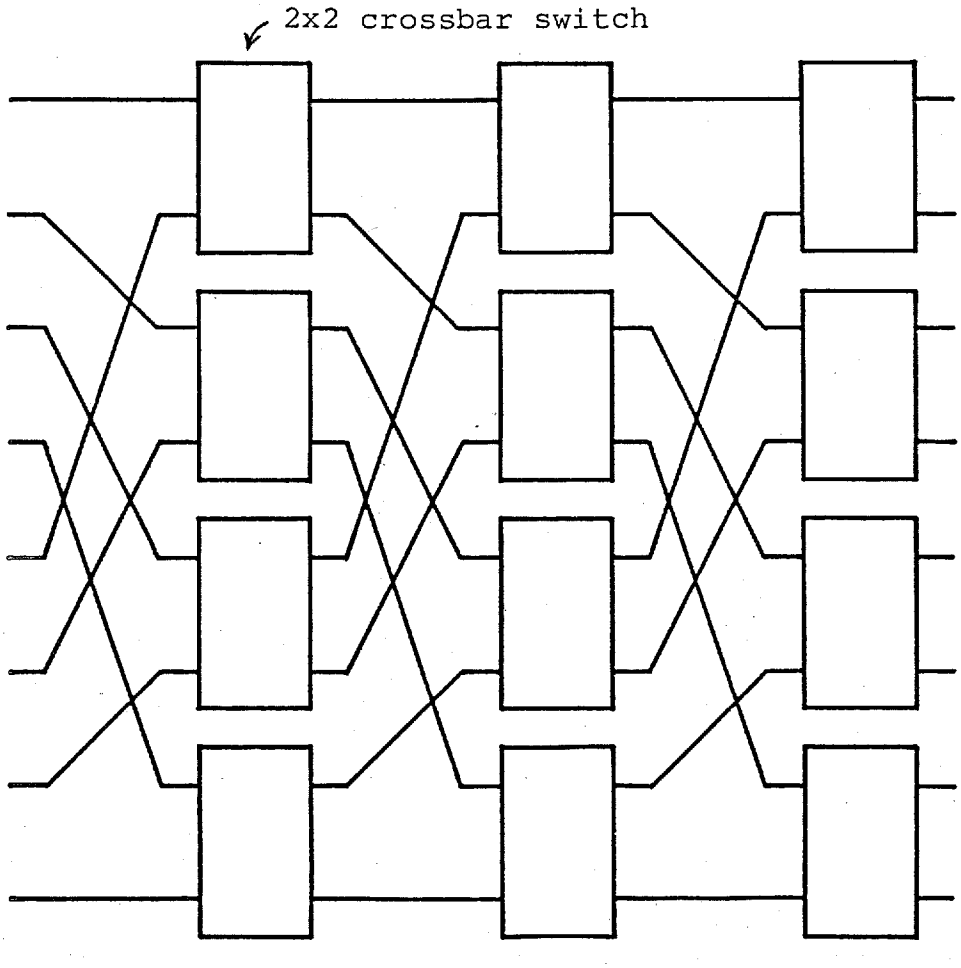


Fig. 4.2 Typical multistage network.

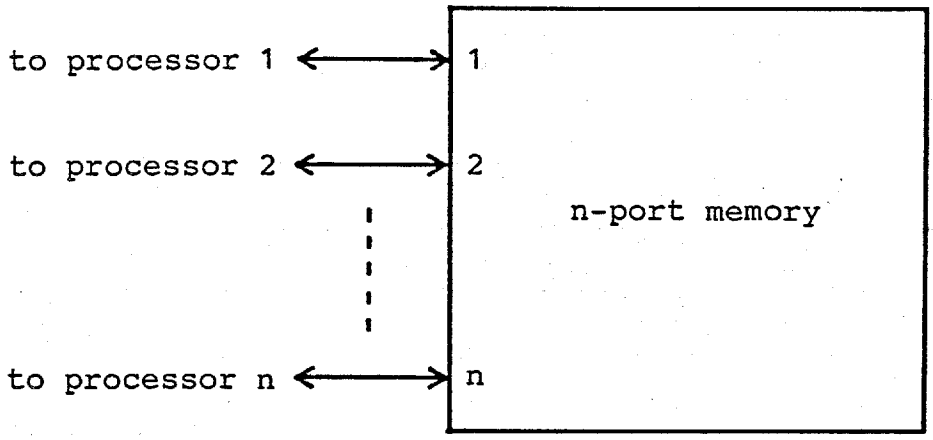


Fig. 4.3 N-port memory.

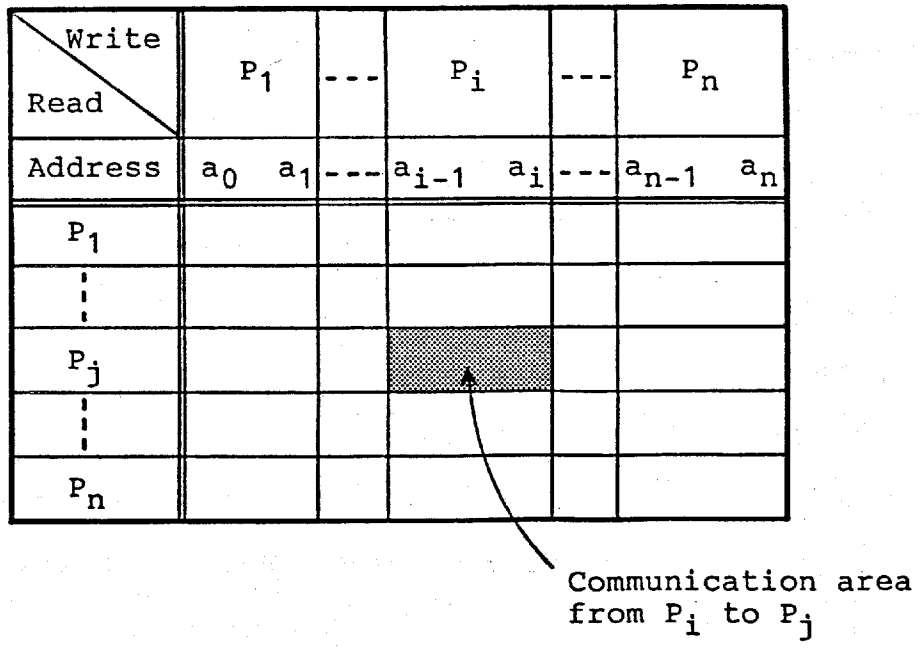


Fig. 4.4 Memory area divided into each processor-to-processor communication.

4.2 Three-Dimensional Common Memory

About the multiport memory, Chang [CHANG80] has proposed a multiple-read, single-write memory which can be realized with the conventional (i.e., two-dimensional) LSI/VLSI technology. Recent semiconductor technologies make it possible to realize the three-dimensional integrated circuit [KAWAMURA83][KAWAMURA84], which is expected to be extensible to the large-scale or even to the very-large-scale. When assuming the three-dimensional VLSI (in short, **3-D VLSI**) technology, the multiple-write as well as the multiple-read can be realized easily [AE84]. Note that, even in this type of memory, only multiple-write in the same address at a time must be avoided.

In this chapter, however, we focus the communication between processors by the message passing (like the programming in Ada^{TM} [DOD80], Occam^{TM} [INMOS84] etc.), instead of the variable sharing communication. For this case the conflict of multiple-write at the same address access is ignored, because P_i ($i=1,\cdots,n$) in Fig. 4.4 can read the whole area but can write only the area from a_{i-1} to a_i.

In this section, we describe how to design a multiport memory with such multiple accesses using the idealized 3-D VLSI technology. Suppose that the fundamental part (hereafter, we call it the **memory cell**) of the memory consists of NMOS SRAM (N-type Metal-Oxide-Semiconductor Static-Random-Access-Memory). Our proposal of the memory cell is shown in Fig. 4.5, where Q_1, Q_2, Q_3, Q_4, R_1, and R_2 construct a memory cell in the conventional NMOS SRAM, and Q_1, Q_2, R_1, and R_2 work as a flip-flop for holding the single bit data. Though Q_3 and Q_4 are

used for both read and write of data in the conventional memory cell, they work only for data-read, and Q_7 and Q_8 are provided especially for data-write.

The total configuration of a three-dimensional common memory is shown in Fig. 4.6, where each layer works as its own memory and the value of the same address is always identified through all layers.

The interconnection among memory cells at vertical axis is shown in Fig. 4.7. When the write-access for the memory cell at address j of i-th layer occurs, the cell is accessed through Q_6 and Q_7, or, Q_5 and Q_8 for data-write. At the same time the cell at address j of all layers receives the same data through Q_7 and B_1, or, Q_8 and B_2. At other cells, Q_5 or Q_6 writes the data received from B_2 or B_1.

B_1 and B_2 work as buses through all layers when a data is written at a layer. The behavior of B_1 and B_2 is represented as in Table 4.1.

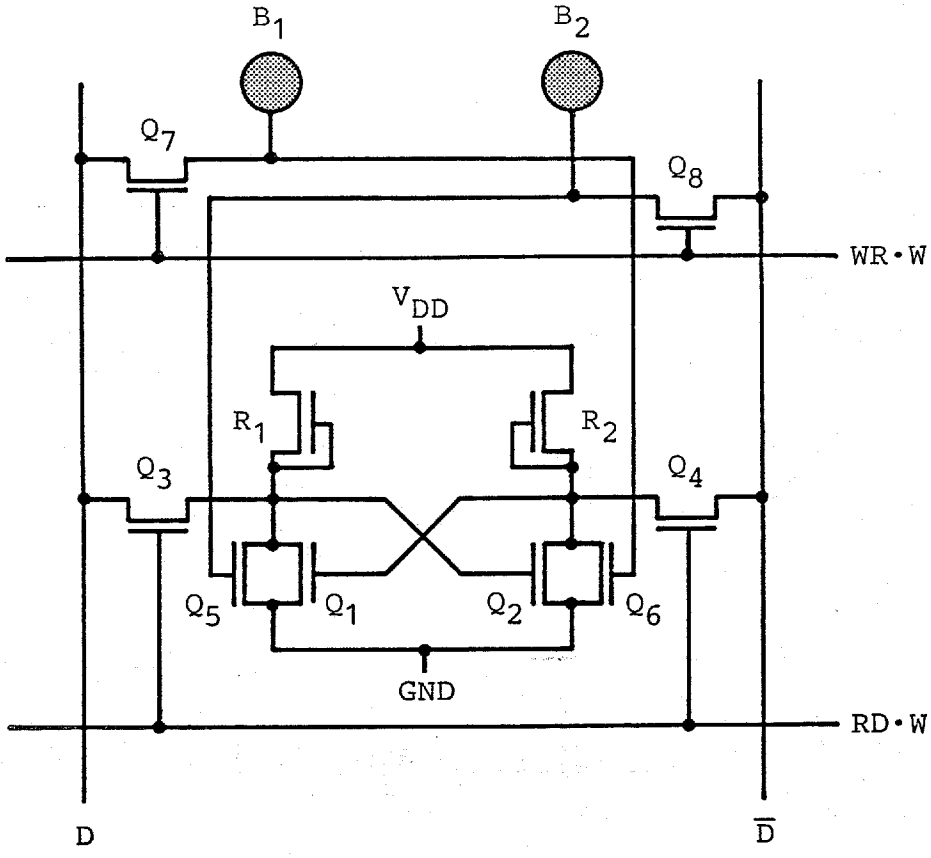
Table 4.1 Behavior of B_1 and B_2.

B_1	B_2	Behavior
0	0	Nothing.
0	1	Write "0".
1	0	Write "1".
1	1	Conflict.

The conflict occurs only when the different values are written into the same address at multiple layers (e.g., "0" at p-th

layer and "1" at q-th layer, where p≠q). The problem of conflict, however, is not discussed here because of the reason described above.

Obviously, the number of layers corresponds to the number of ports. Even if we assume the multi-layered 3-D VLSI memory, the number of layers should be expected rather small (e.g., less than ten). For convenience, we assume that the number of layers is four (,which is known to be already realizable experimentally). Hereafter we restrict our attention to the four-layered 3-D VLSI memory, i.e., the four-port (or 2x2-port) memory.



D : Data
RD: Read
WR: Write
W : Word Selection

Fig. 4.5 Example of memory cell.

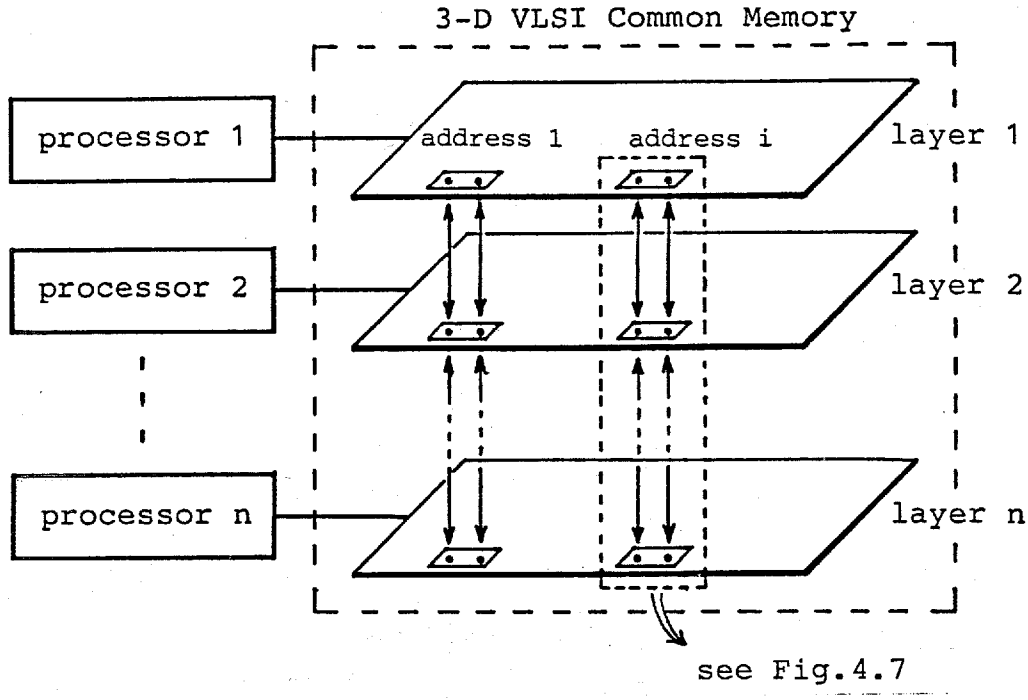


Fig. 4.6 Three-dimensional common memory.

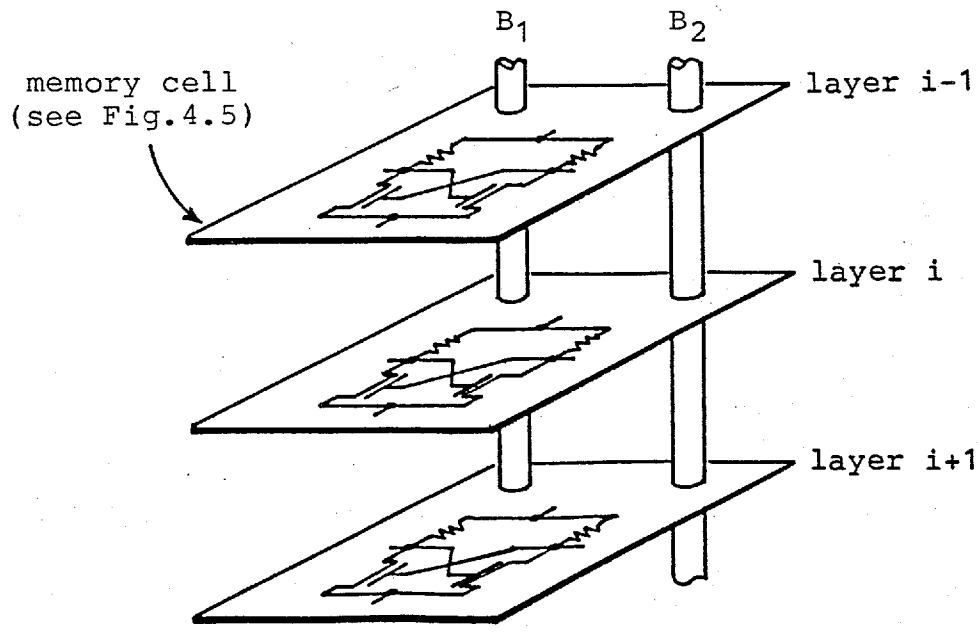


Fig. 4.7 Interconnection among memory cells at vertical axis.

4.3 Common-Memory-Based Interconnection Network

The COMBINET is obtained from the idealized n-port memory as the multistage network is derived from the crossbar switch. Therefore, the topology of COMBINET is the same as the multistage network. Though many possible configurations may exist for COMBINET, we restrict our attention to the type of multistage configuration consisting of the 2x2-port memory modules (four-layered 3-D VLSI realizable, and topologically, the same as the 2x2-switching-element networks) due to the reason described in the previous section.

A 2x2 dual interconnecting modular network device —Dimond— is shown in Fig. 4.8, quoting from [JANSEN80]. On the other hand, the module of COMBINET is shown in Fig. 4.9 which is based on a 2x2-port memory. Comparing this module with the Dimond,

(1) the 2x2-port memory block is very large. It occupies almost the total size of the module, and
(2) the transfer block plays a role of data-transfer processor, together with the write block connected to it.

The 2x2-port memory has logically four circular queues on the memory space. Each queue has two pointers, i.e., one is a pointer to read a data from the queue, and the other is to write in (see Fig. 4.10). The transfer block of module i and the write block of module j are connected as shown in Fig. 4.11. The transfer block of module i reads the data from its own address which the read pointer indicates, and send it to the

write block of module j connected to module i. This operation repeats until the packet ends. Each packet includes the routing information, i.e., path code or destination address, and the packet length. The write block of module j begins to write the data from module i into the 2x2-port memory of module j just after it receives a data (e.g., 32 parallel bits). Note that the transfer block and the write block work together just as a data-transfer processor, and that this operation of two blocks is pipelined only with the delay of **transfer time** as shown in Fig. 4.12. The transfer time is constant for the direct data-transfer and is included within the time required to transfer a packet through the module (t_pass [DIAS81]).

In this chapter the case of the direct data-transfer is only discussed. However, the function f may be added through transfer as shown in Fig. 4.13. Since f may increase the transfer time, only simple functions are allowed. (The transfer time should be uniform even if f is not unique on each module.)

Each module has four-way connections for two inputs and two outputs as shown in Fig. 4.14. Note that 1-3 and 2-3 (or 1-4 and 2-4) transfers can be done in parallel as well as 1-3 and 2-4, or 1-4 and 2-3, where the number corresponds to the port. This comes from the fact that the 2x2-port memory has separated queue buffers for each transfer. Reading operation, however, is done sequentially for the buffer including more than one packet sequence. The similar situation occurs for multiple broadcasting. When the data is broadcasted, e.g., from port 1, both the buffer of 1-3 and that of 1-4 include the same data.

Note that these broadcasted transfers are done in parallel with another broadcasting (2-3 and 2-4) and that the sequential reading is required also for this case.

The COMBINET may have all configurations already realized in the multistage network consisting of 2x2-port modules. Since multiple transfers between ports in the module can be done in parallel, the blocking (i.e., the conflict in the communication link) does not occur. An 8x8 shuffle-exchange COMBINET is shown in Fig. 4.15, where the line with a dot shows the connected transfer block with the write block.

The memory size of the module is estimated as shown in Table 4.2, comparing it with the case of the conventional SRAM. The estimation is done by figuring the three-dimensional mask pattern of the memory cell. The buffer size of each queue (e.g., 85K bits) is long enough to improve the throughput.

Table 4.2 Estimated capacity of a 3-D VLSI RAM.

2-D SRAM	2x2 3-D VLSI RAM	
bit/chip	bit/chip	bit/queue
64K	21K	5.3K
256K	85K	21K
1024K	340K	85K

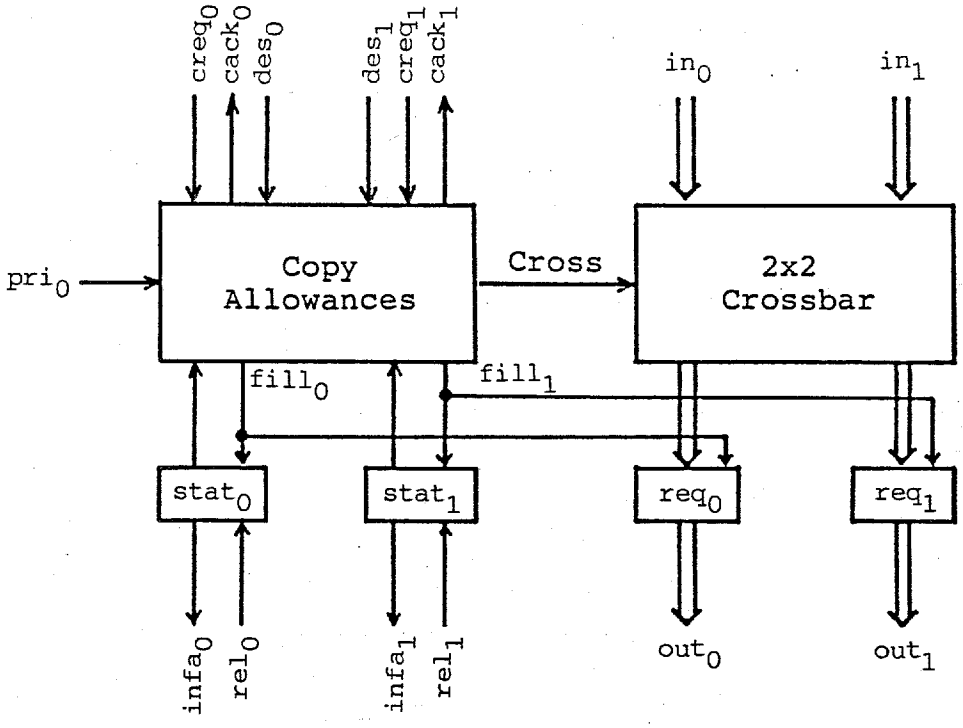


Fig. 4.8 A 2x2 dual interconnecting modular network device "Dimond" for packet switching [JANSEN80].

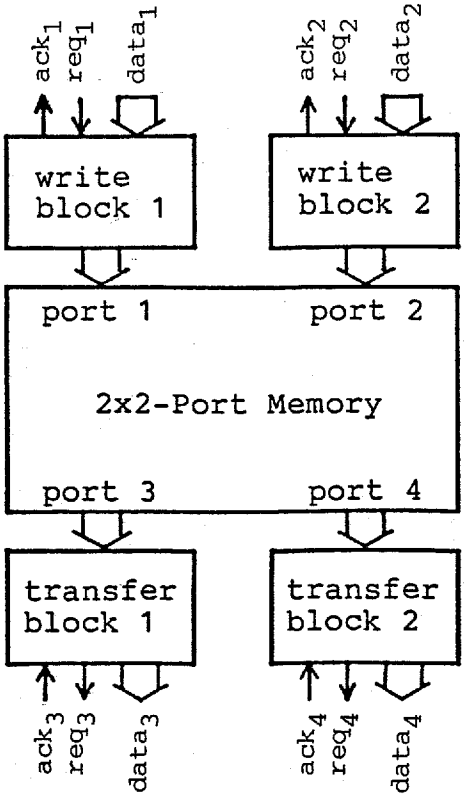
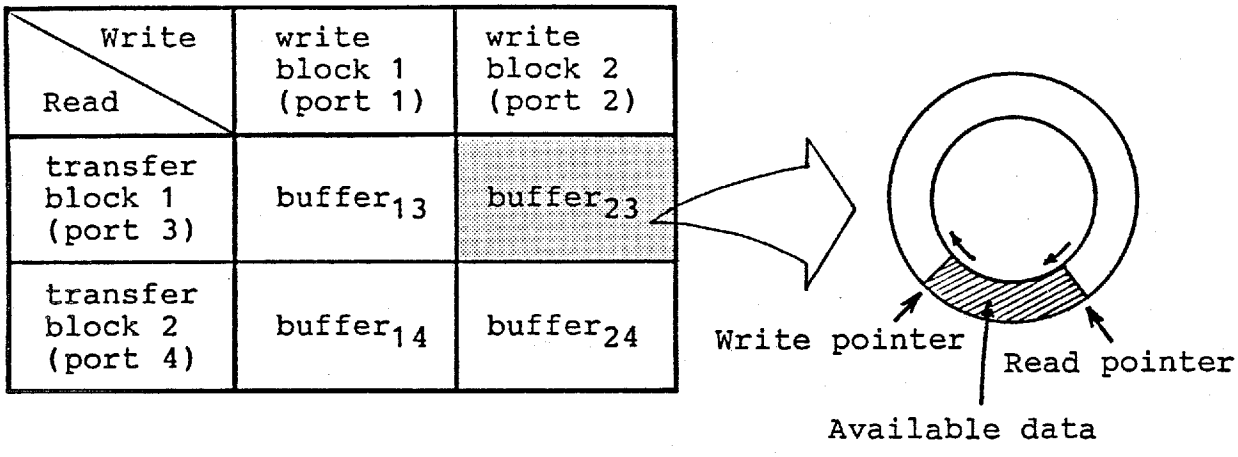


Fig. 4.9 A 2x2 dual interconnecting modular network device based on a 2x2-port memory.



(a) Memory space. (b) A circular queue.

Fig. 4.10 Circular queues on a 2x2-port memory.

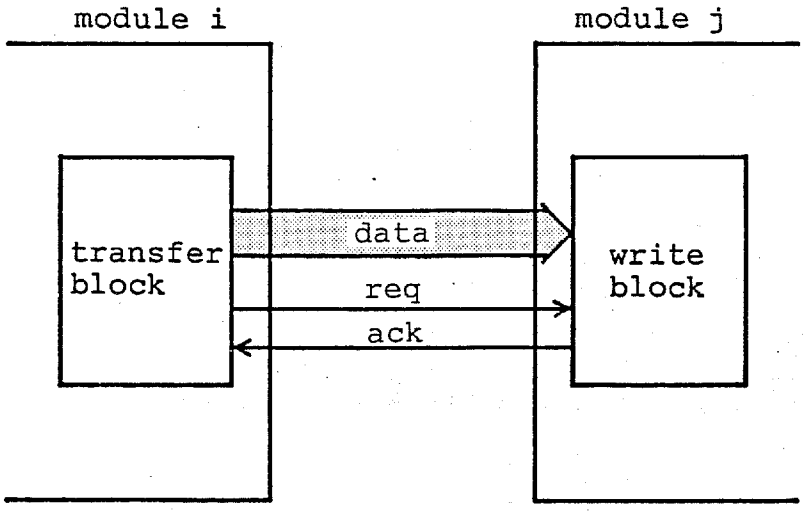


Fig. 4.11 Connecting two modules.

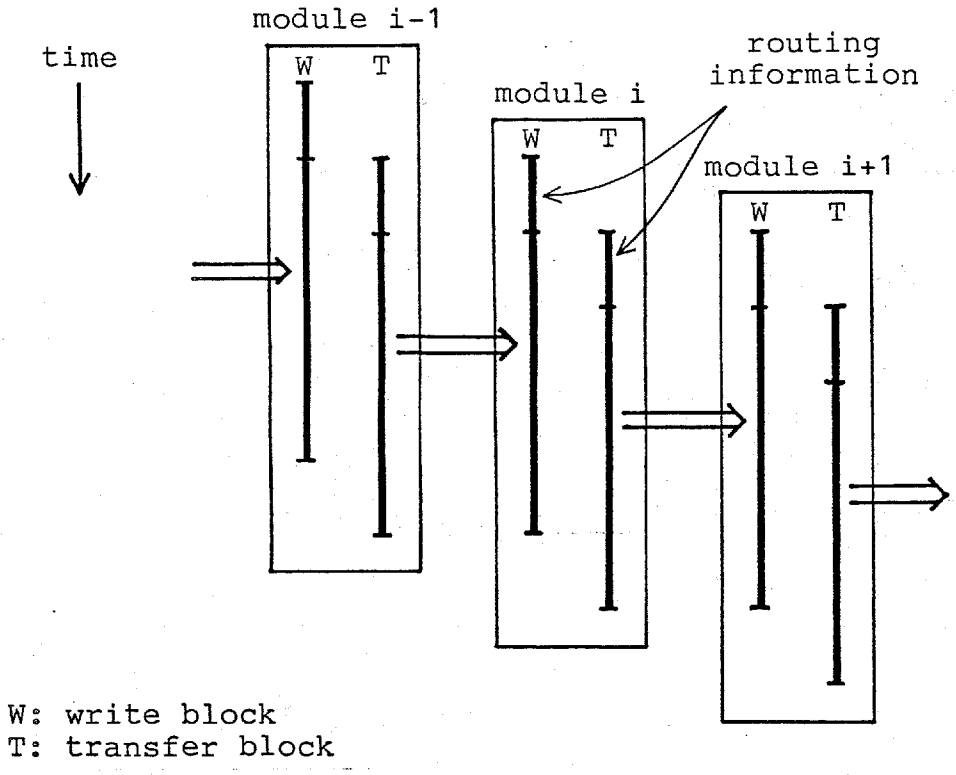


Fig. 4.12 Pipelined time chart of transfer between modules.

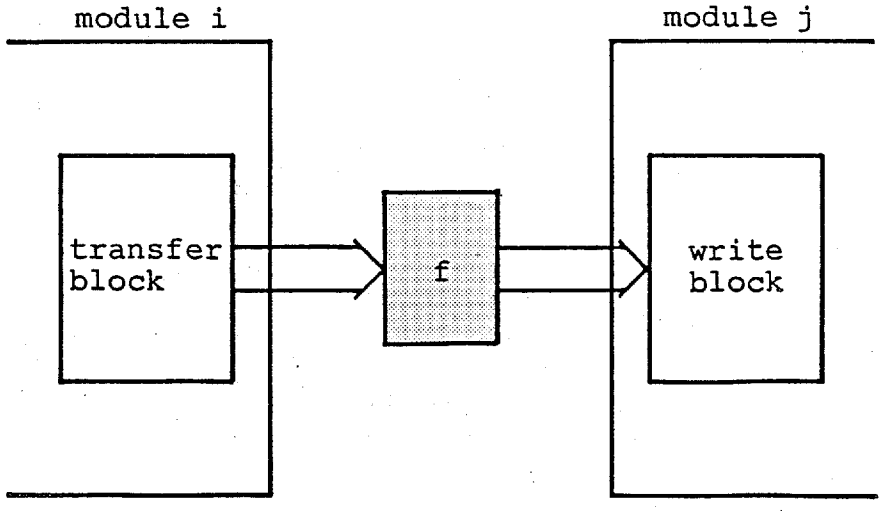


Fig. 4.13 Function **f** added through transfer between modules.

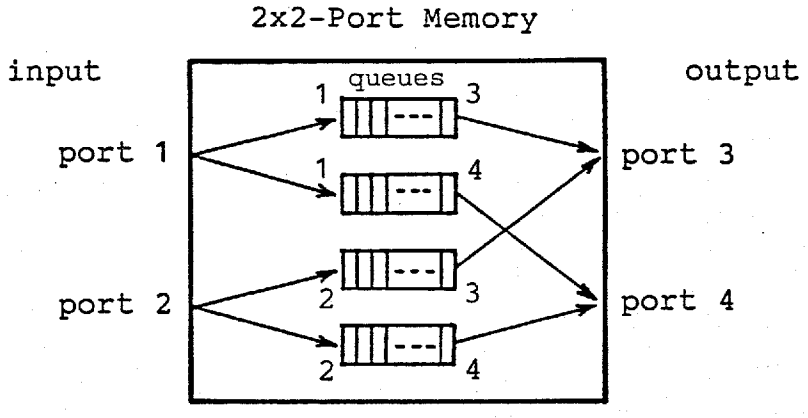


Fig. 4.14 Four-way of connections for two input and two output ports.

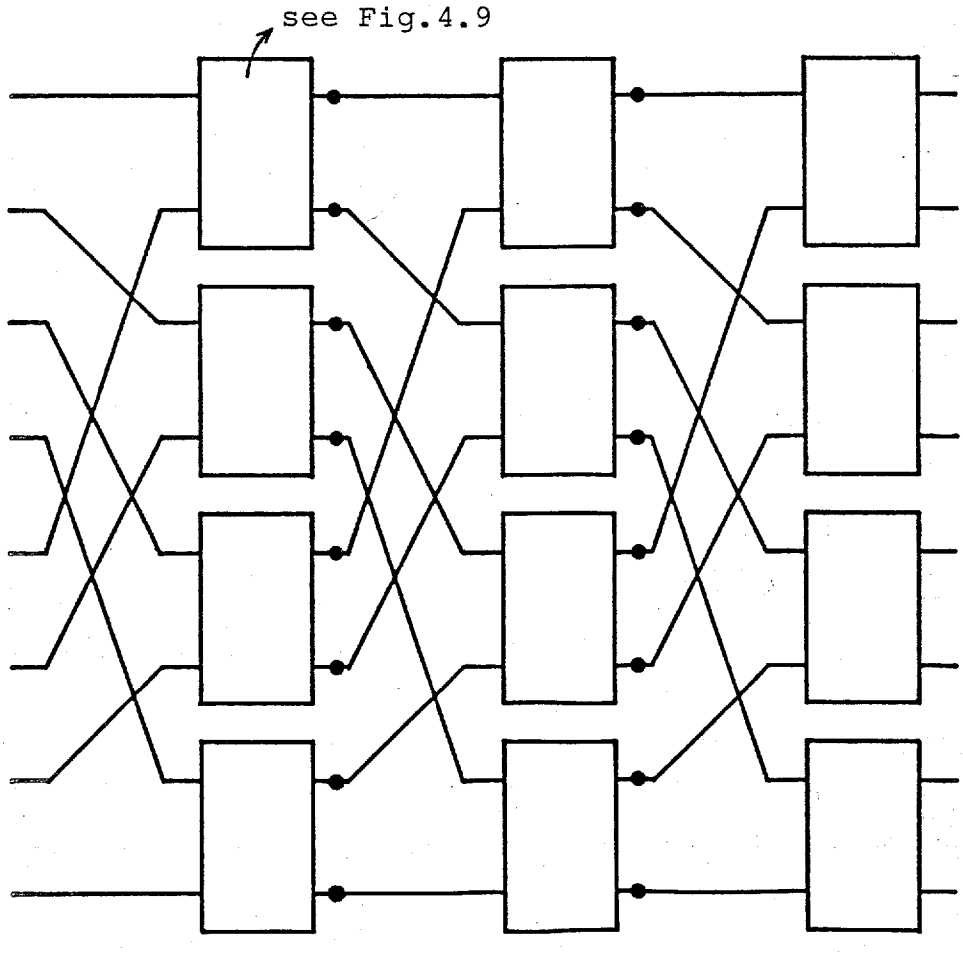


Fig. 4.15 An 8x8 shuffle-exchange COMBINET.

(A dot represents a connected transfer block with a write block.)

4.4 Performance Evaluation

Though the 2x2-port memory module has four queues described in the previous section, it is modeled by two queues and an idealized 2x2-crossbar switch for a performance analysis in this section. We assume that each queue connects the input port of the 2x2-crossbar switch to the output port of the preceding stage, i.e., it plays a role of a buffer between stages. This case has been analyzed by Dias et al. [DIAS81]. The environment of the networks and their operation are assumed mainly as follows:

(1) The performance of the networks is compared in an environment of **maximum loading**, i.e., there is a buffer at network input links which is filled by an input packet whenever it is emptied by the network. It is assumed that buffers at network output links are emptied instantaneously.

(2) All input packets are assumed to be independently and equiprobably directed to each network output link.

(3) The delay at a 2x2 switch module is modeled as consisting of two time intervals: time **t_select** for selecting a switch output link and time **t_pass** for passing the data to the selected output link [DIAS81].

For the case of 2x2-port memory module, we may assume that t_pass=0 because of the circular queues. The transfer block of a module can transfer data of a packet from the queue instantaneously, when the transferring direction of the packet is

recognized by the module and the buffer of the following stage is available. Moreover, the write block of a module can put data of a packet into the queue instantaneously, when the transfer block of the module begins to get data from the queue which is completely occupied by data.

One of the results of the analysis, i.e., the normalized throughput versus the number of stages, is shown in Fig. 4.16. The throughput of a network for a particular environment is the average number of packets put out in unit time, and the normalized throughput is the ratio of the throughput to the maximum throughput.

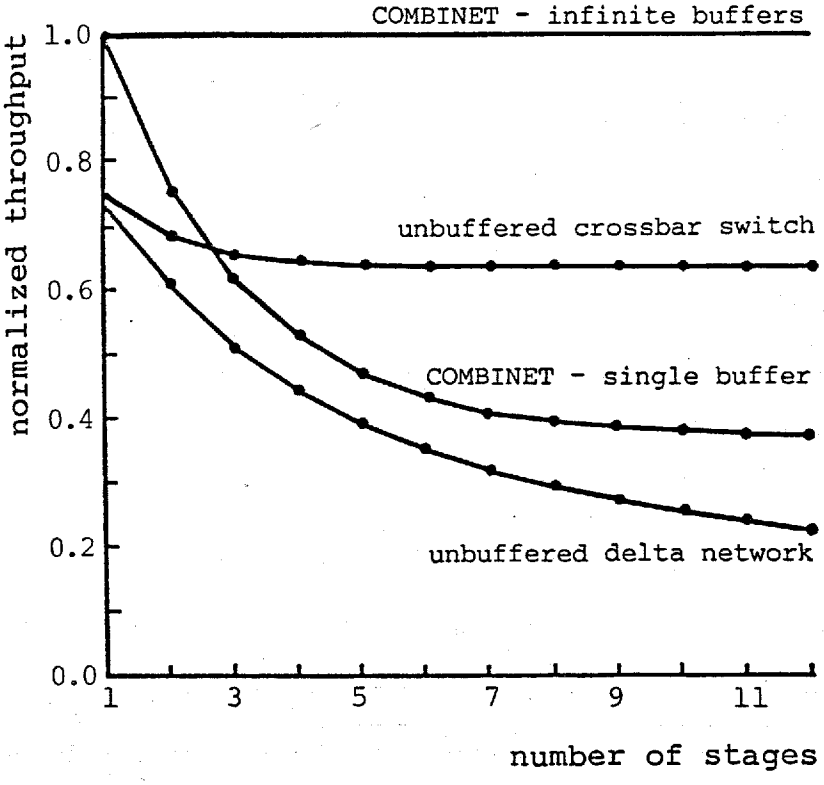


Fig. 4.16 Normalized throughput versus number of stages [DIAS81].

4.5 Summary

In this chapter we proposed the COMBINET as a new multiprocessor interconnection network, and discussed its realization using the 3D-VLSI technology.

The features of COMBINET are summarized as follows:

(1) the throughput becomes similar to that of the multistage network with infinite buffers, because the module, i.e., the 2x2-port memory works as a buffered 2x2-crossbar, where the buffer length is long enough, and
(2) the delay is constant for each stage, not depending on the buffer length, because the transfer between modules is pipelined.

The operation for data between modules in the COMBINET can be inserted, although this feature is not discussed in the chapter. This may bring varieties of functions into the massive multiprocessor architectures with the COMBINET.

The COMBINET stated here is only a prototype, i.e., constructed by the 2x2-port memory modules. For a large number of n, e.g., 512, the number of modules becomes 256x9. This number, however, is reduced to 64x3, if the module is an 8x8-port memory, although it requires the 16-layered 3-D VLSI technology. More configurations of COMBINET will be derived from many researches of the multistage networks (e.g., [SIEGEL81a] [SIEGEL81b] [CHEN81] [BARNES81]). Moreover,

other configurations than the multistage network are also realizable (e.g., the processor array [AE85c]).

Chapter 5

Conclusions

5.1 Conclusions of the Dissertation

We have presented fundamental issues for hardware design of MMS's aimed at increasing their performance. The objective systems are assumed to be designed as dedicated processors for the purpose of highly parallel processing. Moreover, the systems are required to apply a wide variety of problem. The first assumption reasonably comes from needs for supercomputing architecture realization; however, the second assumption mainly comes from fabrication costs. In reality, significant reduction of the cost in recent computer systems is due to the mass production of the same chips. Therefore, the cost of systems is closely related to the number of their production, i.e., a variety of their applications. The cost, however, has not explicitly presented in this dissertation because it is difficult to precisely predict the costs. We have implicitly referred to the system costs which can be reduced by wide applications of the systems and the construction using a number of homogeneous processing elements.

First, we have discussed about multiprocessor approaches toward MMS's. Through the design of the realistic multiprocessor system and several experiments using the system, we have pointed out the crucial issues at the design and realization

of MMS's. We have focused on two issues of them: performance prediction and evaluation of the MMS's, and interconnection between the processing elements for a realization of MMS's.

The first issue have been solved by simulation method. Our trial began with modeling a processing elements by a node which had only simple state transition with two types of control functions. The performance prediction and evaluation is obtained by the simulator based on the model. Owing to the model simplicity, the simulator realized on a UNIX system has sufficient simulation capability for the number of processors in MMS's. The performance measurements proposed in Chapter 2, i.e., interval time and response time, for designated MMS's can be interactively obtained with various information about the execution time.

For the second issue, we have proposed interconnection networks containing several switch elements using the common memory that is based on the 3-dimensional VLSI technology. The switch elements can select a processor communication path with data buffering. Since the common memory in the elements can be accessed without conflict, the interconnection networks will connect processors in MMS's at high throughput.

5.2 Future Problems

The following future problems are left unsolved in our investigation, when we realize such Massive Multiprocessor Systems.

(1) Improvement of the performance evaluation system

The simplicity of the model proposed here decreases the simulation time in the performance evaluation of MMS's. However, since it allows description inaccuracy of processor behavior, the model could not apply to detail analysis and verification of MMS's with complex behavior elements. It is expected to improve the model and the simulation system under consideration of a trade-off between model granularity and simulator consuming time.

(2) Switch element of the interconnection network

The switch element with 3-dimensional common memory has only simple data transfer function. In order to applied it to a variety of network topology, the switch element should be extended to a flexible and high performance communication element, i.e., the communication-oriented processor.

(3) Fault-tolerance

With the increase in processing elements of MMS's, the probability of existing faulty units in the system goes up increasingly. Since it is always difficult to expect a complete system that includes no faulty unit, MMS's need to equip fault-tolerant capability in levels of hardware and software.

In particular, since an interconnection network in MMS's become considerably complex, its faulty should be inspected dynamically or statically [AE85c].

(4) Software utilities (operating systems, description languages, etc.)

MMS's require system software which helps application programs efficiently utilize hardware resources, e.g., processing elements, and description languages for easy representation of highly parallel processing.

(5) Three-dimensional VLSI technology

It is required for realization of the low conflict common memory to improve and establish the 3-dimensional VLSI technology.

Acknowledgments

I would like to thank many who contributed to the development of the ideas and the systems in this dissertation, and its successful completion.

I am especially grateful to my supervisor, Professor Tadashi Ae of Hiroshima University. He supported me in all aspects of the study; he nurtured the research leading to the dissertation by a combination of encouragement, criticism, and guidance.

I am also indebted to Associate Professor Masafumi Yamashita of Hiroshima University for his careful and critical reading of the preliminary version of this dissertation. I wish to thank Professors Noriyoshi Yoshida, Tadao Ichikawa, and Kenji Onaga of Hiroshima University, who are members of my dissertation committee, for their critical comments and valuable suggestions, and to thank all professors of Graduate Course of Systems Engineering of Hiroshima University for their guidance.

Thanks are also due to many of my former and present colleagues in Computer Systems Laboratory of Hiroshima University for their helpful discussions and cooperations in this research. Messrs. Masaru Iida, Takayuki Okada, and Shigeru Morifuku assisted me in developing the UNIP system software and the application programs. Their help considerably contributed toward the experiments of UNIP. Mr. Hiroshi Matsumoto realized the high-speed simulator on the UNIX operating system. I am grateful to Mr. Minoru Etoh for his helpful discussions about the performance evaluation and his cooperation for the simulator implementation.

References

[AE82]

T.Ae and R.Aibara, "Experimentation and Analysis of Multiprocessor Systems," **Proc. IEEE Real-Time Systems Symposium**, pp.69-80, Dec. 1982.

[AE84a]

T.Ae, R.Aibara, and M.Etoh, "Design and Realization of Many Processor System using Parallel Flow Graph," **Proc. IEEE Workshop on Language for Automation**, pp.7-12, Nov. 1984.

[AE84b]

T.Ae and R.Aibara, "An Optically-connected Common Memory for Multiprocessor System," **Proc. 28th National Meeting of Information Processing Society of Japan**, 3C-7, 1984, in Japanese.

[AE85a]

T.Ae and R.Aibara, "A Realization of Parallel Processing System Based on Tree-Dimensional Integrated Circuits," **Trans. IPS Japan**, vol.26, no.6, pp.1145-1148, Nov. 1985, in Japanese.

[AE85b]

T.Ae, R.Aibara, M.Etoh, and H.Matsumoto, "A Massive Multiprocessor Simulator for Performance Evaluation," **First International Conference on Supercomputing Systems**, pp.73-82, Dec. 1985.

[AE85c]

T.Ae, R.Aibara, W.C.Cunha, and A.Fang, "On Fault Diagnosis of Tree Networks," **Proc. China 1985 International Conf. Circuits and Systems**, pp.220-223, Jun. 1985.

[AIBARA85]

R.Aibara and T.Ae, "An Implementation of Sort/search Engine

on a Multimicroprocessor," **Trans. IPS Japan**, vol.26, no.2, pp.349-355, Mar. 1985, in Japanese.

[AIBARA86]
R.Aibara, M.Etoh, H.Matsumoto, and T.Ae, "A Many Processor Simulator for Performance Evaluation," **Trans. IPS Japan**, vol.27, no.2, Feb. 1986, to appear.

[BARNES81]
G.H.Barnes and S.F.Lundstrom, "Design and Validation of a Connection Network for Many-Processor Multiprocessor Systems," **IEEE Computer**, vol.14, no.12, pp.31-41, Dec. 1981.

[BEETEM85]
J.Beetem, M.Dennean, and D.Weingarten, "The GF11 Supercomputer," **Proc. 12th International Symposium on Computer Architecture**, pp.108-115, Jun. 1985.

[BELL71]
C.G.Bell and A.Newell, **Computer Structures: Readings and Examples**, McGraw-Hill, New York, 1971.

[CHANG80]
S.S.L.Chang, "Multiple-Read Single-Write Memory and Its Applications," **IEEE Trans. Comput.**, vol.C-29, no.3, pp.689-694, Mar. 1980.

[CHEN81]
P-Y.Chen, D.H.Lawrie, D.A.Padua, and P-C.Yew, "Interconnection Networks Using Shuffles," **IEEE Computer**, vol.14, no.12, pp.55-64, Dec. 1981.

[CHOU82]
T.C.K.Chou and J.A.Abraham, "Load Balancing in Distributed Systems," **IEEE Trans. Software Eng.**, vol.SE-8, no.4, pp.401-412, Jul. 1982.

[CHRIST84]
N.H.Christ and A.E.Terrano, "A Very Fast Parallel Proces-

sor," **IEEE Trans. Comput.**, vol.C-33, no.4, pp.344-350, Apr. 1984.

[DIAS81]
D.M.Dias and J.R.Jump, "Analysis and Simulation of Buffered Delta Networks," **IEEE Trans. Comput.**, vol.C-30, no.4, pp.273-282, Apr. 1981.

[DOD80]
DoD; **Reference Manual for ADA Programming Language**, United States Department of Defence, Jul. 1980.

[DUBOIS84]
M.Dubois and F.A.Briggs, "Performance of Synchronized Iterative Processes in Multiprocessor Systems," **IEEE Trans. Software Eng.**, vol.SE-8, no.4, pp.419-431, Jul. 1984.

[ENSLOW74]
P.H.Enslow,Jr., **Multiprocessors and Parallel Processing**, John Wiley & Sons, Inc., 1974.

[FENG72]
T-y.Feng, "A Survey on Interconnection Networks," **IEEE Computer**, vol.14, no.12, pp.12-27 Dec. 1981.

[FLYNN72]
M.J.Flynn, "Some Computer Organizations and There Effectiveness," **IEEE Trans. Comput.**, vol.C-21, no.9, pp.948-960, 1972.

[GAJISKI82]
D.D.Gajiski, D.A.Padua, D.J.Kuck, and R.H.Kuhn, "A Second Opinion on Data Flow Machines and Languages," **IEEE Computer**, vol.15, no.2, pp.58-69, Feb. 1982.

[GAJISKI85]
D.D.Gajiski and J-K Peir, "Essential Issues in Multiprocessor Systems," **IEEE Computer**, vol.18, no.6, pp.9-27, Jun. 1985.

[HEIDERBERGER82]
P.Heiderberger and K.S.Trvedi, "Queueing Network Models for Parallel Processing with Asynchronous Tasks," **IEEE Trans. Comput.**, vol.C-31, no.11, pp.1099-1109, Nov. 1982.

[HOCKNEY81]
R.W.Hockney and C.R.Jesshope, **Parallel Computers**, Adam Hilger Ltd., Bristol, 1981.

[INMOS84]
Inmos, **Occam Programming Manual**, Prentice-Hall International, London, 1984.

[IRANI82]
K.B.Irani and K.W.Chen, "Minimization of Interprocessor Communication for Parallel Computation," **IEEE Trans. Comput.**, vol.C-31, no.11, pp.1067-1075, Nov. 1982.

[JAJODIA83]
S.Jajodia, J.Liu, and P.A.Ng, "A Scheme of Parallel Processing for MIMD Systems," **IEEE Trans. Software Eng.**, vol.SE-9, no.4, pp.436-445, Jul. 1983.

[JANSEN80]
P.G.Jansen and J.L.W.Kessels, "The DIMOND: A Component for the Modular Construction of Switching Networks," **IEEE Trans. Comput.**, vol.C-29, no.10, pp.884-889, Oct. 1980.

[KAWAMURA83]
S.Kawamura et al., "3-Dimensional SOI/CMOS IC's Fabricated by Beam Recrystallization," **Proc. IEEE International Electron Devices Meeting**, pp.364-367, 1983.

[KAWAMURA84]
S.Kawamura et al., "3-Dimensional Gate Array with Vertically Stacked Dual SOI/CMOS Structure Fabricated by Beam Recrystallization," **Digest of 1984 Symposium on VLSI Technology**, San Diego, pp.44-45, 1984.

[KODRES78]
U.R.Kodres, "Analysis of Real-Time Systems by Data Flow-graphs," **IEEE Trans. Software Eng.**, vol.SE-4, pp.169-178, May 1978.

[KUNG82a]
H.T.Kung, "Why Systolic Architectures?" **IEEE Computer**, vol.15, no.1, pp.37-46, Jan. 1982.

[KUNG82b]
S.Y.Kung, K.S.Arun, R.J.Gal-Ezer, and D.V.Bhaskar Rao, "Wavefront Array Processor: Language, Architecure, and Applications," **IEEE Trans. Comput.**, vol. C-31, no. 11, pp.1054-1066, Nov. 1982.

[KUNG84]
S.Y.Kung, "On Supercomputing with Systolic/Wavefront Array Processors," **Proc. the IEEE**, vol.72, no.7, pp.867-884, Jul. 1984.

[LAWRIE75]
D.H.Lawrie, "Access and Alignment of Data in an Array Processor," **IEEE Trans. Comput.**, vol.C-24, no.12, pp.1145-1155, Dec. 1975.

[MA82]
P.R.Ma, E.Y.S.Lee, and M.Tsuchiya, "A Task Allocation Model for Distributed Computing Systems," **IEEE Trans. Comput.**, vol.C-31, no.1, pp.41-47, Jan. 1982.

[MA84]
R.Perng-Yi Ma, "A Model to Solve Timing-Critical Application Problems in Distributed Computer Systems," **IEEE Computer**, vol.17, no.1, pp.62-68, Jan. 1984.

[MAY84]
D.May and R.Shepherd, "The Transputer Implementation of Occam," **Proc. International Conf. Fifth Generation Computer**

Systems 1984, edited by ICOT, pp.533-541, 1984.

[MEKLEY80]
L.J.Mekley and S.S.Yau, "Software Design Representation Using Abstract Process Networks," **IEEE Trans. Software Eng.**, vol.SE-6, no.5, pp.420-435, Sep. 1980.

[MURPHY68]
J.E.Murphy, "Resource Allocation with Interlock Detection in a Multi-Task System," **Proc. AFIPS FJCC**, vol.33, Part 2, pp.1169-1176, 1968.

[MYERS82]
G.J.Myers, **Advances in Computer Architecture, Second Edition**, John Wiley & Sons, Inc., 1982.

[OLDEHOEFT83]
R.R.Oldehoeft, "Program Graphs and Execution Behavior," **IEEE Trans. Software Eng.**, vol.SE-9, no.1, pp.103-108, Jan. 1983.

[PATERSON82]
D.A.Paterson and C.H.Sequin, "A VLSI RISC," **IEEE Computer**, vol.15, no.9, pp.8-18, Sep. 1982.

[PETERSON81]
J.L.Peterson, **Petri Net Theory and The Modeling of Systems**, Prenice-Hall Inc., N.J., 1981.

[POTTER85]
J.L.Potter (ed.), **The Massively Parallel Processor**, Research Report and Notes, Scientific Computation Series, MIT Press, 1985.

[RAMAMOORTHY80]
C.V.Ramamoorthy and G.S.Ho, "Performance Evaluation of Asynchronous Concurrent Systems using Petri Nets," **IEEE Trans. Softare Eng.**, vol.SE-6, no.5, pp.440-449, Sep. 1980.

[SIEGEL81a]

H.J.Siegel et al., "PASM: A Partitionable SIMD/MIMD System for Image Processing and Pattern Recognition," **IEEE Trans. Comput.**, vol.C-30, no.12, pp.934-947, Dec. 1981.

[SIEGEL81b]
H.J.Siegel and R.J.McMillen, "The Multistage Cube: A Versatile Interconnection Network," **IEEE Computer**, vol.14, no.12, pp.65-76, Dec. 1981.

[SPECIAL82]
Special Issue on Highly Parallel Computing, **IEEE Computer**, vol.15 no.1, Jan. 1982.

[SPECIAL83]
Special Issue on Performance Evaluation of Multiple Processor Systems, **IEEE Trans. Comput.**, vol. C-32, no.1, Jan. 1983.

[SPECIAL84]
Special Issue on Effect of Hardware-Software Interaction on System Performance, **IEEE Computer**, vol.17, no.7, Jul. 1984.

[STONE73]
H.S.Stone, "Problems of Parallel Computation," **Complexity of Sequential and Parallel Numerical Algorithms**, Academic Press, 1973.

[STONE77]
H.S.Stone, "Multiprocessor Scheduling with the Aid of Network Flow Algorithms," **IEEE Trans. Software Eng.**, vol.SE-3, no.1, pp.85-93, Jan. 1977.

[SULLIVAN77]
H.Sullivan and T.R.Bashkow, "A Large Scale, Homogeneous, Fully Distributed Parallel Machine, I," **Proc. 4th International Symposium on Computer Architecture**, pp.105-124, 1977.

[SWAN77]

R.J.Swan et al., "Cm* -A Modular Multi-Microprocessor," **Proc. AFIPS Conf.**, Nat. Comput. Conf., vol.46, pp.637-644, 1977.

[TANAKA80]
Y.Tanaka et al., "Pipeline Searching and Sorting Modules as Components of a Data Flow Database Computer," S.H.Lavington (ed.): **Information Processing 80**, pp.427-432, North-Holland Pub. Co., 1980.

[TRELEAVEN82]
P.C.Treleaven, D.R.Brownbridge, and R.P.Hopkins, "Data-Driven and Demand-Driven Computer Architecture" **ACM Computing Surveys**, vol.14, no.1, pp.93-143, May 1982.

[WESSELKAMPER82]
T.C.Wesselkamper, "Computer Program Schemata and the Processes They Generate," **IEEE Trans. Software Eng.**, vol.SE-8, no.4, pp.412-419, Jul. 1982.

[WULF81]
W.A.Wulf, R.Levin, and S.P.Harbison, **HYDRA/C.mmp, An Experimental Computer System**, McGraw-Hill, Inc., 1981.

*** End of References ***

Appendix
Syntax of the Script

A.1 Extended Series-Parallel Flow

```
script ::= main_module_declaration
         | logical_scheme
         | time_factors
         | set_pbserver
         | workload
         | set_variable
         | set_breakpoint
         | hardware_constraint
         | print_variable

main_module_declaration ::= '@' module

logical_scheme ::= module '=' sequential_module
                 | module '=' parallel_module
                 | module '=' selective_module
                 | module '=' iterative_module

sequential_module ::= module
                    | sequential_module '-' module
                    | copy '-' module
                    | sequential_module '-' copy '-' module

parallel_module ::= module '|' module
                  | parallel_module '|' module
                  | copy '-' module
                  | parallel_module '|' copy '|' module
                  | module '|' copy '|' module
```

```
selective_module ::= module '+' module
                   | selective_module '+' module
                   | copy '+' module
                   | selective_module '+' copy '+' module
                   | module '+' copy '+' module

iterative_module ::= module '*' repeat

copy ::= number
       | '(' expression ')'

repeat ::= number
         | '(' expression ')'


time_factors
    ::= '#' module '(' time ',' time ',' time ')'
      | '#' module '(' time ',' time '|' time ',' time ')'

time ::= /*empty*/
       | expression


set_observer ::= "observer" '{' path_name '}'


workload ::= "%{" interval "}%"
           | "%{" "}%"

interval ::= expression
           | interval ',' interaval
           | expression '(' interval ')'


set_variable ::= variable '=' expression
```

```
set_breakpoint
     ::= "break" '{' element "input"  '(' token ')' '}'
       | "break" '{' element "procs"  '(' token ')' '}'
       | "break" '{' element "output" '(' token ')' '}'
       | "break" '{' "at" number '}'
       | "break" '{' "interval" number '}'
       | "break" '{' "queue" number '}'
       | "break" '{' "term" number '}'
       | "break" '{' element "wait_time" '(' number ')' '}'

token ::= /* empty */
        | number


hardware_constraint ::= "exelem" '{' element_list '}'
                      | "exedge" '{' edge_list '}'

element_list ::= element
               | element_list ',' element

element ::= path_name
          | path_name '/' "fork"
          | path_name '/' "join"
          | path_name '/' "select"
          | path_name '/' "merge"
          | path_name '/' "loop"

path_name ::= '/' module
            | path_name '/' module
            | path_name '/' module '.' number

edge_list ::= edge
            | edge_list ',' edge

edge ::= element "(i)"
       | element "(o)"


print_variable ::= '?' variable
```

```
expresion ::= '(' expression ')'
            | expression '+' expression
            | expression '-' expression
            | expression '*' expression
            | expression '/' expression
            | expression '%' expression
            | '-' expression
            | variable
            | argument
            | number

module ::= alpha
         | module  alpha
         | module  numeric

variable ::= '$' alpha
           | variable  alpha
           | variable  numeric

argument ::= '$' number

number ::= numeric
         | number  numeric

alpha ::= 'a' | .. | 'z' | 'A' | .. | 'Z'

numeric ::= '0' | .. | '9'
```

A.2 Two-Dimensional Array

```
script ::= variable_def  array_size_def  cell_def
           default_def  workload


variable_def ::= /* empty */
               | "VAR" '{' variable_list '}'

variable_list ::= variable_assign
                | variable_list  variable_assign

variable_assign ::= variable ":=" expression


array_size_def ::= "ARRAY" '(' expression ',' expression ')'


cell_def ::= /* empty */
           | cond_list

cond_list ::= cond_statement
            | cond_list  cond_statement

cond_statement
       ::= "CONDITION" '(' cond_expr ')' '{' time_def '}'
         | "CONDITION" '(' cond_expr ')' '{' cond_list '}'

cond_expr ::= comp
            | cond_expr "AND" cond_expr
            | cond_expr "OR"  cond_expr
            | "NOT" cond_expr
            | '(' cond_expr ')'
```

```
comp ::= expr '<'  expr
       | expr '<=' expr
       | expr '>'  expr
       | expr '>=' expr
       | expr '='  expr
       | expr '<>' expr

time_def ::= /* empty */
           | time_def  set_time
           | time_def  variable_assign

set_time ::= "PROC"  ":=" expr
           | "UP"    ":=" expr
           | "DOWN"  ":=" expr
           | "LEFT"  ":=" expr
           | "RIGHT" ":=" expr


default_def ::= "DEFAULT" '{' time_def '}'


workload ::= source_list

source_list ::= source_statement
              | source_list  source_statement

source_statement
          ::= "SOURCE" '('  "UP"   ')' '{' source_cond '}'
            | "SOURCE" '(' "LEFT" ')' '{' source_cond '}'

source_cond
   ::= "CONDITION" '(' cond_expr ')' '{' input_interval '}'

input_interval ::= "INTERVAL" '(' interval ')'

interval ::= expr
           | interval ',' interval
           | expr '(' interval ')'
```

```
expr ::= expr '+' expr
       | expr '-' expr
       | expr '*' expr
       | expr '/' expr
       | expr '%' expr
       | expr '!'
       | expr '^' expr
       | '-' expr
       | '(' expr ')'
       | number
       | variable
       | argument
       | X_parameter
       | Y_parameter

X_parameter ::= 'X'
Y_parameter ::= 'Y'

variable ::= 'a' | .. | 'w' | 'z' | 'A' | .. | 'W' | 'Z'

number ::= numeric
         | number  numeric

argument ::= '$' number

numeric ::= '0' | .. | '9'
```