THE DEVELOPMENT OF MODELS FOR COMPUTER SIMULATION WITH DETAILED APPLICATION TO A CDC 6400 SYSTEM

by

William P. Beaumont,
B.Sc., Dip.Comp.Sc.

Department of Computing Science,

University of Adelaide

Thesis submitted for the degree of Doctor of Philosophy

24th September, 1975.

SUMMARY

This thesis describes research aimed at the development of an accurate, validated simulation model of the Control Data 6400 computer under the control of the Scope 3.2 operating system. In the course of the study, a number of new approaches to the development of such models is investigated.

The thesis starts with a discussion of the approaches that other writers have taken to the study of computer systems, and outlines the problems they have encountered. In this study, particular attention is given to three of these : level of detail, validation, and obtaining information about the system. This discussion is followed by a description of the computer and operating system being studied.

Discussion of the simulation itself starts with an overview, which is followed by a detailed description of the research undertaken. A very detailed, instruction-level simulator is used to perform a quantitative analysis of the important system software. Description of this simulator and its application is followed by a discussion of the peripheral equipment in the system, which includes a series of experiments to measure the operation times of critical devices.

The information obtained from the detailed simulation and the device measurement is combined in an event-based simulation model. This model is validated using deterministic techniques, which require the avoidance of all stochastic effects in both the model and the real system. The

results of the validation exceed expectations and justify the considerable effort required to obtain the information on which the model is based. A number of demonstrations of possible applications of the model are presented.

The thesis concludes with a reivew of the aims and achievements of the study, and an exploration of the possibilities for future research using the simulator.

CONTENTS

This thesis contains no material that has been accepted for the award of any other degree or diploma in any university, and to the best of my knowledge and belief, contains no material previously published or written by another person, except where due reference is made in the text of the thesis

(W. P. BEAUMONT)

24th September, 1975.

ACKNOWLEDGEMENTS

I am greatly indebted to my supervisor, Dr. J. L. C. Macaskill, of the Department of Computing Science, University of Adelaide, for his invaluable guidance and encouragement throughout the course of this research.

I would also like to thank Dr. I. N. Capon, Director of the University Computing Centre, and his staff for their patient co-operation and assistance with the practical aspects of the research.

CHAPTER 1

INTRODUCTION

This thesis describes research aimed at the development of a simulation model of the Control Data 6400 computer running under the Scope 3.2 operating system. In the course of this development, a number of new approaches to the development of such a model was investigated. Particular emphasis was placed on validation of the model, and the value of simulation as a means of studying the behaviour of a computer system was demonstrated.

1.1 FACTORS AFFECTING THE STUDY OF COMPUTER SYSTEMS

A large number of factors influence the development of a particular study of a computer system, and this has led to a wide variety of techniques being used by those who carry out such studies. These factors, and the methods to which they have lead, are described under the various sub-headings below, so that the place of the present simulation in the range of such studies can be readily seen.

1.1.1 Purposes of Computer Studies

The techniques used for studying a computer system will obviously depend on the purposes of the study. The common applications of computer studies are sometimes compared and classified in the literature (2, 8, 41, 47). There are generally considered to be three areas of importance, which will be called

performance measurement, performance prediction, and performance analysis.

In performance measurement, some estimate of the efficiency of the system is the final result required. Measures for efficiency include utilization of resources, throughput of jobs, etc. This type of study often assists in the selection of a computer system (16).

Performance prediction requires an estimate of how the system will behave under different conditions. This is required, for example, when a new operating system or computer configuration is designed, and evaluation of the system is required before it exists in real life (25). It is also important when changes to an existing system (or its workload) are contemplated, so that the effects of the changes can be investigated before they are introduced (55). A particular form of this application is optimization of performance, where prediction is used to determine the best of several possible changes to a system (4, 28, 30).

Analysis of performance can increase knowledge and understanding of the mechanisms at work within the operating system. These are not always obvious form a theoretical study of the operating system. The interaction of many processes, such as those that usually exist in a complex operating system, can produce unexpected results. While performance measurement can

only detect such results, performance analysis can also explain them. Performance analysis is mentioned by Naylor and Finger (47), who call it "descriptive analysis" and is discussed by Bell (2) as "diagnostic evaluation".

The applications towards which the present study is directed include both performance prediction and performance analysis, with a bias towards the latter. The reasons for the bias is that, while performance evaluation and prediction are of greater immediate value, analysis produces more (and more meaningful) information, and would therefore be of greater use in research.

1.1.2 Methods Available

Another factor influencing the method chosen to study a computer system is the range of such methods that are available. Several techniques are in common use, each with advantages and disadvantages in particular applications.

Hardware monitors (58, 63, 68) provide one such technique. There are simple circuits (counters, timers, etc.) wired into appropriate places in the system. They accumulate data which are usually recorded in machine-readable form for later analysis. Hardware monitors do not affect the operation of the system in any way, but they require physical modifications to its hardware, and can be expensive.

A more flexible approach is to use software monitors (16, 28, 38), which are modifications to the programs in the operating system (or new programs added to it) so that the system provides information about itself. This method has the disadvantage that it alters the system being measured. If the information collected is very detailed, the system can be appreciably slowed.

Benchmarks (34), kernels (64) and synthetic programs (7) provide a measuring technique based on a standard program which is run on the computer being evaluated. These methods, described in detail by Lucas (41), are often used in selecting computer systems. Their scope for predicting and analysing performance is limited.

Analytic models of computer systems usually envisage the system as a Markov chain (23, 61) or as a set of queues (60). An entire computer system is usually too complex to be completely modelled in this way. Analytic models of subsystems, for example paging systems (20) and queues (65), are more common.

Simulation involves the construction of a logical model of the system, usually realized in the form of a computer program. It has the advantage that, once the simulator is constructed, the study can be performed without reference to the real system, so that it is particularly suited to performance prediction.

Comparisons of the above techniques have appeared regularly in the literature (8, 19, 27. 41). The authors who include simulation in such comparisons (8, 41) generally conclude that it is the most effective and flexible approach, but immediately warn that it is beset with difficulties. The difficulties of simulation are discussed below (1.2). Calingaert (8) believes that these difficulties are sufficient to prejudice the effectiveness of simulation as a measuring technique, and Lucas (41) suggests that, while it is the best method, simulation is justified only for large-scale performance predictions. On the other hand, Nielsen (50) and Bell (2) while realizing the difficulties involved, regard the effort required as justifiable.

The present study involves both performance prediction and performance analysis. Indeed, adequate performance analysis often includes predictive experiments to examine aspects of the system under extreme conditions. It is therefore considered that simulation, despite its difficulties, is the most appropriate method for this study.

1.1.3 Historical Influences

Simulation models of computers have existed almost as long as computers themselves. It is inevitable, therefore, that past techniques and ideas have a strong influence on the methods used in present simulation studies.

The earliest computer simulation models involved only hardware (39, 62) and were used mainly to design processing units. This application has gained wide acceptance and is still in use today (22).

Nielsen (50) gives a comprehensive account of how the need to simulate software increased with the increasing complexity of operating systems. The simplest models to take account of software are those of sequential batch processing systems (18, 32, 70). In these models, however, software modelling amounts to little more than queue management and estimation of compilation and execution times. With the more complex systems of the third generation, more attention to software is required, as shown by Fox and Kesler (25) and Nielsen (50). Unfortunately, simulation of software is difficult, and the influence of previous work, with its emphasis on hardware, is such that software simulation has tended to fall behind the development of new systems.

1.1.4 Generality

Another factor influencing the design of a computer simulation is the generality required of it. Many authors (25, 29, 31, 41) regard the ideal simulation model as one that can simulate any system, once it is fed sufficient information. The development of such a model is beyond present-day technology (41), but serious attempts in this direction have been made. The SCERT

program (29) is an example. Programs designed along these lines are generally based on empirical data about the systems being studied, and there is little simulation of the processes involved.

Simulation of the processes within an operating system is generally done using a different technique in the form of an event-based simulation (6, 35, 36, 40, 42, 43, 51, 55, 70), and is nearly always restricted to simulation of a particular system. The event-based simulations sacrifice generality for greater detail and accuracy.

For the purposes of the present study, it was thought that the additional detail gained from an event-based simulation more than made up for the resulting loss of generality. In any case, generality was not an important consideration because the study is aimed at analysis of a particular system, and not a widely varying range of systems.

1.1.5 Types of Systems

The evolution of simulation techniques with the development of more sophisticated operating systems has already been mentioned (1.1.3). There is a great variety of computer systems in use today, and the technique used to simulate them naturally depend on the type of system involved (1, 50).

In a system dominated by interactive users (4, 26, 49, 53), processing depends largely on the response times of users, the sort of requests they make, and the policy used by the system to share resources among them. In a sequential batch processing system, the off-line spooling of jobs is important, so the activities of human operators must be simulated carefully, as in the studies by Hutchinson (32) and Day and Watmough (18). In a multiprogramming system dominated by batch jobs (35, 40), such as the one involved in this study, the dependence on human activity is almost eliminated, and the system is free to schedule the available jobs as it likes. This last type of system is fundamentally different from the others, therefore, in that it is much less dependent on stochastic processes.

1.2 THE PROBLEMS OF SIMULATION

It was stated above (1.1.2) that many authors (2, 8, 31, 41, 50) stress the difficulties of performing a computer simulation effectively. Since simulation was selected as the most suitable method for effecting this study, these problems had to be investigated, and either overcome or avoided. Investigation of these problems was, in fact, one of the principal aims of the development of the simulator, and is discussed further below (1.3.2).

1.2.1 Level of Detail

The problem of choosing the level of detail at which a simulation should take place is often discussed in reviews of simulation (2, 8, 41, 47), but it is not often mentioned explicitly in reports of individual studies. The decision must be made at an early stage in the development of any model. Simulations described in the literature range from very detailed models of the logical elements within processors (22) to relatively coarse models where many assumptions are made about the system. Nielsen (51) and Noe and Nutt (55) discuss the problem, describing in detail the factors they considered in deciding whether or not to simulate a particular process.

The problem is the trade-off between increased accuracy (with more detail) and speed and simplicity (with less detail). Deciding on an optimal point in this trade-off is an important and difficult step in the design of a model.

1.2.2 The Problem of Validation

Most reviewers of simulation techniques (2, 8, 41, 54) stress the importance of validating simulation models. In individual studies, the stress placed on validation varies greatly. Some (33, 55) give the problem a great deal of attention, but it is often omitted completely, as pointed out by Noe and Nutt (55).

Validation is the process of proving experimentally that a simulation model accurately reflects the corresponding real system. The degree of accuracy required, and the range of conditions over which it is tested depend on the purposes and design of the model. When validation is carried out effectively, it is usually a major part of the simulation project.

A few discussions of the philosophy of validation have appeared (47, 69). Discussion of techniques is usually confined to statistical methods (9, 24, 47, 46), where stochastic time series from runs with the real and simulated systems are compared to determine whether they represent similar systems. Because of the stochastic variation in the series, exact agreement between the two systems cannot be expected, and statistical techniques must be used to obtain a confidence estimate of whether the two systems are the same. Naylor (47) describes eight such techniques. Little has been published about deterministic validation, probably because it is conceptually simple (a direct comparison of two time series) and difficult to realize in practice (requiring that both real and simulated systems be free of random effects). Hutchinson and Maguire (33) start their validation by what looks like a deterministic procedure.

1.2.3 Obtaining Information about the System

Another major problem that must be resolved before a model can be constructed is the accumulation of information about the

system. Information is needed about hardware, software and job mix. Only a few authors (51, 55) have given their sources for such information.

Information about hardware is usually obtained from the manufacturer. Manufacturer's information about processors (instruction execution times, etc.) can be accepted with confidence because these devices rely on strictly timed pulse circuitry. Data concerning peripheral devices, however, are generally supplied as average or ideal figures, from which individual units could possibly differ. There does not seem to have been any investigation of this matter by means of physical measurement of peripheral devices, as was done in the present study.

It was explained above (1.2.3 and 1.2.5) that in complex operating systems, information about software is, if anything, more important than information about hardware. Software information has been obtained by consulting systems personnel (51), by analysing accounting files (55) and by use of software monitors (48). The method that gives the most detailed results, however, is tracing of the software in execution, instruction by instruction, as described by Deniston (19) and McKinley (44). This last technique is the one used in the present study.

Information about the jobs run on the system is not essential for constructing the model, but it is essential if the model is to

be of any use. Again, a number of different methods have been used, including the monitoring of users' jobs as they run (48), and analysis of accounting files (57). When the characteristics of the job mix have been measured, there still remains the problem of generating job streams that have these characteristics, so that the model can be set to work on them (36, 42, 51).

1.2.4 Language

A major practical difficulty in writing a simulation program is that the commonly used programming languages do not cater for the special needs of simulation. This has led to the creation of a large number of minor languages and packages (21, 52, 59, 70) specifically for the simulation of computer systems. General purpose simulation languages, such as Simula (17) and Simscript (45) are also used. Despite the availability of special purpose languages and packages, many authors use Fortran, for reasons discussed by Nielsen (49). Packages of subroutines that give Fortran greater facility for simulation have long been popular (11, 37). Choosing between the many languages and packages available is often difficult, and is discussed frequently (31, 49, 51).

1.3 INTRODUCTION TO THE PRESENT STUDY

As indicated above (1.1), the course of development of the simulation model described here was influenced by many factors. Chief amongst these were the nature of the system being studied and the aims

of the investigation.

1.3.1 The System Being Studied

The system being simulated is that of the Computing Centre of the University of Adelaide. The work undertaken by the Centre is typical of that done by most university installations. There are numerous small undergraduate jobs mixed with a significant number of larger jobs submitted by advanced students and staff. Still larger jobs are provided by the university administration, the library, and systems programming staff. The Centre also acts as a service bureau for a number of external users.

The operating system being simulated is Scope 3.2, with extensive additions from Scope 3.3. This is the most sophisticated 6400 operating system to be designed solely for batch jobs. The system is a multiprogramming batch system, with no facility for remote time-sharing. As stated above (1.1.5), this type of system has special characteristics in that it is almost entirely independent of human activity.

Since the present study started, there have been some extensive modifications to the hardware and software of the University of Adelaide installation. The future incorporation of these changes into the simulation is discussed in Chapter 10.

1.3.2 Aims of the Study

The primary aim of this research is was the investigation of methods to develop a well validated simulation model of a complex, multi-programmed computer operating system, specifically the Scope 3.2 system. The requirement for accuracy resulted in two secondary aims being defined for the development of the model.

Firstly, the model was to contain no assumed information about the system. Information about the hardware and software is never taken at face value. The response times of the one peripheral device that proved critical (a disc) were measured explicitly, and software was investigated by using an instruction trace. In the final model, there is not a single quantity resulting from assumption or intuition. All the information has been taken, as it were, from first principles.

Secondly, the model was to be validated in as exact and detailed a way as possible. Most of the validation is done deterministically, so that recourse to statistical methods is not necessary.

The realizations of these two aims are closely related. Because the quantities required for the model are measured rather than estimated, there is little call for stochastic simulation. The operations of most processes are known exactly. Hence the validation can be performed exactly, using deterministic techniques.

1.3.3 New Features of the Study

A number of procedures used in the course of this research do not seem to have been described before. Of particular interest are the approaches made to some of the problems described in 1.2 (level of detail, validation, and information about the system).

The usual approach to levels of detail is firstly to simulate at some level, and then to validate the simulation. If the model proves to be inadequate, the offending process is identified and simulated with more detail. The direction of development is thus from less detailed to more detailed models. The approach taken here is in the reverse direction. Simulation starts at a very detailed level, and detail is removed only when its removal will not affect the accuracy of the model. This approach is necessary to incorporate the exactly known information about the system, which is at a very detailed level (instruction execution times, etc.).

The validation is almost entirely deterministic. As explained above (1.2.3), deterministic validation requires that random effects be removed from both real and simulated systems. This, firstly, requires that the system be analysed in sufficient detail for the stochastic processes to be identified. In the Scope system there are few such processes, and their elimination was found to be possible to a surprisingly large extent. This reduction of an operating system to a completely deterministic set of processes seems to be without precedent.

Gathering information about the system also involved new techniques. Software is used to measure the response times of peripheral equipment, for which the manufacturer's data are usually accepted unquestioned in other studies. The methods used here combined the techniques of programming and systems engineering.

1.3.4 Synopsis of Subsequent Chapters

The remainder of the thesis is largely devoted to describing the methods used to effect the ideas introduced above (1.3.2 and 1.3.3). In preparation for this, Chapter 2 describes in some detail the CDC 6400 computer and the Scope 3.2 operating system. Chapter 2 is largely independent of the other chapters, and is provided for those who may not be familiar with the details of the system. Chapter 3 gives an overview of the development of the simulator, and introduces the practical implementation of the ideas discussed above.

The detailed description of the research starts with Chapter 4, which describes a detailed (instruction based) simulation model. Chapter 5 describes how this model was validated, and how its instruction traces were used to analyse the software of the Scope system. Chapter 6 presents the methods used to measure the response times of peripheral equipment. These response times are combined with the information of Chapter 5 (about software) to construct a second, less detailed model, which is the main

product of the study. The construction of this model is described in Chapter 7, and Chapter 8 discusses its validation. This completes the development of the simulation study.

Chapter 9 presents some informal demonstrations of the possible applications of the model. Finally, in Chapter 10, the problems and aims introduced above are reviewed in the light of the completion of the study, and proposals are made for future research.

CHAPTER 2

DESCRIPTION OF THE SYSTEM BEING SIMULATED

In describing the features of the CDC 6400 under the Scope 3.2 operating system it will be useful to consider two areas separately. Firstly the processors and other hardware will be described and secondly the use to which this hardware is put by Scope is discussed. This approach provides a convenient separation of available features into those which are determined by the hardware and those which are provided by the operating system. This distinction shows the high degree to which the behaviour of this type of computer depends on the characteristics of its operating system. It also clarifies the philosophy behind Scope, and shows how it exploits and is constrained by the hardware for which it was designed.

Throughout the following discussion "Scope" will be used to mean "Scope 3.2". All numbers quoted will be in decimal, unless otherwise indicated.

2.1 CDC 6400 HARDWARE

A detailed description of the processors and other hardware of the 6400 would be out of place here. Control Data (13) provide adequate descriptions of the instructions sets of the processors and Thornton (67) discusses in detail their design and the reasoning behind it. This section is simply a summary of the main components of the computer,

in sufficient detail for an understanding of the simulation described later. The various components to be discussed are shown in Figure 1.

2.1.1 The Central Processor

The 6400 component of most interest to the ordinary user is the central processor (CPU). This unit performs rapid logical and arithmetic computations and provides the programmer with an instruction set (of 74 instructions) in keeping with these capabilities. The CPU executes programs from central memory, to which it has restricted read/write access. The computations are performed in a set of 24 registers. There are further seven registers, not directly accessible to the programmer, which hold such things as the base address and upper bound of memory (which define the area of central memory available to the CPU) and the address of the current instruction. Except for this last quantity, which necessarily changes during execution of a program, the CPU has no power to alter any of its special registers. It has no ability to communicate with input output devices or to control other components of the system. It is purely a computational unit. Of course, CPU programs will need to initiate control of input output processing, and methods for doing this must be included in the operating system.

2.1.2 Central Memory

Central memory is the main fast-access store for the 6400 system. It comprises a large number of 60-bit words (in the

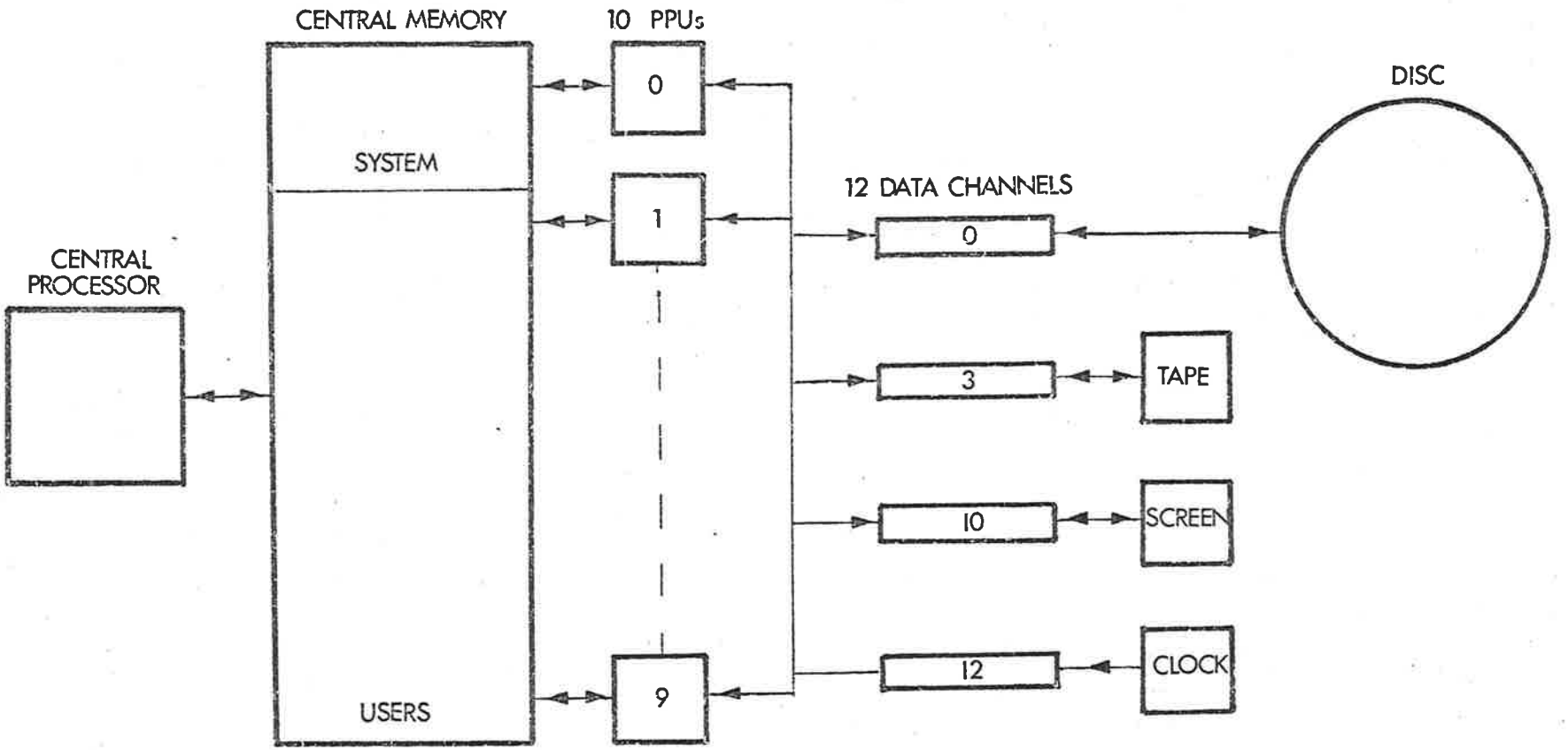


Figure 1 : The Structure of the CDC 6400 Computer, showing information paths.

installation being simulated there were 65536), with an access time of one microsecond. The organization of the CPU registers, which define a base and bound for central memory, requires that central memory be allocated to a running program in a contiguous block.

2.1.3 Peripheral and Control Processors

The ten peripheral and control processors (PPUs) are the real controllers of the system. Each has its own memory of 4096 12-bit words, which holds the program that the PPU executes. A PPU has a repertoire of 62 instructions, designed not for arithmetic computation but to give wide-reaching facilities for communicating with and controlling other parts of the system. A peripheral processor can read or write in any word of central memory. It can interrupt the CPU and change any of its special registers. Each PPU can send or receive data along data channels, and can control any input output equipment connected to such channels. However, a PPU has no access to the memory of another PPU, nor can it directly affect the operation of another PPU in any way.

Note that although the ten PPUs can be regarded as separate entities for all practical purposes, they in fact share one processing unit. Each PPU has access to the processing unit for 100 nanoseconds in each microsecond. The remaining time (when the other PPUs have the processor) is used to complete its memory cycle. This means that only one processing unit is necessary.

Nevertheless, the PPUs are completely autonomous processors so far as their interaction with the system is concerned.

The complete independence of a PPU means that it is impossible in practice to investigate what it is doing. It cannot be stopped ; neither its memory nor its registers can be inspected. This raises problems in measuring the performance of PPU programs in the system. Methods used to overcome these difficulties are discussed in the next chapter.

2.1.4 Data Channels

The main purpose of the data channels, of which there are twelve on the simulated installation, is to provide communication between the PPUs and the input-output equipment they control. Either data or control codes may be sent to the equipment by a PPU, and either data or status codes may be returned. There is no provision for a channel to interrupt a PPU, although in some circumstances (for example, an attempt to read data from an empty channel) a PPU can be temporarily stopped by the channel.

Alternatively, two PPUs may use a data channel as a communications link between themselves, without any input-output equipment being involved.

2.1.5 The Clock

One of the data channels gives access to a clock, which counts memory cycles (microseconds) in a 12-bit register. The value of this register can be read at any time by any PPU. Note that, because the register is only 12-bits long, the clock operates in cycles of 4096 microseconds. This clock was used extensively in the validation experiments described in later chapters.

2.1.6 Input-Output Equipment

A wide variety of input-output devices may be attached to the data channels, including adaptors for communication with other computers and remote terminals. More than one device may be connected to the same channel. Each device has a set of function codes that are used by the PPUs to control its operations. In effect, these devices can be regarded as processors that execute programs, not read from a memory, but sent down a channel by the controlling PPU. The devices of interest in the simulation model are card readers, line printers, magnetic tape units, the operators' display console and, most particularly, discs.

In view of its considerable importance in the construction of the simulation model, one device, the 6603-11 disc, deserves special attention. This disc is the main secondary store (about 75 million characters) for the system under study, and it is accessed constantly by frequently used parts of the operating

system. The storage medium is a set of 14 discs, giving 24 recording surfaces. This is shown in Figure 2. The controlling program may select either the left-hand bank of 12 surfaces or the right-hand bank. Reading or writing of 12-bit words is done in parallel on the 12 surfaces of the chosen bank. Each surface is scanned by four heads, of which the controlling program selects one. There is thus a choice of a total of eight head groups. The heads can be moved radially across the disc surface to one of 128 tracks. Data on a track is organized into sectors of 324 12-bit words each. Selecting a data item on the disc thus involves three operations : positioning the heads over the required track, switching to the appropriate head group, and waiting for the wanted sector to come under the heads. The detailed operation of this disc was found to be one of the most critical aspects of the simulation.

2.1.7 Communication and Control Paths

The Scope operating system depends very largely on communication between processors. It is therefore of interest to consider the forms of communication permitted by the hardware between the various types of processor. This is summarized in Figure 3. The CPU can communicate with each other either through central memory or along a data channel, and they can communicate with an input-output device along the channel(s) to which that device is connected. Direct communication between the CPU and

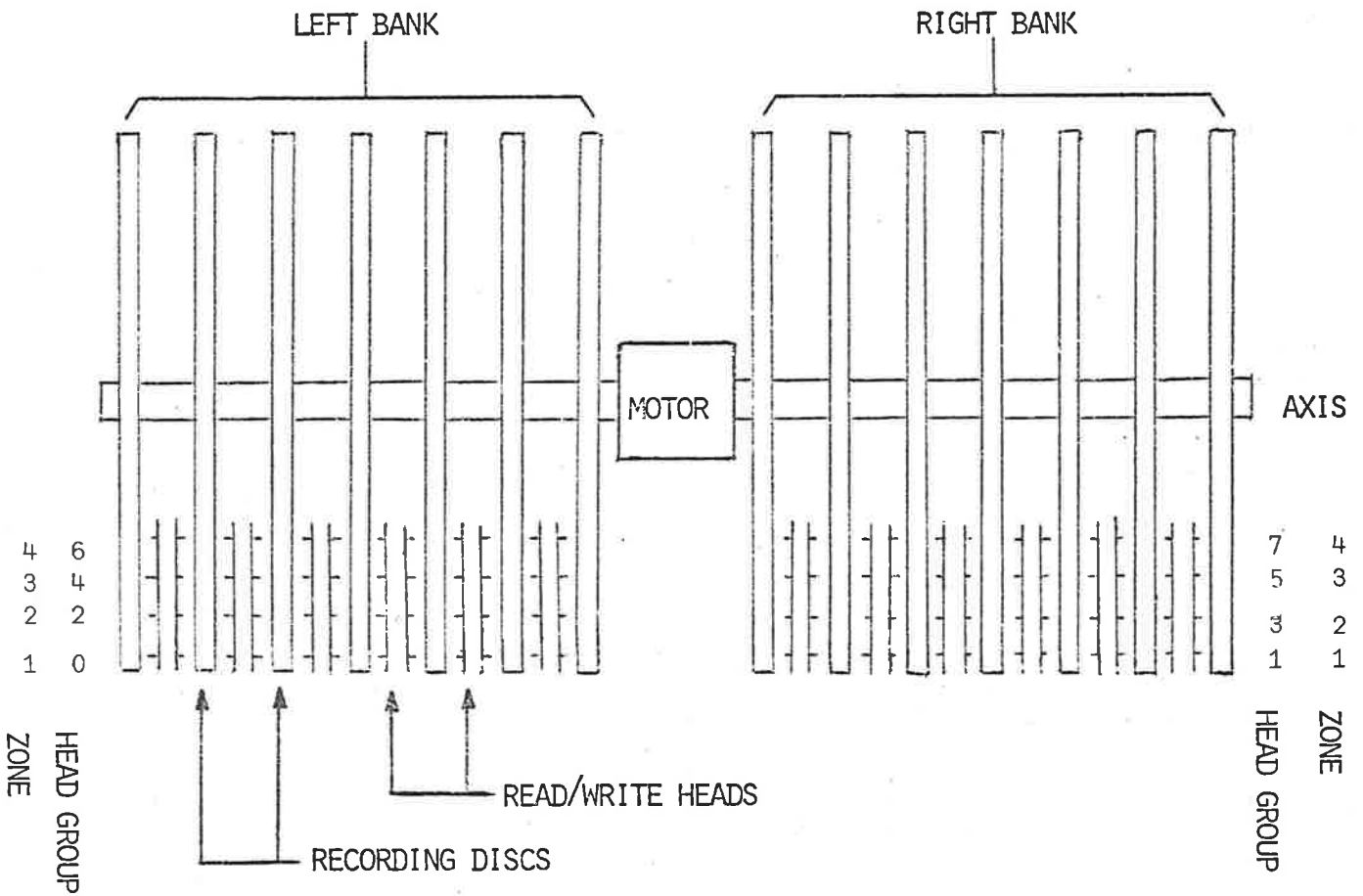


Figure 2 : The structure of the 6603 disc

Destination of Information	Source of Information: CPU	PPU	Peripheral
CPU	-	central memory	-
PPU	central memory	central memory or a channel	channel
Peripheral	-	channel	-

Controlled Device	Controlling Device: CPU	PPU	Peripheral
CPU	-	interrupt	-
PPU	-	-	-
Peripheral	-	channel	-

Figure 3 : The Paths of Communication (above) and Control (below) between the components of the 6400.

the input-output devices is not possible.

The paths along which control can flow are also of interest, and are fairly simple. Any PPU can control the CPU and any input-output device, but cannot control another PPU. The CPU and the input-output devices have no power to control anything.

2.2 THE CDC 6400 UNDER SCOPE 3.2

The Scope 3.2 operating system is designed to be embedded in the 6400 computer to make it function as an efficient multi-programming system. The whole of Scope is a vast topic and the only aspects of it to be discussed here are its overall purposes and operation, and features which proved to be important in constructing the simulation model. More detailed descriptions can be found in various publications by Control Data (14, 15).

2.2.1 The Purposes of Scope

There are many ways in which the hardware facilities described in the last section can be used to support batch jobs. A very early and simple approach, Sipros, was described by Clayton, Dorf and Fagen (10). The differences between this system and the sophisticated facilities offered by the latest versions of Scope illustrate how much the behaviour of a computer can be affected by its operating system, without any changes in its hardware configuration. For example, Sipros processed only one job at a time. The user had a choice of three languages (CPU and

PPU assembly language and Fortran) and the operator had available three displays to show what the system did. Scope 3.2 processes up to seven jobs at once, with many languages available (about 12 at the installation under study) and about twenty displays.

The basic aim of Scope 3.2 is the handling of a large number of batch jobs, several of which may be in execution at the same time. The various resources required by the jobs must be allocated in a reasonably efficient way, and according to such rules that deadlocks are prevented. A wide variety of library programs (compilers, file-handling utilities, etc.) is made available to the jobs. Data areas belonging to one job must be protected from both reading and writing by any of the others. Information for accounting purposes must be generated and collected for later analysis.

2.2.2 Distribution of Tasks

The peripheral processors, with their considerable powers of access and control, are ideally suited to the tasks required by the operating system. By far the greater part of Scope processing is handled by a relatively small number of PPU programs. These control the flow of jobs from input to output, and supervise the allocation of resources. Tasks which are closer to the user, such as compilation and execution of his own programs, are left to the central processor. Users are not permitted to program a PPU, which can only be done by altering the system library, but

they can call certain PPU programs when required for input-output and other system operations. It is this feature that ensures the security of Scope against misuse by programmers.

PPU programs are given 3-character mnemonic names (e.g. DSD, the display driver) which will be introduced as the discussion proceeds.

2.2.3 The Monitor and the Display Driver

The most important effect of Scope on the raw hardware configuration is to enable one PPU to supervise the others. This is achieved by software ; the supervising PPU executes a monitor program and the others must request its permission before doing anything that could cause resource conflicts. This technique prevents such situations as two PPUs trying to control the same input-output device at the same time, or simultaneously writing into an area of memory to which each thinks it has exclusive access. The monitor program is the only one in the Scope 3.2 system that interrupts the CPU or alters any of its special registers. This one monitor program can therefore control the allocation of all resources, deciding between conflicting programs according to the rules of the system.

Another important PPU program is the display driver, DSD, which controls the operators' display console and relays commands entered by operators to the appropriate parts of the system.

Both the monitor and DSD are loaded when the system is initialized, and remain in execution as long as the system is running. The remaining eight PPUs become resources to be allocated by the monitor as required.

2.2.4 Pool PPUs

The eight allocatable or pool PPUs execute a program called PP resident, which continually inspects a word of central memory peculiar to each PPU and called its input register. When the monitor wants a PPU to execute a particular program, it writes the program name in the corresponding input register. PP resident in the appropriate PPU detects this, and searches the library directory (stored in central memory) for the required program. The library directory holds pointers to where the program itself is stored (either in central memory or on the disc) and PP resident can thus read the program into its PPU and start executing it. When the program finishes, it jumps back to PP resident to await another request from the monitor.

In the course of execution, a PPU program will need the attention of the monitor, and this is obtained by the PPU writing an appropriate request code in its output register (another word of central memory). The monitor periodically inspects each output register looking for these requests, processing them and then clearing them. The requesting PPU, sensing that its output register has been cleared, knows that its request has been dealt

with. Monitor requests are used for the allocation of such resources as the CPU, another PPU, an area of central memory, or a channel to the job associated with the requesting PPU. Even if it requires none of these, the PPU program must at least tell the monitor that it has finished and that the PPU is idle once more. These communications between the pool PPUs and the monitor are the heart of Scope 3.2. They played a crucial role in the validation of the simulation model described later.

2.2.5 System Tables

A considerable area of central memory (typically about 10000 words) is used by Scope for system tables. These include the library directory, the PPU-monitor communications area, status information for the resources of the system, and parameters for each of the jobs in the system, whether waiting to execute, executing or completed and waiting to print. This area is never assigned to a user program, and a user can never directly access it. PPU programs frequently refer to these tables and sometimes alter them. Before an alteration is made, permission must usually be sought from the monitor. The need for this preparation is best illustrated by an example. Suppose two PPU programs, P and Q, are both looking for an empty entry in a table. P finds an empty entry and is just about to fill it when Q finds the same one, similarly deciding it to be empty. P will write out its entry, which will then be overwritten by Q, and therefore is lost. When changes to

system tables are interlocked through the monitor such chaos is prevented.

2.2.6 Job Flow through the System

In order to keep track of the jobs currently executing, the system uses the concept of a control point, an abstract entity to which resources can be allocated. There are seven control points under Scope 3.2 and in order to hold resources, a job must first be assigned to one of them. For each control point there is a system table (the control point area) that holds parameters necessary to continue processing of the job (e.g. control card images, CPU register contents and accounting information).

When a job is read in, it waits on an input queue. Scope notes its central memory requirements and computes a priority for it. A resource allocation program, IRA, is periodically called into a PPU by the monitor. If there is a vacant control point, IRA searches the input queue and selects the highest priority job for which memory is available. If such a job exists it is assigned to the vacant control point and memory is allocated. Another PPU program, IAJ, is called to advance the job through the various steps described by its control cards, through which the user specifies how Scope is to process the job. Control cards are discussed below (2.2.7).

Processing a control card usually involves execution of a

CPU program. When a CPU program needs the services of a PPU program (for example to perform input or output operations) it communicates this need to the monitor. A word of each control point's allocated memory is reserved for this purpose, and the monitor periodically inspects this word for the job using the CPU. When it detects a request, the monitor assigns an idle PPU as described previously (2.2.4). In issuing a monitor request, the CPU program has the option of recall, that is, of requesting that it relinquish the CPU until the PPU activity is completed. The monitor will then assign the CPU to another control point, if any is waiting for it. This can be done very rapidly by the exchange jump instruction, which interrupts the CPU and exchanges its registers with an area in central memory (which will hold the registers of the program to which the CPU is being transferred). All the information needed by the CPU to continue on the new program is then available. Monitor requests are also issued by the CPU to inform the system that the program has finished or wishes to abort the job.

At the end of processing for each control card, IAJ is called to advance the job to the next. When the last card has been processed, or if the job is aborted, the end-of-job processor, IEJ, is called. This PPU program releases all the resources acquired by the job, and prepares its output file for printing.

Control of the reading and printing processes is managed by a complex package of PPU programs that run at a specially assigned control point and are collectively named Janus. The effective number of control points available to users is thus six.

2.2.7 Control Cards

The wide range of software making up the Scope system library is made available to users by means of control cards. The first control card in a job deck is always the job card, which identifies the job and defines its initial resource requirements. Other control cards may be used :-

1. to call by name a PPU or CPU program from the system library and to start it executing, supplying parameters to direct its execution. This type of card is used to call compilers, editing packages, file-handling programs and the like. For example, the control card

   COPY(X,Y)

   directs that the program COPY be executed with parameters X and Y. The result will be that the contents of file X will be copied on to file Y. The form of the control card is independent of the devices on which X and Y reside.

2. to load a compiled program from a user's file (where it will usually have been written by a compiler), and to load also any system subroutines referenced by the program loaded. Execution of the program is then started.

3. to request a change in the allocation of resources, such as an increase or decrease in memory allocation. For example, the control card

RFL(20000)

requests that the central memory allocation of the job be changed to 20000_8 words.

The control cards appear together at the beginning of each job deck. Any number and any combination of control cards may be used to produce whatever affect is required. The control cards form, in fact, quite a powerful "language" of their own.

2.2.8 Disc Management

As mentioned above (2.1.5), some PPU programs in the system reside in central memory and others on the disc. Space in central memory is valuable, so that only the most frequently used programs can be stored there. A large number of important PPU programs and all the CPU programs (except for a few that work exclusively for the monitor, and stay in central memory) reside on the disc. The method used by Scope to manage the allocation of disc space and access to it is therefore of considerable importance to any study

of the system.

When a PPU program needs to access the disc (and a CPU program can only access it via a PPU program, so that this discussion covers all possible cases), it issues a request to the monitor called a stack request. The monitor places this request on a queue in central memory. If a PPU is already executing the program controlling the disc (called the stack processor, ISP), then this is all that the monitor need to. Otherwise it calls ISP into an idle PPU. Once in execution, ISP inspects the queue and selects the "best" entry for processing. This selection is based on a rough estimate of the access time involved in the request, and uses a shortest access-time first policy. When the stack request has been satisfied, ISP inspects its queue once again, and the process is repeated for the next request.

All requests for access to the disc must be made through the monitor, which directs them to the stack processor. Once the stack processor selects the request, it communicates directly with the requesting PPU about its progress. Issuing all requests through the monitor avoids the conflicts that would arise if each PPU made its own disc accesses.

The monitor always reserves one PPU for stack processing. If all the pool PPUs were assigned to other tasks and each of them issued a stack request, deadlock would occur because no PPU could ever become available to process the requests. As stack

requests occur very frequently, the liklihood of such a deadlock would be high if precautions were not taken.

The disc involved in the system being studied is the 6603-11 disc. The management of this disc employs the technique of half-tracking. Logically consecutive blocks of data are written on alternate sectors. Thus the stack processor can read or write a sector, and, while the unassociated next sector is under the heads, can prepare itself for the following one. This technique greatly increases the efficiency of disc processing.

As well as actually controlling the disc, the stack processor performs all the tasks associated with the allocation of disc space. On the 6603-11, the unit of space for allocation is the half-track, that is, the alternate sectors for one track (either 3200 or 4096 central memory words, depending on the track's position on the disc). The stack processor maintains list structures in central memory to record which half-tracks are assigned to each file, the order in which they should be accessed, and the file's current position. A record of reserved and available half-tracks is also kept. Some stack requests (e.g. to rewind or to evict a file) only require changes in these tables, with no actual disc access. Such requests have zero access time and are dealt with before the others.

2.2.9 Operator Activities

One of the aims of Scope 3.2 is to minimize the need for human intervention and indeed there are few situations where operator action is required. These instances fall into two classes. First there are the tasks of maintaining input output equipment, assigning tape units, responding to equipment faults, etc. Secondly, Scope allows the operator to manipulate the priorities of jobs, to declare whether certain control points are to be used or not, and to roll out and roll in jobs. The last two operations result in the memory of a job (which must be at a control point) being written out on to the disc and released for allocation to other users. The rolled out job keeps its control point, equipment assignments, disc space and other resources, and can easily be re-instated by the system. In certain situations a skilled operator can significantly improve the performance of the system by using these commands. For example, if a very long job occupying all the available memory is running, and a number of very small jobs are waiting, the operator would roll out the long job for a short time. The small jobs could then run, and the long job would continue later with relatively little increase in its running time.

2.3 SUMMARY

A principal characteristic of Scope is the means by which processors communicate - by issuing messages in appropriate places and waiting for them to be acknowledged. Also important is the distribution of tasks among the processors. The PPU programs control the system (the monitor and DSD), control job flow (IRA, IAJ and IEJ), drive input-output devices (ISP) and process CPU requests, while the CPU attends more directly to the users' needs.

The characteristics of PPUs, with their complete sovereignty and inaccessible memories, make investigation of their activities very difficult. This is a major problem since the PPU programs are so important to the running of the system that their behaviour and execution times need to be known in considerable detail.

Another aspect of the system that assumes particular importance in the development of the simulator is the communication between the monitor and the PPUs. Requests issued by peripheral processors to the monitor provide a valuable set of frequent events that can be detected and timed in the running system. These events are a key feature of the validation methods used for the model.

Investigation of disc performance presents severe difficulties both in the initial measurement of disc response times and in the validation of disc simulation. A detailed knowledge of disc behaviour is necessary because accesses to the main disc are frequent during even the most fundamental system operations.

CHAPTER 3

DESIGN AND DEVELOPMENT OF THE SIMULATOR

The research described in this and later chapters is directed towards the construction of a simulation model of the CDC 6400 system running under Scope 3.2. Subsequent chapters describe the development of the model in detail. This chapter considers the main problems involved in the simulation, and how these problems determined the overall design of the project.

3.1 PROBLEMS IN DEVELOPING A MODEL

Any model of a computer system is subject to a number of partially conflicting requirements. It should be efficient, in the sense that simulating a given period of real computer operations has a significantly lower cost than that of the real operations themselves. On the other hand, the model must achieve an adequate standard of accuracy, and this implies simulation at a sufficiently detailed level. As requirements for accuracy become more stringent, the level of detail must increase, and the simulator will run more slowly. If the simulator is made faster by removing detailed operations, then its accuracy will probably be impaired. Choice of a suitable balance between detail and efficiency is a most important part of designing the model.

Before this problem of balance is considered, however, a considerable amount of information about the computer and its operating system must be massed. Study of the structure of the computer and the operating

system before the model is constructed is indispensable, but is difficult on the 6400, in view of the autonomy of the PPUs and the inaccessibility of their memories.

Added to these problems is the necessity of proving the accuracy of the simulator by means of suitable validation trials. The model needs to be designed in such a way that the technical problems of performing a convincing validation are minimized.

3.2 THE DEVELOPMENT OF THE SIMULATION MODEL

The methods used to overcome the problems discussed above (1.2 and 3.1), together with the peculiarities of the 6400 (introduced in Chapter 2), largely determined the course of development of the simulation. The various phases in the construction of the simulator are shown schematically in Figure 4. A brief outline of the development will now be given, and this will be expanded in later paragraphs in the light of the problems solved during the development.

As shown in Figure 4, information about the 6400 system was obtained by two methods : detailed simulation of Scope system programs and direct measurement of the behaviour of input output devices. The first of these, involving a simulation, had to be validated. Timing information from the input-output devices divided into two classes : that which was deterministic (i.e. depending only on some electronic response, and involving definitely predictable times) and that which was stochastic (i.e. usually dependent on mechanical motion and therefore unpredictable). The deterministic information was combined with the results of the

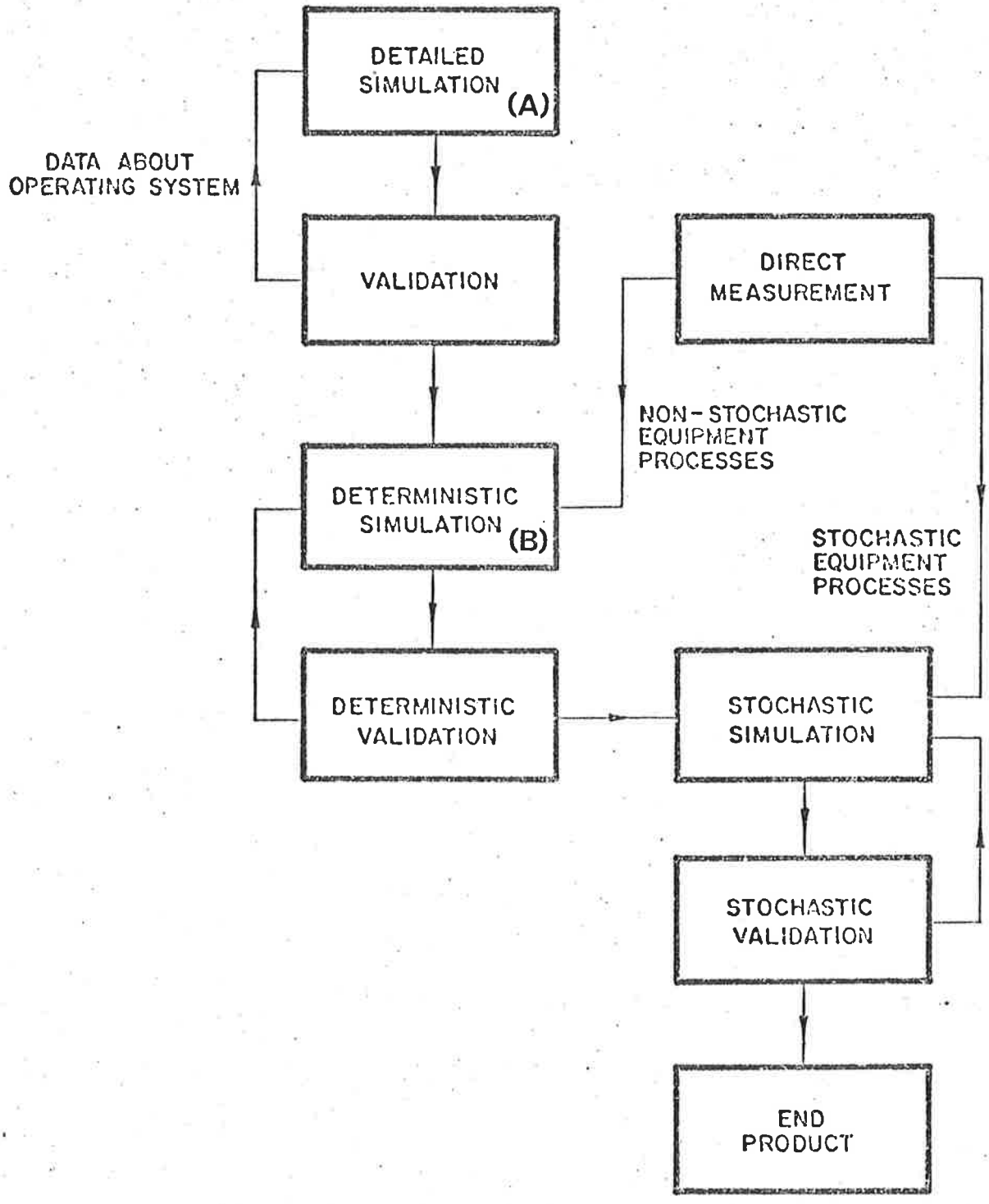


Figure 4. Scheme for constructing and validating CDC 6400 simulation.

detailed simulation (which were also precisely defined) to form the deterministic event simulator, which could then be validated using deterministic methods. The (relatively few) stochastic processes were then introduced, and the simulator was validated again using suitably adapted methods.

3.2.1 The Problem of the PPUs

It was shown in Chapter 2 that a major difficulty in studying the 6400 is the investigation of PPU activities. Much the greater part of the activity of the Scope operating system comprises the execution of a small number of PPU programs. These programs themselves consist largely of different combinations of a few basic system tasks, such as searching the library directory or constructing a stack request. It is important that the performance of these basic building blocks of Scope be accurately measured. Three kinds of investigation are required : identification of the basic processes, measurement of their execution times (and how these times vary with different states of the system), and gathering information about how the processes fit together in the various system programs.

It is not possible to obtain this information directly by monitoring the PPU activities in the real system. The PPUs cannot be interrupted or interferred with in any way and their memories are inaccessible to the rest of the system.

One way of carrying out the investigation would be to inspect the actual text of the PPU programs concerned. The execution time for each PPU instruction is known (with a few exceptions to be discussed later) and so the time required for any segment of a program could be computed from the instruction sequence. The execution times of the instructions executed would simply be aggregated. Doing this manually would, however, be very tedious and therefore error-prone. The paths taken by the programs often depend on the contents of the system tables, which would not be available if such a technique was used. In addition, the program text does not show immediately where programs spend their time. A much-executed loop that looks insignificant in assembly code might well take the greater part of the execution time of the program of which it forms a part.

3.2.2 Detailed Simulation

Many problems, and in particular those concerned with the PPUs, were avoided or overcome by carrying out simulations of the 6400 at two different levels of detail. A detailed simulator (Simulator A) was used to gather accurate information about the operations in the system, especially in the PPUs. This simulation was of course, slow, but the speed was not important because most of the processes being studied needed to be measured only once. This was because the only processes of interest here are those internal to one processor (i.e. not involving communication between processors). These results depend only on the programs

being executed and the processors executing them, and are identical for each occurence. They do not vary with the state of the system or with the number of jobs being processed.

Simulator A is fully described in the next chapter, and its results are discussed in chapter 5. Briefly, it operates at the instruction level, simulating in exact detail every instruction that takes place in each processor of the computer. Its results include execution times for processes local to one processor, an analysis of the way in which these processes are interlinked, and information about how processing varies with the state of the system. The main purpose of this simulator was investigation of PPU programs, about which it produced as much information as was required. Gathering this information by any other method would have been very difficult, if not impossible.

Because it is slow, Simulator A cannot be used to analyse computer performance over long periods ; a less detailed model is required for this. It is worth noting, however, that any quantitative results (obtained, for example, from the detailed simulator) incorporated in this less detailed model are of great importance because their accuracy can be increased (by more precise measurement), and thus the accuracy of the main model can be increased, but not at the expense of its efficiency. The less detailed simulation program will take the same time whether or not

the numbers it uses are accurate. Thus accuracy in the numerical results of the detailed simulator provides a limited escape from the dilemma of detail against efficiency described in Section 3.1.

3.2.3 Input-Output Equipment

The approximate response times for input-output devices are given by the manufacturer in the relevant reference manuals. Operations on these devices usually involve mechanical motion, so that the response times cannot be defined as precisely as are the instruction execution times. They are likely to be subject to stochastic variation.

In most instances, the operation speeds of these devices are not crucial to the processing times of the system. For example, the card readers, punches and line printers are controlled by a subsystem of PPU programs called Janus (see 2.2.6). This subsystem operates with a large measure of independence from the rest of the system. The response times of the devices it controls may affect the execution times of the programs in Janus, but not those of the system at large. For such devices as these, which have some degree of isolation from the important parts of the system, the figures quoted in the reference manuals are sufficiently accurate.

Certain devices, however, are very closely tied to the fundamental operation of the system, for example the 6603 disc. Most processes that require access to the disc wait until the

access is completed before continuing. The response times of the disc therefore have a direct effect on the performance of these processes, an important consideration, since disc accesses are frequent. The only way of obtaining more accurate timing information than that already available is to measure directly the required times on the system itself. The details of how this was done are given in Chapter 6. The response times for the disc were found to fall into two classes. Some, depending only on electronic switching, were predictable with considerable accuracy, (i.e. comparable with the accuracy of the detailed simulation results). Others involved mechanical motion and were subject to considerable random variation.

3.2.4 The Event Simulator

As is shown in Figure 4, the accurately predictable equipment response times and the information from the detailed simulation were combined to form a preliminary version of the final model, the deterministic event simulator (Simulator B). This model is realised in the form of a program that runs much faster (and with considerably less detail) than the instruction-level simulator. Its operation is described in Chapter 7. It works from a list of scheduled events, which can be modified during simulation. The earliest scheduled event on the list is selected, and the simulated time is set at the time of the event. The event is then simulated, and the whole process repeated.

All the information incorporated in Simulator B was exactly predictable because all the processing rules and execution times were known either from the detailed simulation or from the equipment measurements. The result was that, although detail was lost in going from Simulator A to Simulator B, accuracy was maintained at a comparable level. This was shown by the validation runs described in Chapter 8. Simulator B included all the 6400 processes except a few equipment responses. The advantage of not including the stochastic responses at this stage was that it enabled a precise and detailed deterministic validation to be carried out on the (almost complete) model, as described below.

The last aspects of system operation to be included in Simulator B were the stochastic processes in the input-output equipment. This needed no change in the structure of the simulator ; it merely involved activating those parts of it that simulated the random processes. The distributions from which random variables were selected were obtained by measurement in the case of the disc and from the manufacturer's information for other equipment which, as previously explained, is less critical.

3.3 VALIDATION

Multilevel simulation enabled the accuracy of the model and its results to be checked and closely controlled throughout its development. There were three points at which major validations were performed, as shown in Figure 4.

Firstly, Simulator A was validated before its results were used in Simulator B. This meant that the data used by Simulator B, was correct, and errors in its performance could be attributed to errors in the logic of its programming.

The next validation was of the deterministic version of Simulator B. Because the simulation at this stage was still exact, with no random effects, a validation, to be a good one, needed to show exact agreement between what the real computer did and what the simulator predicted for the same situation. Any discrepancy would immediately have pointed to an error in the simulator (or in the validation experiment). If validation had been delayed until the stochastic processes were included in the simulator, such discrepancies might possibly have been due to these processes. The approach used eliminated this uncertainty, and thus ensured that the deterministic parts of the simulator were as accurate as possible. Since the deterministic processes make up the majority of system operations, this stage of the validation was most important.

When the model was completed the only parts not validated were the stochastic processes. When these were introduced, the exactness of the previous validations no longer applied. Different techniques were required, and these are discussed in Chapter 8.

CHAPTER 4

THE DETAILED SIMULATOR

As explained in the last chapter, the main purpose of the detailed simulator (Simulator A) is to provide information, both qualitative and quantitative, about PPU activities in the Scope operating system. The idea is for this simulator to be an exact copy of the 6400 running under Scope. The advantage of the simulator over the real system is that any part of the former can be inspected during simulation.

4.1 OVERVIEW OF THE INSTRUCTION SIMULATOR

Simulator A works at the level of individual processor instructions. The level of detail of this simulator is about as fine as is possible without considering the internal mechanism of the 6400 processors. For a few exceptional PPU instructions, indeed, it is necessary to consider the internal mechanism in order to simulate the instructions properly (see 4.2.2). Except for these instructions, the internal workings of the processors do not affect the processing time, so that further refinement in the detail of the simulation would be pointless.

The main part of the simulator is a set of subprograms that simulate the instructions of the central processor and the ten PPUs. These subprograms operate on simulated registers, central memory and peripheral memories, which are exact copies, word for word, of their real system counterparts. The sub-routine simulating a processor simulates the programme stored in the processor's simulated memory instruction at

a time. The simulator is arranged to give the effect of the CPU and the ten PPUs running at the same time. A limited range of input-output equipment and the twelve data channels are also simulated. This whole system is incorporated in a single CPU program that runs on the 6400 as an ordinary user job. The program, when running, is an exact copy of the 6400 in microcosm, simulating the same instructions in the same programs as the real system in which it is embedded. This situation is represented in Figure 5, which is the same as Figure 1 with the simulation program running in the users' area of the system.

The simulator is a central processor program. The simulated CPU and PPU registers, central and peripheral memories, instruction codes and so on are stored within it and may be printed at any time during simulation. Hence the difficulties of investigating the processes taking place in sovereign PPUs is overcome.

4.2 SIMULATION OF THE PROCESSORS

The operation of the various parts of the instruction simulator will now be described more fully. The discussion will illustrate some of the finer details and difficulties involved in such a simulation.

4.2.1 Simulation of the CPU

Simulation of the CPU at the instruction level is a relatively simple operation and has been used before on the 6400 for such applications as debugging aids. The simulating program keeps a set of "registers" among which is the program address register, P,

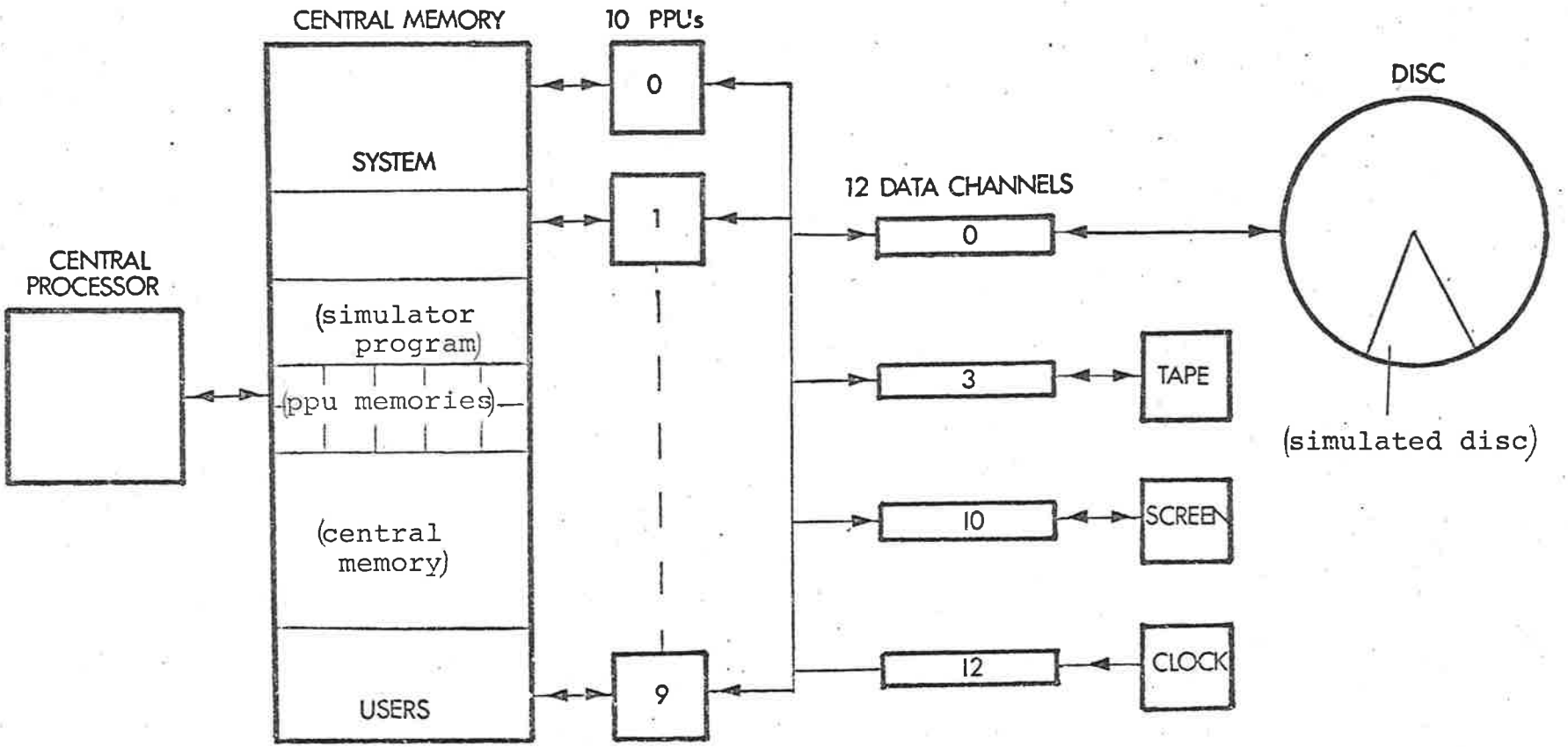


Figure 5 : The structure of the 6400 (as in Figure 1) with Simulator A embedded in it. Items in parentheses are simulated entities.

containing the address of the next instruction in central memory. The instruction in simulated central memory address P is decoded, and appropriate action is taken depending on its requirements. The register contents (including P) and central memory may be changed by this process. The procedure is repeated for as long as simulation continues or until a program stop instruction is encountered. In the latter case, further CPU simulation is suspended until a PPU restarts it.

Between execution of two instructions by the CPU simulator, a simulated PPU may execute a CPU interrupt instruction. This instruction redefines all the registers in the CPU so that it is swopped from one program to another. If the CPU has stopped, this will restart it. Otherwise there is no need for the CPU simulator to take special note of the event ; it continues as before, executing the new program in the new area of memory.

Certain complications arise in the detailed implementation of the CPU simulator. For example, some instructions use 18-bit arithmetic, which must be simulated using the 60-bit operations available to the simulator program. Although these problems add considerably to the complexity of the program, they are conceptually minor.

4.2.2 Simulation of the PPUs

Simulation of the PPUs operates on the same principle as for the CPU. The memory of each PPU holds the program being executed.

The mechanism by which the program gets into the simulated memory is described later (5.1.2, 5.1.3). Each simulated PPU has a P register, containing the address in that PPU's memory of the next instruction. The instruction is read, decoded and acted upon. One subprogram simulates all the PPUs, the number of the PPU to be simulated being passed to it as a parameter. Normally one instruction is executed on each call.

A major complication is that execution of an instruction is not necessarily completed in a definite time. Operations involving data channels, for example, depend on the state of the channel, which in turn may depend on another PPU using the channel, or on the response of equipment attached to the channel. This situation is simulated by breaking the instructions into segments, one of which is executed on each call to the PPU simulation subprogram, rather than the whole instruction. This is, in fact, the way such instructions are executed by the PPU hardware. The mechanism of these instructions can be complex and a typical example (the instruction to accept a data stream from a channel, referred to as the IAM instruction) is shown in Figure 6.

4.2.3 Central Memory Management

The CPU and the PPUs both access central memory. The CPU can potentially request two accesses at the same time (one for data access and one to read the next instruction). Requests for access to central memory are processed by a hardware device called the stunt box. Central memory is organized into banks of 4096 words,

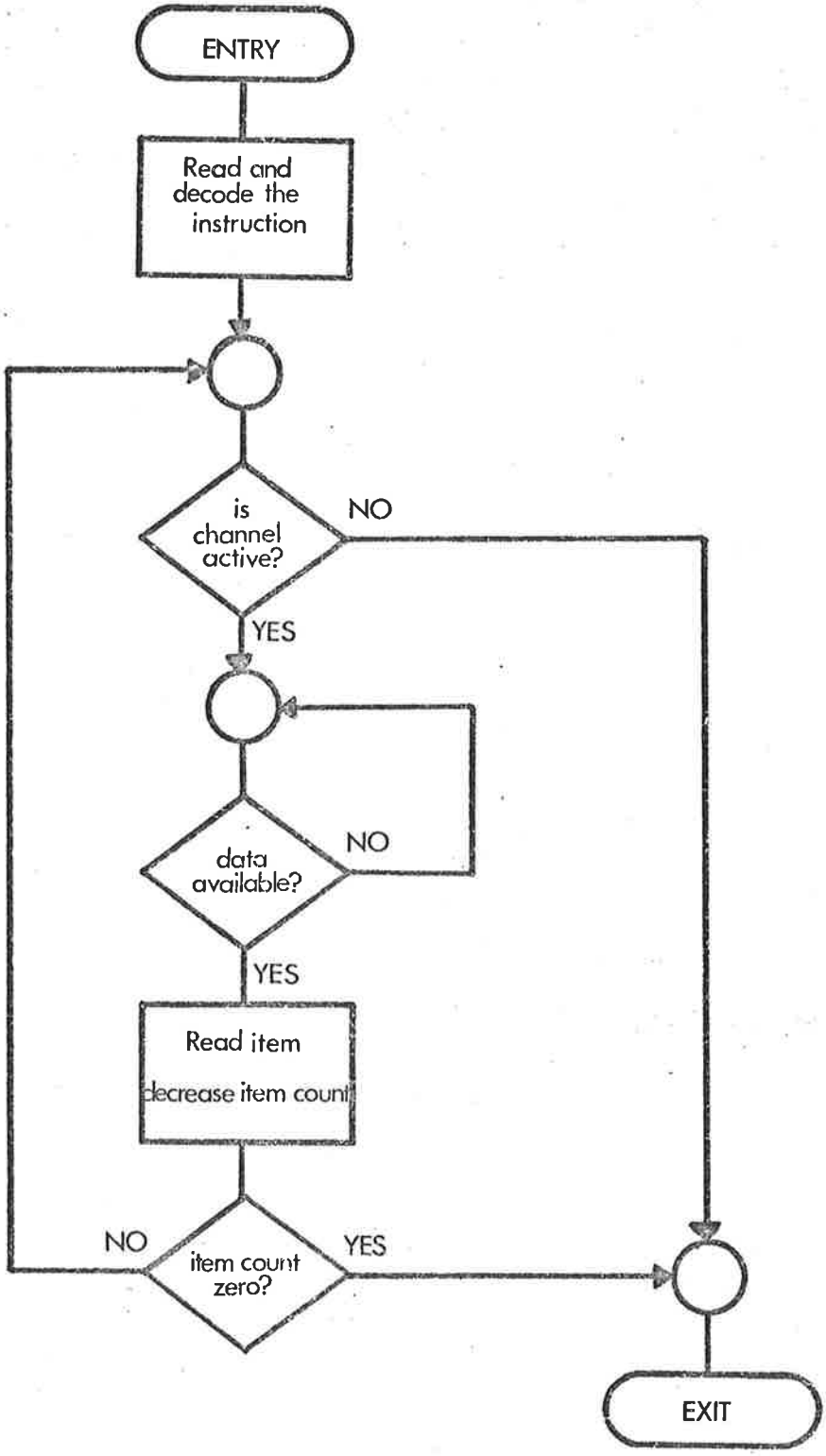


Figure 6 : The internal mechanism of the IAM instruction in a PPU

consecutive addresses being in different banks. Simultaneous requests for access to memory can be handled in parallel by the stunt box if they are for different banks. If they are for the same bank, however, a delay of about.2 micro-seconds, called a memory conflict, can occur. The number of central memory access requests being issued by the system at any particular time depends on the programs being executed, but on average is about 1.5 for each major cycle. This figure was taken from about 100000 cycles of processing in the detailed simulator. With sixteen memory banks, this gives a probability of .03 that a memory conflict will occur in a given cycle. The expected value of delays due to memory conflicts is thus about .006 microseconds for each major cycle (one microsecond) of processing. This was considered to be too small an effect to justify the relatively complex task of simulating it.

Another aspect of central memory simulation is clearly shown by Figure 5. The whole of simulated central memory is included in the real central memory of the simulating job, which itself occupies only a part of the real central memory of the computer. Thus the area of central memory simulated is considerably smaller than that available in the real system. This does not give rise to any major problems because there is sufficient simulated central memory to support the Scope operating system and to run small jobs. This is sufficient to simulate all the processes that the simulator was designed to study. Similar considerations apply to the

simulated disc which, as shown in Figure 5, is a part of the real disc.

4.3 SIMULATION OF THE INPUT-OUTPUT EQUIPMENT

Only the minimum number of input-output devices necessary to support Scope was simulated in the detailed model. Three devices were included : the 6603-11 disc, the operators' display console and a magnetic tape unit. The first two are intimately connected with system operations ; the tape unit is required to initialize the system, as described later. These devices are simulated by a sub-program which operates according to the following general pattern. It periodically inspects the status of each channel to which a device is connected, looking for a function code. Function codes will be sent along the channel by the simulated PPU controlling the device. When a function code appears, the subroutine performs the necessary processing for the device to reply to the simulated PPU. The device simulator is also active during transfer of data between a PPU and an input-output device. Essentially, the equipment simulator is an interface between the simulated PPUs (which expect the input-output devices to behave as the real equipment would) and the data areas used to represent the various devices. The methods used to achieve this are described in detail below.

The detailed simulator is concerned only with the activities of the processors in the 6400. The measurements obtained from it are the execution times for given segments of processing, as explained in the next chapter. Simulation of the input-output equipment simply provides logical responses and data to the PPUs so that they can continue properly

with their work. The equipment simulator therefore does not need to reflect the response times of the real equipment, as these response times do not affect the segments of processing that are being measured. The simulated equipment, in fact, responds to function codes within a cycle of simulation.

4.3.1 Simulation of the Display Console

Two processes take place in the display console. The more complex of these is the display of information on the console screens. In the real system this is done by first sending a pair of coordinates down the display channel, and then sending a stream of characters to be displayed at the position so defined. In the simulator the display screens were represented by an array of characters, and data sent down the simulated display channel were entered in the appropriate position in the array. The contents of the array could be printed at any time during simulation, producing an image of what the operator would see on the screens of the real system at the same point in processing. During the simulation runs, this facility proved to be a convenient way of seeing the state of the simulated system at a glance.

The display console also accepts entries from its keyboard, sending them back along the channel to the PPU. In the simulator, keyboard entries were read from cards into a keyboard buffer, where they were available to the equipment simulator when required. Keyboard entries were not often simulated in practice since more powerful and faster methods of communicating with the simulated

system were readily available. These methods were based on inspection and modification of the simulated registers and memories, and are described below (5.1.2).

4.3.2 Simulation of the Tape Unit

The simulated tape unit was required only for simulating the initialization of the system (described in 5.1.4). The operations performed by the tape unit during this process are simple, comprising only rewinding and reading. The simulated tape unit thus needs to provide only this small subset of the capabilities of a real tape unit. The tape itself is represented by a sequential data file stored on the (real) disc, which the equipment simulator reads and positions as demanded by the function codes sent along the simulated tape channel.

4.3.3 Simulation of the Disc

The disc is the most difficult of the input-output devices to simulate. The information stored on the simulated disc is held on a large file on the real disc. Each record of this file represents a sector of the real disc, and contains an exact copy of the 64 central memory words and 4 control words that this sector would contain in the real situation. This disc simulator accesses this file under the direction of the function codes sent along the disc channel by the PPUs.

The disc simulation program can recognise five function codes on the simulated disc channel. Two codes are to select a track and to select a head group, and for these the simulator simply records the number of required track or head. Another code asks the disc to send the number of the sector currently under the heads back along the channel (as a data item). In this instance the simulator always returns the sector following the last one accessed. During normal operations the real disc would be in this position, unless an unusually long time had elapsed since the last access.

The other two function codes initiate reading from or writing on a given sector. When one of these function codes is received, the sector number is combined with the previously recorded track and head group numbers to compute a unique disc address. The corresponding position on the indexed file representing the disc is computed from this address. In the case of a read operation, information is read from the appropriate record of the file and sent down the disc channel. For writing, information is accepted from the channel and written on to the file.

The detail of what happens during transfer of data between the disc (or any other input-output device) must be reflected in the simulator for the operation to be modelled correctly. When a sector is to be read from the disc, the simulated PPU will issue track seletion, head selection and read sector function codes. Using these, the disc simulator finds the appropriate record on

the indexed file and reads it into a buffer. The PPU then executes an instruction to read from the disc channel (the IAM instruction, illustrated in Figure 6). This causes it to wait until the first data item appears on the channel. It accepts the item, stores it and waits again for the next one. Meanwhile, the disc simulator sends a data item down the channel, and waits for the channel to be cleared, i.e. for the PPU to accept the item. The channel can hold only one data item (twelve bits) at a time. In order to simulate this process correctly, the processing in both the PPU and the disc must be broken into very small segments (transmission or acceptance of one item) and simulated alternately. The necessity of breaking down some PPU instructions in this way has already been discussed, (4.2.2). Here we see the process from the other end of the channel, from the disc's point of view.

4.4 TIMING CONSIDERATIONS

The subprograms that simulate the CPU, PPUs and input-output equipment must be combined so that they work together successfully. The simulated processors must also remain synchronized with each other so that the timing measurements are accurate. Some representation of the real-time clock (2.1.5) must be provided within the simulator to give measurement of simulated time. This section explains how these requirements were met in the simulator program.

4.4.1 Parallel Processing

Simulation of the parallel operation of the processors and input-output equipment is achieved by calling the simulating subroutine for each processor in turn as shown in Figure 7. This technique provides an exact simulation, so long as each call simulates a segment of processing that is small enough to be unaffected by the activities of other processors during that same segment. For example, suppose that a processor P issues a request to another processor, Q, at a certain time T, and suppose that Q detects the request 100 microseconds later. Suppose also that each processor is simulated, starting at T, for 200 microseconds at a time and that Q is simulated before P. Q will look for the request from P, but will not find it, because the issuing of the request by P would not have been simulated. Q would therefore behave differently in the simulation from how it would behave in the real computer. Suppose, however, that each processor is simulated for 50 microseconds at a time, still a much longer interval than that actually used in the simulation. The issuing of the request will then be simulated before Q looks for it, and the simulation will be correct.

The time for which each processor is simulated must be small enough for correct simulation of any communication between processors. When a stream of data is read from a channel, as described in paragraph 4.3.3, the PPU and input-output device interact within one PPU instruction, so that the instruction itself must be split up, and only part of it executed on each call to the PPU or device

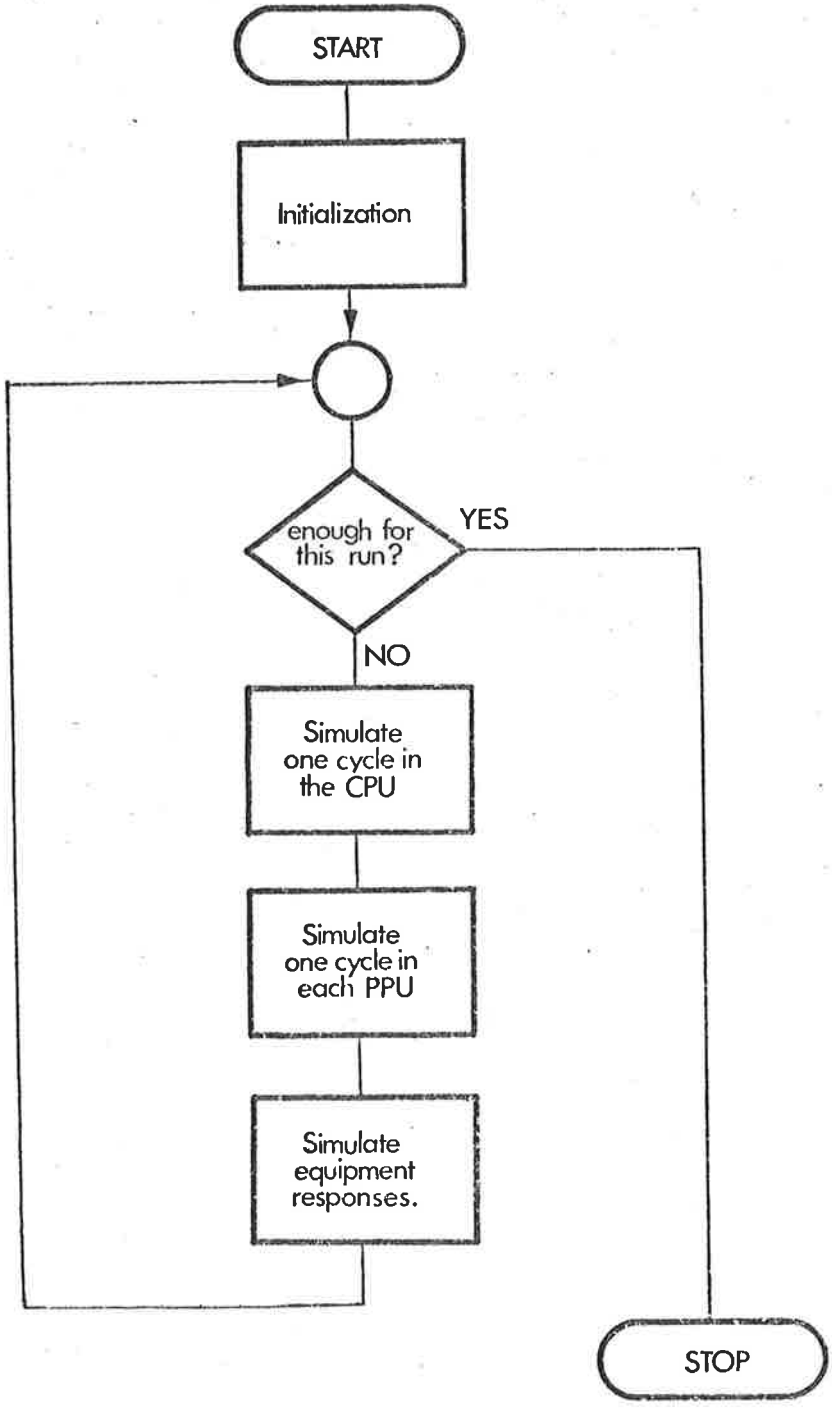


Figure 7 : The operation of Simulator A

simulator. The same applies to the instruction for writing blocks of central memory, where another processor can be affected before the instruction is completed. These instructions are exceptional, though, and in all other instances simulation of one instruction on each call to the simulating subroutine gives sufficient division of processing for accuate parallel simulation.

4.4.2 Synchronizing the Processors

Another problem in combining the various parts of the simulator is that different instructions take different times to execute. Those instructions that have variable execution times have been covered above (4.4.1). In addition, some fixed instructions take longer than others. With the processor simulators executing one instruction on each call, this could lead to trouble. Suppose one PPU, A, is running a program with a high proportion of slow instructions (for example, an operation on a table accessed by indirect addressing instructions which could take 5 microseconds each) while another PPU, B, has a program consisting mainly of short instructions (such as a simple counting loop, using instructions that take 1 microsecond each). For as long as this state of affairs exists, A will run faster (relative to B) in the simulator than it would in the real system. This kind of deviation in instruction mix would tend to even out in time, since PPU programs usually comprise a varied mixture of the types of processing used in A and B. Such deviation is undesirable, however, and is eliminated in the simulator (referring to the example) by blocking simulation

of A for four calls after each long instruction. Thus simulation of a 5-cycle instruction effectively takes five calls to the simulating subroutine, although all the necessary computation is done in the first call. Similar methods are used to keep the CPU synchronized with the PPUs. This technique does not interfere with the parallel processing considerations discussed in 4.4.1 since any instruction that can influence another processor within its own execution time is dealt with specially. The present discussion applies only to instructions that are completely internal to a processor, or whose influence on other processors is not felt until after their execution is complete.

4.4.3 The Clock

The basic time unit of the 6400 system is the major cycle (one microsecond), which is the memory access time for both central and peripheral memories. Instruction execution times are measured in terms of this unit, as is the clock maintained by the simulator, on which its timing results are based. This clock simply is a counter that is incremented once for each execution of the main simulation loop (i.e. for each call to all the simulation subroutines, the outer loop of Figure 7. By recording the value of this counter at the beginning and end of a segment of processing, the time (in microseconds) taken by the segment can be calculated.

4.5 SUMMARY

Simulator A is designed to give as detailed and as accurate a model of the 6400 system as practicable. To do this it simulates the execution of the programs in each processor instruction by instruction, maintaining registers and memory that are exact images of those in the real system. Limited simulation of input-output equipment is carried out, to provide the processors with the necessary logical responses and data for their work. These facilities enable a detailed qualitative representation of the 6400 to be set up and studied. Introduction of a clock and resolution of some problems of synchronization enable the simulator to be used quantitatively as well, to provide timing measurements for program segments.

The main application of Simulator A is the investigation and timing of inaccessible PPU processes in the Scope operating system. This chapter has described how the simulator provides the necessary facilities for doing this. The next chapter will discuss the experimental techniques evolved for the investigation.

CHAPTER 5

DETAILED SIMULATION OF THE SCOPE 3.2 SYSTEM

As Simulator A represents exactly the workings of the computer, its operations have the same order of complexity as those of the real system. Maintaining the simulated system and analysing results from it need special techniques which are described in this chapter. Also described here is the method used for validating Simulator A, and the results of the validation.

5.1 USE OF THE SIMULATOR

Because it uses program segments to decode and execute each instruction, and because it uses one real processor (the CPU) to do the work of eleven simulated ones, the detailed simulator is slow. Its precise speed depends on how detailed a report of processor activities is required, but an average figure would be about 1000 times slower than the real computer. This ratio demonstrates the need for a practical simulation to use faster methods, such as those employed by Simulator B, described in Chapter 7. Detailed simulation of the processing associated with even a small job (involving say five seconds of real time) is a large task, too large to be practicable in one simulation run. Techniques are therefore required to enable the simulation to be continued over a large number of runs. This section describes such techniques, and discusses a number of other matters related to the practical use of the simulator.

5.1.1 The Simulation Data Base

The (real) central memory used by the simulation program can be divided into two parts : that which changes as simulation proceeds (simulated memories, registers, channels and equipment information, etc.) and that which is constant (the program itself and constant data areas such as tables of instruction mnemonics and execution times). In order to continue simulation across several runs, it is sufficient to save the variable area of memory at the end of a run in such a form that it can be reloaded by the simulator at the start of the next run. The constant areas are recreated automatically when the program itself is loaded. The saved variable area will be referred to as the simulation data base.

The data base has other uses besides carrying a simulation from one run to the next. It greatly facilitates initialization of the system (described in 5.1.3). It is also possible to inspect and modify the data base between runs, thus communicating with and altering the system.

5.1.2 Editing the Data Base

The efficiency of the simulation process can be greatly improved by making careful changes to the data base between runs. Areas of processing that have been studied in detail already can be skipped by making the same changes to the data base as the omitted processing would make. This technique is particularly

useful for loading a PPU program from the disc, a frequent and lengthy operation. After such a loading process has been simulated a number of times, and its characteristics have been thoroughly studied, much time can be saved by, instead of simulating the loading operation, performing equivalent operations on the data base. The required program is written into the simulated memory of the requesting PPU in the data base. Other changes (to pointers and registers in the PPU) are also made, so that the result is exactly as if the simulated PPU had performed the operation by itself by issuing a stack request in the usual way. The effect of this modification can be allowed for in the results of the simulation because the processes skipped in this way are well understood from analysis of the results of previous simulation. An example of the adjustment of results to allow for such modifications is presented below (5.2.3).

Another application of changes to the data base is recovery after bad simulation of an instruction. Despite the extensive qualitative checking described below (5.1.4), some errors in the simulator were not detected until production runs had started. The effects of such a malfunction could be eliminated by suitable modifications to the data base. For example, a difference in the branching operations of the CPU (which was doing the simulation) and the PPUs (which were being simulated) resulted in a zero test taking the wrong branch when given an operand of -0. The first time this situation was encountered during simulation, the error

resulted in a small table in the PPU's memory being set up incorrectly. By changing the table in the data base, the error was corrected without repeating the simulation. This was a valuable saving because re-starting the simulation from the beginning would have taken considerable amounts of computer time and programming time. At the same time, the simulation program itself was corrected so that future execution of the offending instruction under the same conditions would be correct. Fortunately, errors of this kind were rare, only three being detected in some 500000 cycles of simulation.

5.1.3 Initializing the System

In the real computer, the system is brought into operation by forcing a short program into a PPU from a matrix of switches. The other PPUs are forced to execute instructions to read from a channel (the IAM instruction of Figure 6), and so wait until the first PPU sends a program to each along the appropriate channel. The instant at which this initialization commences is the only exception in the system to the sovereignty of the PPUs. Initialization continues with the first PPU reading the system tape, which contains all the programs in the system library. The system tables and the library directory are set up in central memory, and the other parts of the system are started. The first PPU becomes the monitor when the initialization process (called dead-start) is complete.

In theory, the simulated system could be initialized by making it simulate the entire dead-start process. A copy of the system

tape could be used for the simulated tape unit. Correctly initializing the PPUs and then simulating for as long as necessary would result in a facsimile of all the processes of dead-start. The simulator would eventually have central memory set up. The monitor and DSD would be executing in two PPUs and PP resident would be running in the other eight. All the programs from the system tape would be set up on the simulated disc and their positions recorded in the library directory (in simulated central memory). The simulated system would then be ready to run jobs.

Unfortunately the entire dead-start process is too long to be simulated in full. The problem of initializing the simulated system, which is decidedly non-trivial, was solved by writing a program to set up the data base from the system tape. Central memory, PPU memories, registers and channels are initialized and written on the data base. When the data base, thus initialized, is used by the simulation program, simulation begins at the final stage of system initialization. The monitor, DSD and PP resident have all been loaded into the appropriate PPUs and are executing. The CPU is idle and the tables in central memory have been set up.

The most difficult part of the dead-start simulation is automatic construction of the library directory. This must contain, in simulated central memory, the address on the disc of each program in the system library. In order to compute these from the lengths and sequence of the programs on the system tape, all the disc allocation algorithms of the Scope system have to be simulated. For

example, suppose that at a given point in the dead-start process the disc has been filled up to sector S of half-track T, and that this half-track contains a total of M sectors (M is 50 or 64 depending on which zone of the disc half-track T is in). Suppose also that the next program to be entered in the library is N sectors long. If N<M-S, then the program will fit into the remaining space in half-track T. If N>M-S, the first M-S sectors are written on to half-track T, and it is then necessary to select a new half-track (usually T+1), which could involve a zone change (and thus a new value of M). The remaining N-M+S sectors of the program are then written on the new half-track beginning at sector 0. For some programs, N-M+S is still greater than M, so that a further half-track must be selected and the process repeated. This whole procedure is carried out for each program in the system library. The half-track and sector at which each program starts is recorded in that part of the data base that represents the library directory (a part of simulated central memory). The half-tracks assigned to the system library are also recorded, in the disc reservation tables (another area of simulated central memory). The programs of the operating system are written out on to the simulated disc file in the appropriate positions.

5.1.4 Qualitative Validation

A program as complex as Simulator A, with its bit-by-bit accuracy requirements, must be fully tested qualitatively before being used. Qualitative validation involves checking that every

instruction executable by every processor is simulated in the correct way and gives the correct results. Malfunction of an instruction in the simulated system, through an error in programming, is analagous to a hardware fault in the real system. The effects of such a fault can be obvious or subtle, depending on the nature of the fault and the frequency with which it is encountered. Most simulation errors have immediate and far-reaching effects, usually culminating in deadlock of the system before simulation has proceeded very far. Errors of this type are easy to detect and can be quickly eliminated. For example, one of the first errors detected was that when a PPU read a single word from central memory, the address of the word read was one more than it should have been. This soon led to trouble because, among other things, the PPU programs were looking in the wrong place for their instructions from the monitor. An example of a subtle error was that, when -0 resulted from an addition in a PPU, the simulator did not convert it to +0, as is done by the PPU hardware. The result of such an addition, in the simulator, passed a test for negativity when it should have failed. The path taken by a PPU program, (and thus its execution time) sometimes depends on the result of such a test. Incorrect simulation can therefore cause the simulator to produce incorrect timing results, even if it is not fatal to the system. A similar error, described above (5.1.2) escaped the qualitative testing and was not detected until simulation was underway.

An ideal method for the qualitative testing of the simulator is to simulate some of the dead-start process. Simulation of the entire process is impracticable, for reasons given above (5.1.3), but only a part need be covered for testing. Dead-start is better than normal system operations because the interaction of the processors is much more vigorous (and so more sensitive to bad simulation). All processors, all channels, and all three input-output devices included in the simulation are used at dead-start. Most PPU instructions are encountered during dead-start processing, but CPU usage is more limited. If the simulator behaves correctly during dead-start, then the qualitative aspects of the program (i.e. those involving the processes being simulated, but not necessarily the timing measurements) can be regarded as being fairly reliable. Correct behaviour of the system can be checked by watching how it acts at various decision points. For example, if a simulated keyboard entry is made requesting that a memory dump be produced during dead-start, and the system skips over the dump program, then something is wrong. Other areas that can be checked are the simulated display (which should contain, character for character, the same information as the real display screens at the same point in processing) and the tables in central memory, which can be compared with real central memory.

The CPU simulator was not much involved in the early stages of dead-start and had to be more fully tested separately. This was easily done by running CPU programs with it and comparing the

results obtained with those given by the same programs run with the real CPU.

The qualitative validation detected a number of errors that would probably otherwise have remained in the simulator until well into the production runs. The area where it was of greatest value was in checking the simulation of the disc, and the close interaction between the disc and the PPU controlling it during data transfers, described above (4.3.3).

Once the simulator has been carefully checked it can be used to obtain quantitative results, as described in the next section. When these have been obtained, they can be used for quantitative validation as described in 5.3.

5.2 RESULTS FROM THE DETAILED SIMULATION

This section describes the results from the detailed simulation and the methods used to reduce them to a convenient form. The significance of some of the results is also discussed.

5.2.1 The Test Job

A short but varied test job was designed to run using the simulated system. This job first did some simple resource manipulations (changing its memory allocation and setting system flags). This was followed by a Fortran compilation with errors, a recovery, and then a compilation of a correct Fortran program. Finally, the compiled program was loaded (with the required CPU programs from the system library) and executed. The program comprised a short

arithmetic computation and a print statement. After executing this program, the job terminated normally. This test job was designed to include execution of all the commonly used PPU programs, which were the programs of most interest. Work with Simulator A consisted mainly of working through the processing connected with the running of this test job.

5.2.2 Output from the Simulator

As it analyses and executes the instructions for each of the processors of the 6400, Simulator A can trace what each processor and input-output device does during a run, and parts of this trace can be selected for printing. For processors, the trace comprises a list of the instructions executed, the contents of any registers changed by each instruction, and information read from or written into central memory. For an input-output device, the trace includes any function codes sent along the appropriate channel, and data sent in either direction along the channel. In addition, timing marks can be included in the trace at intervals of 64 major cycles. Using these, the times at which events occur in the system can be measured, as described below. The trace can be printed for any number of processors at a time, but usually only one was taken at a time, for the sake of clarity. The timing marks can be used to match up separate listings for simultaneous processes. An extract from the simulation trace for the monitor, running in PPU 0, is shown in Figure 8. Most of the items in this extract represent executions of PPU instructions, and include (from

```
PP 00    0413    3407 0000    STD  07 0000    A = 000600
PP 00    0414    1621 0000    ADN  21 0000    A = 000600
PP 00    0415    6037 0000    CRD  37 0000    A = 000621
WORD 000000000000000000000 (                ) READ  FROM CM LOCATION 000621 BY PP 0
                                                          MAJOR CYCLES: 1105300

PP 00    0416    3041 0000    LOD  41 0000    A = 000621
PP 00    0417    0463 0000    ZJN  63 0000    A = 000000
PP 00    0463    0100 0110    LJM  00 0110    A = 000000
PP 00    0110    0310 0000    UJN  10 0000    A = 000000
PP 00    0120    0310 0000    UJN  10 0000    A = 000000
PP 00    0130    2000 0101    LDC  00 0101    A = 000000
PP 00    0132    3457 0000    STD  57 0000    A = 000101
PP 00    0133    6032 0000    CRD  32 0000    A = 000101
WORD 001200020000010031372 ( J B  HCK<) READ  FROM CM LOCATION 000101 BY PP 0
PP 00    0134    3032 0000    LOD  32 0000    A = 000101
PP 00    0135    0403 0000    ZJN  03 0000    A = 000012
PP 00    0136    0100 1250    LJM  00 1250    A = 000012
PP 00    1250    1740 0000    SBN  40 0000    A = 000012
PP 00    1251    0703 0000    MJN  03 0000    A = 777751
PP 00    1254    5032 1317    LDM  32 1317    A = 777751
PP 00    1256    3401 0000    STD  01 0000    A = 002067
PP 00    1257    0302 0000    UJN  02 0000    A = 002067
PP 00    1261    0101 0000    LJM  01 0000    A = 002067
PP 00    2067    0200 4373    RJM  00 4373    A = 002067
PP 00    4374    0200 4556    RJM  00 4556    A = 002067
PP 00    4557    5057 5457    LDM  57 5457    A = 002067
PP 00    4561    3407 0000    STD  07 0000    A = 000400
PP 00    4562    1070 0000    SHN  70 0000    A = 000400
PP 00    4563    3406 0000    STD  06 0000    A = 000002
PP 00    4564    0415 0000    ZJN  15 0000    A = 000002
PP 00    4565    3007 0000    LOD  07 0000    A = 000002
```

▲ Detects a monitor request from PPU 3

Figure 8 : A segment of the trace from Simulator A. This segment shows part of the execution of the monitor.

left to right), the PPU number, the P (program address) register, instruction code, instruction mnemonic, and A (accumulator) register. Also shown are words read from and written into central memory by the monitor, as well as the timing marks every 64 major cycles. The number of instructions executed between timing marks varies depending on the execution times of the instructions concerned.

The time at which an event occurs in the simulated system can be measured using the timing marks. For example, in Figure 8, to measure the time at which the monitor detected the request to drop the PPU, the value of the preceding timing mark (1105300_8) is noted. The execution times of the instructions simulated between this timing mark and the request are then added, to give a total of 1105320_8. This is the number of major cycles simulated since the completion of the dead-start process. By similarly noting the time at which the request was answered (which was 1106000_8) cycles, the time taken to process the request (460_8 cycles) can easily be calculated.

During simulation of the test job, the times at which various events occurred in the simulated system were recorded. Events of particular interest were inter-processor communications (such as PPU writing a request in its output register) and changes of processors from one kind of task to another (e.g. from loading a program to executing it). Processing of the test job was normally simulated at a rate of about 2 milliseconds in each run, with

separate traces being produced for each active processor. The event times were collected by hand into a list called the simulation log, and it was from this that the results of the simulation were extracted for application to Simulation B.

5.2.3 The Simulation Log

A typical extract from the simulation log, showing the times of important events in the system during a period of a few milliseconds, is shown in Figure 9. During its processing, the test job produced about 750 of these events, each of which was timed with an accuracy of about 10 microseconds, using the simulation trace as described above (5.2.2). Validation of these results (see 5.3) was only possible, however, to an accuracy of about 200 usec. Figure 9, at 760000_8 cycles, contains a modification to the data base, replacing the execution of a stack request.

The first stage in the analysis of the simulation log was the insertion of events that had been passed over by changes to the data base as described above (5.2.1). In the example of Figure 9, the process passed over in this way was a stack request to read from the disc into central memory. It was known from previous simulations of this process that it comprised the events and corresponding times shown in Figure 10. These events were inserted into the simulation log of Figure 9 by setting the value of T in Figure 10 to 761770 (the times at which the stack request would have been made). The times for the events after stack request were increased by the time the stack request would have taken

```
747350   CPU   FORTRAN   ISSUES A REQUEST FOR CIO TO WRITE ON DISC
747440   PP0   MTR       DETECTS REQUEST FOR CIO
747700   PP0   MTR       PASSES CIO REQUEST ON TO PP3
747720   PP0   MTR       CLEARS REQUEST FOR CIO
747722   CPU   FORTRAN   DETECTS CLEARING AND CONTINUES
747725   CPU   FORTRAN   ISSUES REQUEST TO RELEASE CPU
747750   PP3   RESIDNT   DETECTS REQUEST FOR CIO AND STARTS LIBRARY SEARCH
750160   PP0   MTR       DETECTS RCL REQUEST
750360   PP3   RESIDNT   STARTS READING CIO
750630   PP0   MTR       CPU EXCHANGE JUMP
750630   CPU   FORTRAN   STARTS IDLING
750700   PP0   MTR       COMPLETES PROCESSING OF RCL REQUEST
752110   PP3   CIO       STARTS EXECUTION
753100   PP3   CIO       ISSUES MONITOR REQUEST FOR FST INTERLOCK
753160   PP0   MTR       DETECTS INTERLOCK REQUEST
753240   PP0   MTR       PERFORMS INTERLOCK
753300   PP0   MTR       COMPLETES INTERLOCK REQUEST
753300   PP3   CIO       DETECTS THAT INTERLOCK REQUEST IS COMPLETED
753510   PP3   CIO       RELEASES THE INTERLOCK
754100   PP3   CIO       STARTS SEARCHING LIBRARY FOR SUBROUTINE 4ES
755000   PP3   CIO       STARTS READING 4ES FROM CENTRAL MEMORY
760560   PP3   CIO       FINISHES READING 4ES
761770   PP3   CIO       ISSUES STACK REQUEST TO WRITE ON THE DISC
761770   PP3   CIO       ISSUES REQUEST TO RELEASE PPU
761770   ***   *******   ALTERATION TO DATA BASE TO SKIP STACK REQUEST
762000   PP0   MTR       DETECTS REQUEST TO RELEASE PPU
762240   PP0   MTR       RE-STARTS THE CPU
762240   CPU   FORTRAN   STARTS AGAIN FOLLOWING WRITE REQUEST
```

Figure 9 : Part of the simulation log produced from the results of Simulator A. Times are in octal microseconds.

T	PPX	***	ISSUES STACK REQUEST
T+000070	PP0	MTR	DETECTS STACK REQUEST
T+000250	PP0	MTR	PASSES STACK REQUEST ON TO STACK PROCESSOR
T+000240	PP8	1SP	DETECTS THE STACK REQUEST
T+000320	PP0	MTR	INFORMS PPX THAT REQUEST IS BEING PROCESSED
T+001700	PP8	1SP	ISSUES REQUEST TO RESERVE DISC CHANNEL
T+002020	PP0	MTR	DETECTS REQUEST AND ASSIGNS CHANNEL
T+002110	PP0	MTR	INFORMS 1SP THAT DISC CHANNEL HAS BEEN ASSIGNED
T+002300	PP8	1SP	DETECTS RESPONSE FROM THE MONITOR
T+004300	PP8	1SP	REQUESTS A CHANGE IN CONTROL POINT ASSIGNMENT
T+004400	PP0	MTR	DETECTS REQUEST FOR CHANGE OF CONTROL POINTS
T+004640	PP0	MTR	COMPLETES CHANGE AND INFORMS 1SP
T+004720	PP8	1SP	DETECTS COMPLETION OF REQUEST
T+006100	PP8	1SP	STARTS VERIFICATION OF DISC POSITION
T+045000	PP8	1SP	FINISHES VERIFICATION
T+045500	PP8	1SP	STARTS TRANSFER BETWEEN CENTRAL MEMORY AND DISC
T+046200	PP8	1SP	FINISHES TRANSFER OF INFORMATION
T+046500	PP8	1SP	ISSUES REQUEST FOR CHANGE OF CONTROL POINTS
T+046620	PP0	MTR	DETECTS REQUEST FOR CONTROL POINT CHANGE
T+047000	PP0	MTR	COMPLETES CHANGE AND INFORMS 1SP
T+047030	PP8	1SP	DETECTS COMPLETION OF CHANGE
T+047300	PP8	1SP	RELEASES DISC CHANNEL

Figure 10 : The events that occur when a PPU program issues a stack request to transfer one record between central memory and the disc. The request is issued at time T. Times are in octal microseconds.

```
    754100  PP3  CIO       STARTS SEARCHING LIBRARY FOR SUBROUTINE 4ES
    755000  PP3  CIO       STARTS READING 4ES FROM CENTRAL MEMORY
    760560  PP3  CIO       FINISHES READING 4ES
    761770  PP3  CIO       ISSUES STACK REQUEST TO WRITE ON THE DISC
    761770  PP3  CIO       ISSUES REQUEST TO RELEASE PPU
*   762060  PP0  MTR       DETECTS REQUESTS
*   762240  PP0  MTR       PASSES STACK REQUEST ON TO STACK PROCESSOR
*   762250  PP8  1SP       DETECTS THE STACK REQUEST
    762310  PP0  MTR       RELEASES PP3
*   763670  PP8  1SP       ISSUES REQUEST TO RESERVE DISC CHANNEL
*   764010  PP0  MTR       DETECTS REQUEST AND ASSIGNS CHANNEL
*   764100  PP0  MTR       INFORMS 1SP THAT DISC CHANNEL HAS BEEN ASSIGNED
*   764270  PP8  1SP       DETECTS RESPONSE FROM THE MONITOR
*   766270  PP8  1SP       REQUESTS A CHANGE IN CONTROL POINT ASSIGNMENT
*   766370  PP0  MTR       DETECTS REQUEST FOR CHANGE OF CONTROL POINTS
*   766630  PP0  MTR       COMPLETES CHANGE AND INFORMS 1SP
*   766710  PP8  1SP       DETECTS COMPLETION OF REQUEST
*   770070  PP8  1SP       STARTS VERIFICATION OF DISC POSITION
*  1026770  PP8  1SP       FINISHES VERIFICATION
*  1027470  PP8  1SP       STARTS TRANSFER BETWEEN CENTRAL MEMORY AND DISC
*  1030170  PP8  1SP       FINISHES TRANSFER OF INFORMATION
*  1030470  PP8  1SP       ISSUES REQUEST FOR CHANGE OF CONTROL POINTS
*  1030610  PP0  MTR       DETECTS REQUEST FOR CONTROL POINT CHANGE
   1030700  PP0  MTR       RE-STARTS THE CPU
   1030700  CPU  FORTRAN    STARTS AGAIN FOLLOWING WRITE REQUEST
*  1030770  PP0  MTR       COMPLETES CHANGE AND INFORMS 1SP
*  1031020  PP8  1SP       DETECTS COMPLETION OF CHANGE
*  1031270  PP8  1SP       RELEASES DISC CHANNEL
```

Figure 11 : Part of the simulation log with skipped events (marked with an asterisk) inserted for the stack request in Figure 9.

(46540_8 major cycles). The modified simulation log is shown in Figure 11. In this case, 20 milliseconds of processing was saved by modifying the data base.

The times taken for processes internal to a processor (i.e. not involving communication with another part of the system) are easily determined from the simulation log, which records the times at which such processes start and finish. The times for some common processes are listed in Figure 12. Execution times for processes involving communication between processors (e.g. servicing a request made to the monitor) can be obtained by the log as well, but these can vary, depending on what is happening in other parts of the system. For example, the time taken to reply to a monitor request comprises (1) the time taken for the monitor to detect, process and reply to the request, and (2) the time spent waiting while the monitor processes other requests issued before the one being discussed. The first of these components is fixed, ignoring the slight differences caused by the different positions of the monitor's polling loop at which the request might be issued. The simulation log was used to measure this fixed component of the monitor's response time, and the result (for most types of requests - some take longer) appears in Figure 12 as T_M. Component (2) depends on what other requests have been issued, and must be determined for each situation as it arises. The methods used for this are described in the discussion of the event simulation (7.4.1).

Process	Symbol	Execution Time (milliseconds)
Search of the library directory for the ith program	T_L	.08 + 03i
Reading a PPU program i words long from central memory	T_R	.02 + 005i
A complete search of the file table		3.5
A search for a particular file in position i of the file table		.08 + 02i
Storage re-adjustment (without a memory move)	T_S	1.6
Response time to a request to the monitor from a PPU (monitor idle)	T_M	.1
Verification of a stack request		.75

Figure 12 : Typical execution times for some common Scope processes.
(The symbols are used in Figure 13)

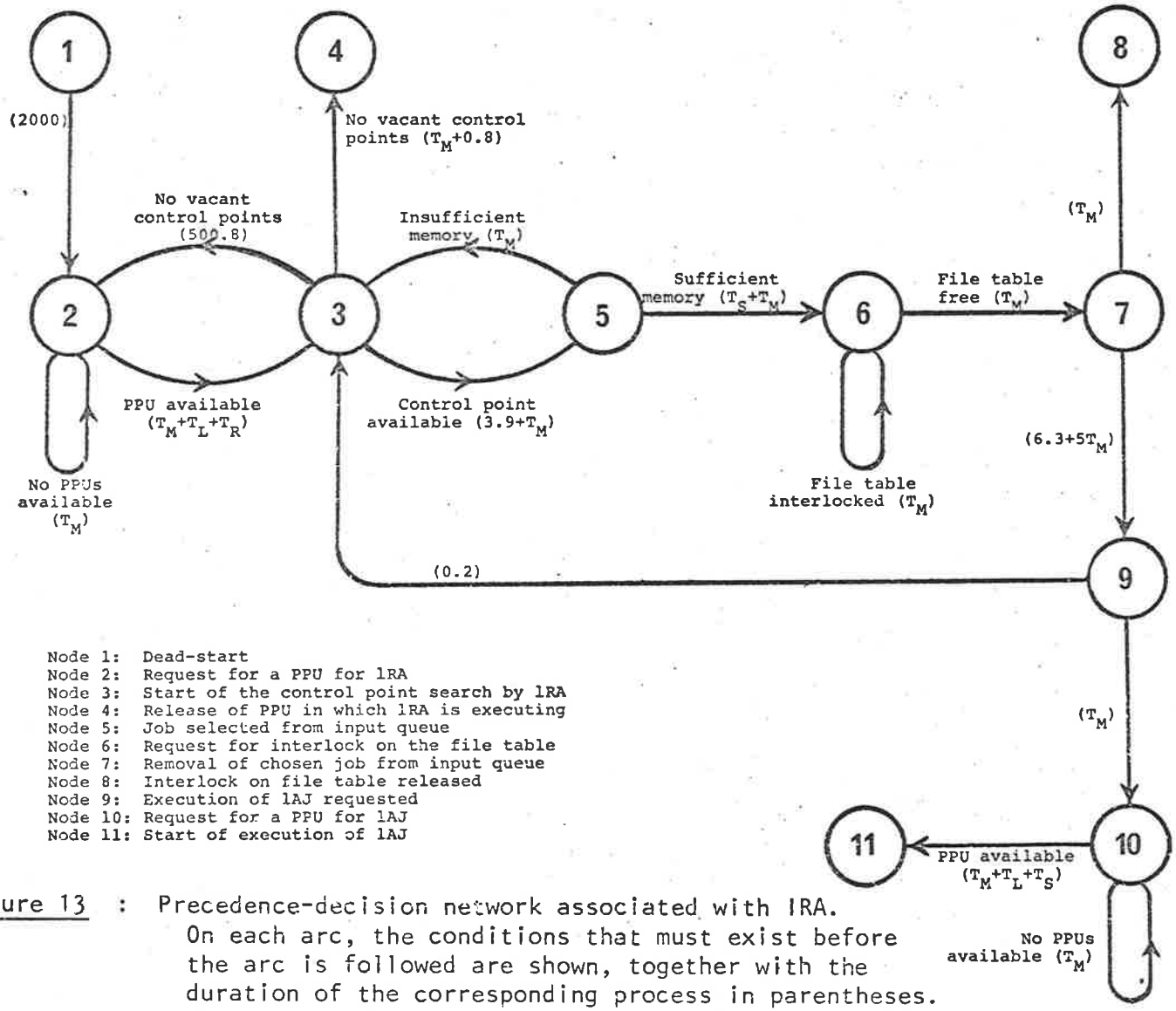
5.2.4 Quantitative Networks

It is difficult to devise a way of representing the processes that take place in a complex operating system. One method is the precedence-decision network, developed extensively by Noe and Nutt (55, 56) in connection with the Scope operating system. These networks can be used to describe a system at any level of detail and provide a picture of system operation from which a simulation model can be designed.

The results of analysing the simulation log can be represented as a series of such networks, one for each of the programs simulated. These program networks link up to form a larger network describing the whole Scope system. The networks formed from the simulation log are very detailed. Each arc represents either a process internal to the program (i.e. not making any requests to other parts of the system) or a delay while waiting from some other process to finish (e.g. waiting for a PPU to become available). Processes in the first class each take a definitely predictable time, which can be measured from the simulation log as described above. The time involved in the delays depends on what is happening in other parts of the system. Although this is not known in the general case, it can be found precisely in any particular instance. Networks obtained from the detailed simulation therefore contain, not only information about precedence and decisions, but also quantitative data about the processing time associated with each arc.

A typical example of the networks produced at this stage of the work is shown in Figure 13. This represents the processing associated with the execution of IRA, which allocates resources to jobs and initiates job execution. Some of the processes represented are not done by IRA itself, but by the monitor in response to requests from IRA. In the following discussion, numbers in brackets refer to the nodes of Figure 13. IRA executes for the first time two seconds after dead-start (1). A PPU is assigned (2) and IRA is loaded from central memory and starts execution (3). If there is an idle control point, the file table is searched for a job waiting to be run. If there is such a job, and there is sufficient memory available to meet its requirements, a request is made for the memory to be assigned (5) and the job is set up at the idle control point. This process requires an interlock on the file table, which is requested at (6) and released at (8). When the control point has been set up, the job is ready for execution (9), so IAJ (the program that interprets job control cards) is called into a PPU for this purpose (10). Meanwhile, IRA returns to the start of its processing (3) to look for another idle control point. When none remains, it asks the monitor to run IRA again after a delay of about half a second, and releases its PPU (4).

These networks, and their application to constructing simulator B, are further discussed in 7.4.1.



Node 1: Dead-start
Node 2: Request for a PPU for 1RA
Node 3: Start of the control point search by 1RA
Node 4: Release of PPU in which 1RA is executing
Node 5: Job selected from input queue
Node 6: Request for interlock on the file table
Node 7: Removal of chosen job from input queue
Node 8: Interlock on file table released
Node 9: Execution of 1AJ requested
Node 10: Request for a PPU for 1AJ
Node 11: Start of execution of 1AJ

Figure 13 : Precedence-decision network associated with 1RA. On each arc, the conditions that must exist before the arc is followed are shown, together with the duration of the corresponding process in parentheses. Where no condition is given, the arc is always followed. Symbolic times are defined in Figure 12.

5.2.5 Other Results

A number of interesting results arose from the detailed simulation besides those used in the construction of the event networks described above. These results, which are discussed in detail below, were sometimes significantly different from what had been expectcd intuitively, and thus reinforce the need for precise methods for measuring the system.

A survey was made of the time spent by the PPUs on various tasks during execution of the test job. The result is shown in Figure 14. The time spent on repetitive tasks, such as searching tables and waiting for requests proved to be considerably higher than expected. Relatively little processor time was spent on the more interesting aspects of processing, such as controlling equipment and job flow. This diagram does not take into account the time that PPUs spent idling (i.e. not assigned to any program, but waiting for the monitor to give them work). It gives an indication of how PPUs spend their time when they have been assigned work.

The amount of time devoted by the system to stack processing (i.e. controlling the disc) was greater than expected. During execution of the test job, twenty stack requests were issued, and processing these formed the greater part of system work on the job. This result demonstrates the importance of accurate disc simulation in Simulator B.

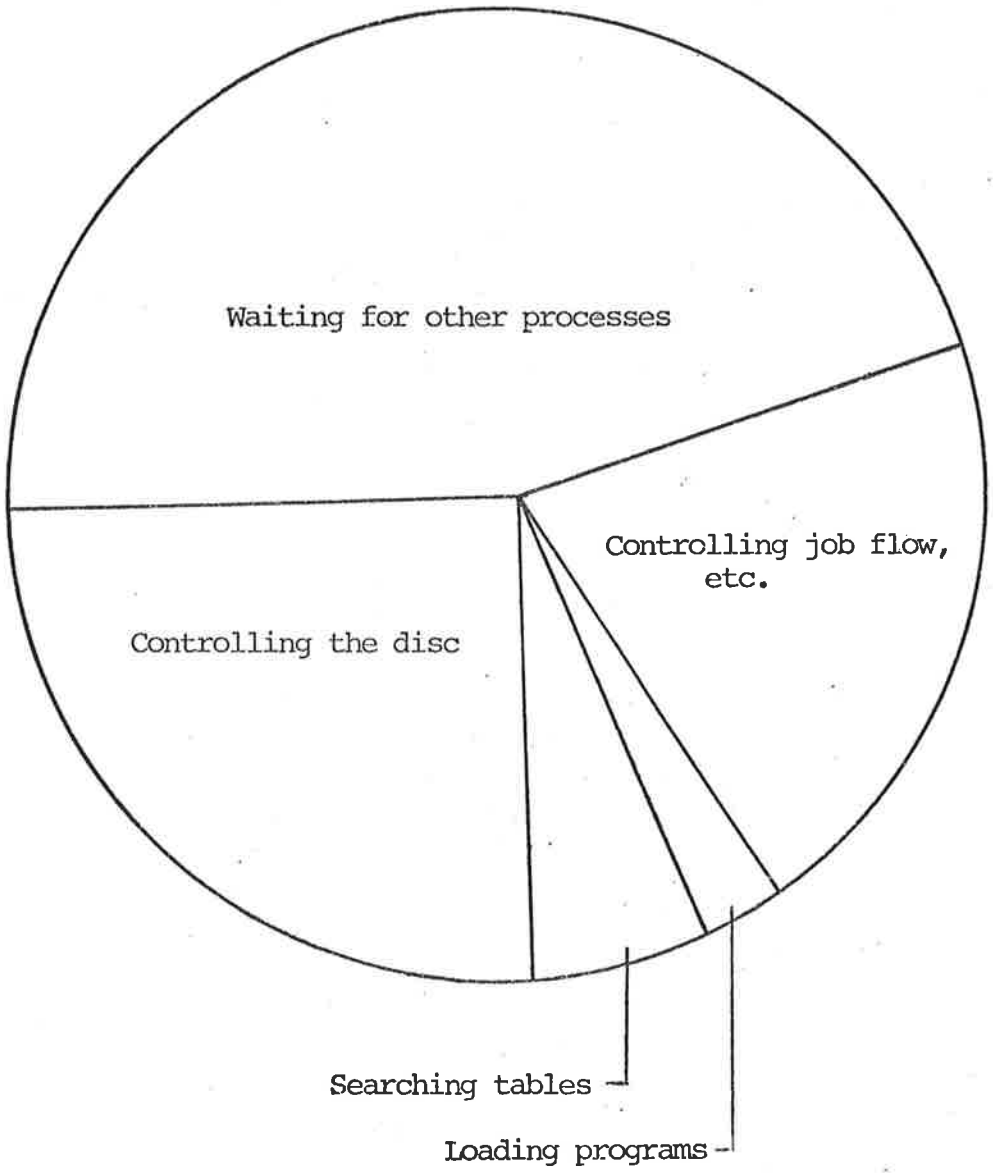


Figure 14 : The processes (in proportion to the time involved) carried out by active PPUs during execution of the test job.

5.2.6 Testing PPU Programs

A secondary application of Simulator A is the qualitative testing of PPU programs. Experiments described later (5.3.2, 6.4 and 8.2.2) need PPU programs specially written for them and these, like all programs, need to be well tested to ensure that they behave in the required manner. This is very difficult to do using a real PPU, because there is no way of monitoring the activities of a real PPU. The usual error tracing techniques (printing key variables, dumps of memory, values of registers, etc.) cannot be applied. Simulator A provide an ideal answer to this problem. If the program to be tested is simulated, then every instruction executed, every decision made and every result computed can be traced in full, and any errors can be readily detected. An additional advantage is that this testing by simulation can be done as a normal job, whereas using a real PPU would require dedication of the whole system, because a malfunctioning PPU program can easily deadlock the system.

5.3 VALIDATION OF THE DETAILED SIMULATOR

Qualitative validation of Simulator A, which is necessary if the simulator is to remain in operation at all, has been discussed already (5.1.4). Quantitative validation (i.e. validation of the timing information produced by the simulator), while not necessary for the operation of the simulator, is essential if its results are to be treated with any confidence. Quantitative validation can be achieved by a comparison of the simulation log with some similar sequence of timed

events from the real system as it runs the same test job. The technical problem posed by the validation is measuring such a sequence of events in the real system.

5.3.1 Use of Monitor Requests

Validation requires that a sequence of events in the real system be timed while the test job is running. It is to avoid just such measurement that the detailed simulator is required anyway. However, for the purposes of validation it is not necessary to have an exhaustive list of every event of interest in the system. All that is required is a subset of reasonably frequent events for comparison with the corresponding events in the simulation log. It is important that the events chosen can be measured without putting a noticeable load on the system. It would be unacceptable, for example, to incorporate timing subroutines in the programs used by the test job, since this would result in the system being significantly different from the one being simulated. The measurement must not alter the system's behaviour with regard to the test job.

Ideal candidates for measurement in the real system are the requests that the PPUs make to the monitor. They are sufficiently frequent (about 100 were issued during the processing of the test job) to provide an accurate comparison. They appear in fixed locations in central memory, and can therefore be easily detected. The nature of the request, the name of the requesting program and the PPU in which the program runs make each request uniquely identifiable. The instant at which a request is issued is well defined by the

changing of the appropriate output register, which can be detected by any PPU. The sequence of requests can be reproduced exactly by repeating the associated processing.

5.3.2 Measurement of Monitor Requests

In order to measure the times at which requests were made by PPUs to the monitor in the real system, a special PPU program was written. This program first scheduled the test job to begin after a small delay, during which it initialized itself. While the test job ran, this program continually scanned the PPU output registers (in central memory) looking for new requests. When one was found, the requesting PPU number, program name, the request code and the time were copied into a buffer in central memory. The time was computed by monitoring the system clock (see 2.1.5). This program was called the sub-monitor, because its scanning of the output registers is similar to that carried out by the monitor.

An important practical consideration in carrying out experiments with the sub-monitor was that special programming of a PPU is required. As explained in Chapter 2 (2.2.2), the stability and security of the Scope system depend largely on users not being able to programme PPUs, and so experimentation with new PPU programs is incompatible with normal system operations. Experiments involving the sub-monitor therefore required dedication of the whole computer, with the attendant difficulties of obtaining such a facility.

The times produced by the sub-monitor were subject to small but inevitable uncertainties. A monitor request could be issued at any time, but it was not detected (and timed) until the sub-monitor read the output register concerned. On rare occasions, a request was missed altogether by the sub-monitor because it was issued, processed and cleared (by the monitor) before the sub-monitor found it. The average time required by the sub-monitor for a complete scan of the output registers was about 200 micro-seconds, so that uncertainties up to this figure can be expected from its results.

The sub-monitor had almost no effect on the rest of the system. During scanning, the only resources it used were one PPU and an area in central memory. Both PPUs and central memory were in plentiful supply at all stages during the experiment, so this produced no problems. The sub-monitor did not make up any requests to other parts of the system while it was scanning.

After the test job had been run, the sub-monitor wrote its buffer (containing the results) from central memory on to a file. During this part of the experiment it could issue requests, since the measurement phase was over. A separate analysis program was used to print the results in the same form as they appeared in the simulation log.

5.3.3 First Validation Results

Figure 15 shows part of the first comparison between the times for real monitor requests (detected by the sub-monitor) and the times for the corresponding events in the simulation log. Note that Figure 15 includes one of the requests (number 12) that was missed by the sub-monitor, but which was detected during simulation. Figure 16 is a graph showing the differences between the event times of Figure 15.

It can be seen that the two sets of times, while comparing reasonably well, do not reflect the accuracy one might expect from a simulator as detailed as Simulator A. On the other hand, the differences remain fairly constant over a number of events, showing which areas in the simulation (or the validation) need closer attention. For example, something appears to have happened to have displaced all requests after the first by about 1.3 milliseconds. Requests 9 and 10 are displaced further from the rest of the list.

Further investigation proved that these errors were due, not to inaccurate simulation, but to differences in the conditions under which the test job ran in the real and simulated systems. For example, in the simulated system, the test job was the first one after dead-start to request central memory. In the real system, the job to start the sub-monitor ran before the test job. Memory assignment for the first job after dead-start takes longer than for subsequent jobs, and this resulted in the time between the first

	Request	Real Time (milliseconds)	Simulated Time (milliseconds)
1.	Memory allocation (IRA)	0.36	0.34
2.	File table interlock (IRA)	0.88	2.12
3.	Job table access (IRA)	1.40	2.62
4.	Job table access (IRA)	1.61	2.88
5.	Accounting message (IRA)	2.28	3.52
6.	Accounting message (IRA)	2.95	4.14
7.	Change control points (IRA)	7.24	8.51
8.	Stack request (IAJ)	11.80	12.81
9.	Schedule re-execution (IRA)	11.80	9.03
10.	Drop PPU (IRA)	12.06	9.30
11.	Reserve disc channel (ISP)	13.05	13.84
12.	Change control points (ISP)	missed	15.12
13.	Change control points (ISP)	48.57	49.53
14.	Accounting message (IAJ)	49.40	50.51
15.	Accounting message (IAJ)	50.53	51.96
16.	Accounting message (IAJ)	51.66	52.79
17.	Job Status Request (IAJ)	52.49	53.50
18.	Accounting message (IAJ)	53.79	54.83
19.	Stack request (IAJ)	60.38	61.49
20.	CPU assignment (IAJ)	60.75	61.94
21.	Drop PPU (IAJ)	61.11	61.94

Figure 15. (continued overleaf).

	Request	Real Time (milliseconds)	Simulated Time (milliseconds)
22.	Reserve disc channel (ISP)	61.33	62.51
23.	Change control points (ISP)	62.52	63.80

Figure 15 : First results from the validation of the detailed simulator.

(continued from previous page)

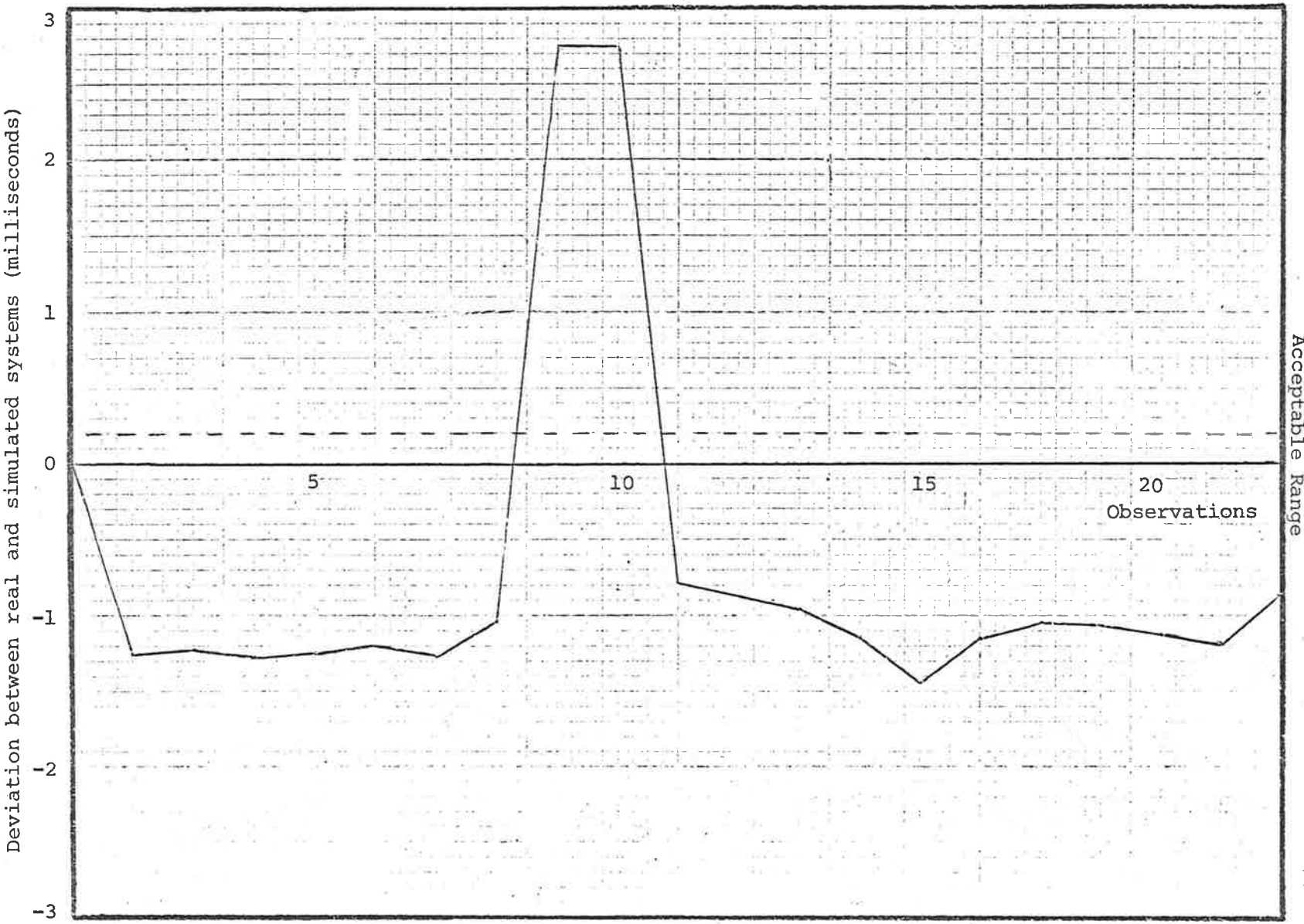


Figure 16 : Graphical representation of Figure 15.

two requests being 1.3 milliseconds longer (in the simulation) than it should have been.

In the simulated system, the idle control points were not eligible for assignment (i.e. IRA could not use them to initiate a new job, even if one was available). In the real system, due to the design of the sub-monitor, this was not the case, and so IRA executed another search for waiting jobs, delaying requests 9 and 10 by about 4 milliseconds.

5.3.4 Final Validation Results

When the discrepancies described above were overcome by simulating the relevant sections of processing under the right conditions, the results shown in Figure 17 were obtained. The differences between the real and simulated systems are plotted in Figure 18. It was mentioned above (5.3.2) that there is a lag of up to 200 microseconds between when a request is made by a PPU and when it is timed by the sub-monitor. It can be seen from Figure 18 that all the differences between the real and simulated request times fall in the interval covered by this 200 microseconds uncertainty. This is a most favourable result, as it indicates that the results of Simulator A are accurate, within the ability of the sub-monitor to test them.

The frequent disc accesses caused difficulty in the validation process. The real disc response times were not included in the simulation, but were included in the sub-monitor results. As stated

	Request	Real Time (milliseconds)	Simulated Time (milliseconds)
1.	Memory allocation (IRA)	0.36	0.34
2.	File table interlock (IRA)	0.88	0.84
3.	Job table access (IRA)	1.40	1.34
4.	Job table access (IRA)	1.61	1.60
5.	Accounting message (IRA)	2.28	2.24
6.	Accounting message (IRA)	2.95	2.86
7.	Channel control points (IRA)	7.24	7.23
8.	Stack request to read c.cards	11.80	11.66
9.	Schedule re-execution (IRA)	11.80	11.74
10.	Drop PPU (IRA)	12.06	12.02
11.	Reserve disc channel (ISP)	13.05	12.96
12.	Change control points (ISP)	missed	13.97
13.	Change control points (ISP)	48.57	48.39
14.	Accounting message (IAJ)	49.40	49.36
15.	Accounting message (IAJ)	50.53	50.38
16.	Accounting message (IAJ)	51.66	51.64
17.	Job status request (IAJ)	52.49	52.35
18.	Accounting message (IAJ)	53.79	53.68
19.	Stack request (IAJ)	60.38	60.34
20.	CPU assignment (IAJ)	60.75	60.69
21.	Drop PPU (IAJ)	61.11	60.99

Figure 17. (continued overleaf).

Request	Real Time (milliseconds)	Simulated Time (milliseconds)
22. Reserve disc channel (ISP)	61.33	61.26
23. Change control points (ISP)	62.52	62.54

Figure 17 : Final results from the validation of the detailed Simulator.

(continued from previous page)

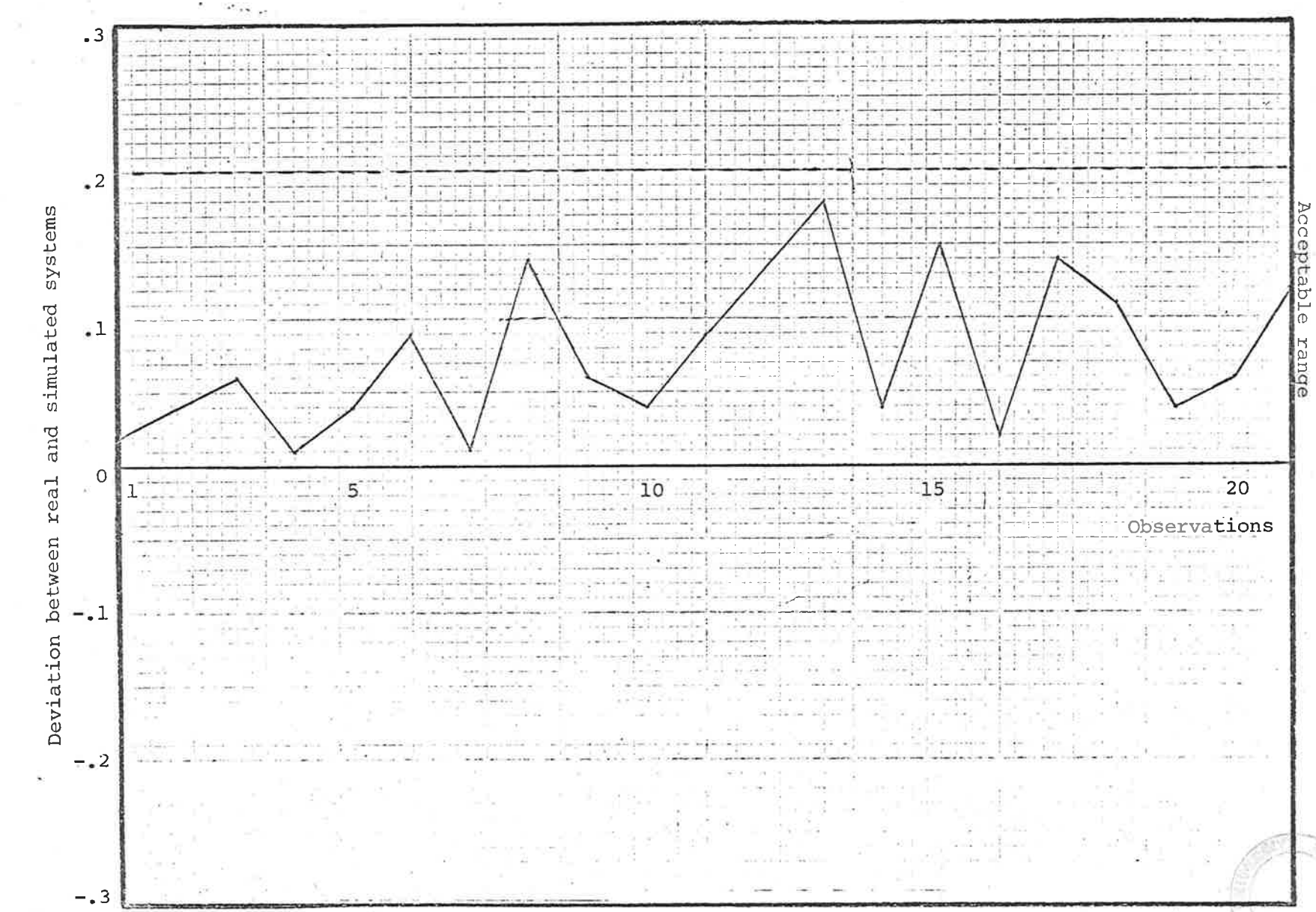


Figure 18 : Graphical representation of Figure 17

above (4.3), the results of Simulator A do not involve disc accesses, so that this does not affect its validity, but it does prevent continuous comparison of request times right through the running of the test job. The comparisons were therefore broken up into sections between disc accesses, and each section gave as good a comparison as that shown in Figure 17. At a later stage, when the disc responses had been studied, it was found, as mentioned in 3.3.1, that the times for some disc accesses could be predicted exactly, whereas others took a random time. Predictions were made for the exactly predictable accesses occuring in the test job, and there were incorporated in the simulation log. The results in Figures 17 and 18 include one of these exact disc accesses, which is responsible for the long delay between requests 12 and 13. As shown in the results, the predicted disc response times maintain the same level of accuracy as the other quantities measured.

5.4 SUMMARY

Simulation methods such as those described in this chapter provide a way of studying the internal operations of the Scope system with detail and accuracy. PPU activities are particularly difficult to observe in the real system, and instruction level simulation, while slow, can provide all the detail necessary, as well as supplying timing information. The instrinsic slowness of such a detailed simulation is alleviated to some extent by suitable design of the simulation program, permitting processes already studied to be skipped.

Collection and analysis of data from the simulator is a major (and tedious) undertaking, but this is offset by the interest of many of the results obtained (5.2.4), the unexpected nature of which justifies simulation at a detailed level. Representing the results of the simulation in a clear way is difficult, but precedence-decision networks of the type discussed in 5.2.3 fill most of the requirements. Such a representation serves to clarify the often intricate interactions of the processes making up the Scope system.

A most important need is for adequate validation of the simulator. Great confidence cannot be placed in the results of an unvalidated simulation. The good validation results obtained for Simulator A increased confidence in the quality of the model, and this more than justified the effort of designing and running the validation experiments.

The final result of this preliminary simulation study is a detailed, quantified description of the important parts of the Scope system programs. When combined with the information about input-output equipment discussed in Chapter 6, this forms an invaluable pool of data from which an accurate and efficient simulation model can be built.

CHAPTER 6

MEASUREMENT OF EQUIPMENT RESPONSE TIMES

The one aspect of system operations not covered by the detailed instruction simulator is the behaviour of the input-output devices. Study of these devices was carried out independently of the detailed simulation and comprised three stages. Firstly, the relative importance of each device to the operation of the system was considered. The information already available (in reference manuals and so on) about these devices was then studied. For some devices this information was sufficient for adequate simulation. For others the available information was not precise enough and a third stage of investigation was undertaken. This involved direct measurement of the response times of the equipment using specially written programs.

6.1 OVERVIEW OF THE INPUT-OUTPUT EQUIPMENT

The degree to which accurate simulation of a device is critical to good overall simulation varies considerably, depending on the device concerned. Many devices are so divorced from the main operations of the system, for reasons to be given, that their response times have little overall effect. Others are intimately involved in the most fundamental processes of the Scope operating system.

6.1.1 Non-critical Devices

Under Scope, all the card readers, line printers, card punches and plotters (collectively referred to as unit record devices) are managed by Janus, a subsystem of PPU programs. Janus controls its own processing by maintaining a set of scheduled processes called alarms. Each alarm comprises a process code and a time at which the corresponding process should be initiated. When the alarm time has passed, the alarm is said to have gone off. For example, suppose that a card reader and some other devices are active. There will be a number of alarms set at any given time, but suppose one for "read a card" has just gone off. Janus, by means of control messages sent to the card reader, initiates the reading of a card. It then sets an alarm for processing the data read in after a time T. T is set so that by the time the alarm goes off, the reading of the card will have finished. In the meantime Janus processes any other alarms that may go off. Processing of the card reading operation will not continue until T has expired. The time actually taken by the card reader to read the card therefore has no effect on the operation of Janus or of the system as a whole. The important time is the alarm time, T, which is pre-defined in the Janus program.

Similar methods are used by Janus to control the other unit record devices. The response times of these devices therefore do not effect the timing of other parts of the system, so long as they do not exceed the corresponding alarm times in Janus. The only

complication is that alarms can go off while another is being processed, so that the alarms queue for attention. The times involved in such a queuing process depend on the time taken to process each alarm, that is on PPU program execution times, which have already been examined using the detailed instruction simulator. The unit record equipment itself has no effect except in exceptional circumstances.

The exceptions include such things as card jams, printers running out of paper, and so on. These are not internal to the input-output devices ; they occur randomly and involve action by the operating staff. Such problems are considered in the discussion of the simulation design, in Chapter 7.

6.1.2 The 6603 Disc

In contrast to the unit record equipment, the 6603 disc is the device in closest association with the operation of Scope. The main reason for this is that the system library is stored on the disc. Except for a few constantly used PPU programs (which are stored in central memory), every program, either PPU or CPU, must be read from the disc whenever it is loaded. For example, many PPU programs call in PPU sub-programs. If the required subprogram is disc resident, the program must wait until the subprogram can be accessed and read from the disc. Thus the execution time of the program is dependent on the response time of the disc. We already know the execution times of the other parts of the program from the detailed event simulator. Ideally, then, disc access times

should be known with comparable accuracy.

6.1.3 Other Devices

The only other devices considered are magnetic tape units. Under Scope, these are controlled by PPU programs that are loaded when required. Once loaded, the programs perform the required operation, tell the requesting program that the operation is complete and finish. The time required for the operation often involves the response time of the tape unit, which in turn depends on tape transport speeds and so on. If tape units were as crucial to the operation of Scope as the disc, it would be necessary to know their response times accurately. They are peripheral, however, and the system, once initiated, can run without them. In the simulation tapes were not considered during the initial stages of construction and validation.

6.2 INFORMATION IN THE MANUFACTURER'S DOCUMENTATION

The main source of information about equipment response times is the appropriate manufacturer's manual (12). Each response time given in the manuals is found either from theoretical considerations, or from measurements on similar (but not the same) devices. Thus, the times given may vary slightly from the corresponding response times of the devices at the simulated installation. For many devices, as has been explained, the exact response times are not critical to accurate simulation, and the times given in the manuals are adequate.

The important times for the unit record devices and the magnetic tape units are given in Figure 19. The card reader and card punch operations are simple ; the only important times are those taken to read or punch a card. The line printer has a more complex set of operations, but the time to print a line is the most important. The operation of plotters is not very important as they are infrequently used. These devices are, of course, mechanical, and some degree of randomness can be expected in their response times. This random variation is not given in the manuals, but in any event it would be small enough to be covered by the design of Janus, as explained above (6.1.1).

The information supplied about disc operation times is shown in Figure 20. As was explained in 6.1.2, these times need to be known with the same order of accuracy as the program execution times. The times in Figure 20 are accurate, at best, to the nearest millisecond. This is not good enough for direct inclusion in the simulation. There is need for more accurate measurements of the performance of the disc.

6.3 OPERATION OF THE 6603-II DISC

The structure and operation of the 6603 disc have already been introduced (2.1.5 and 2.2.7). This section describes in greater detail the processes necessary to access the disc, so that the techniques used to measure its operation times (described in Section 6.4) may be more readily understood.

Card Reader :	
Read first card in a deck	71.2
Read subsequent cards	49.2
Card Punch :	
Punch a card	600.
Line Printer :	
Print line with less than 48 different characters	60.
Print line with more than 48 different characters	75.
Magnetic Tape :	
Transfer 1 data word (high density)	.016
Read involving a record mark	.20
Write involving a file mark	.043
Initiate rewinding	.015
Start tape motion	5.0

Figure 19 : Times (in milliseconds) required by various input/output devices to perform their operations.

Revolution Time (R)	67.
Transfer one sector :	
inner zone	.634
outer zone	.493
Time for disc to rotate from sector s to the reference mark, L(s) :	
inner zone	R(103-s)/103
outer zone	R(131-s)/131
Time to change head groups at sector s	L(s)
Time to change tracks at sector s	L(s)+3R

Figure 20: Times (in milliseconds) for operations on the 6603 disc, as given in the manual.

6.3.1 Structure of the Disc

The 6603 disc unit contains fourteen discs, giving a total of twenty-eight recording surfaces, twenty-four of which are used for data. These are divided into two banks of twelve, one on each side of the driving motor (see Figure 2). Information goes to and from the disc in 12-bit PPU words, and the 12 bits of a word are simultaneously written or read, one on each of the 12 surfaces of the chosen bank.

Information is written and read by heads, of which there are four for each recording surface, arranged radially as shown in Figure 2. Each of these heads has access to a ring-shaped area of the recording surface called a zone. The twelve heads occupying the same zone of the surfaces in a given bank form a head group. There are a total of eight head groups, since there are four zones and two banks. Only one head group may be used at a time, and switching between them is achieved electronically. All the heads together can move radially across the disc surfaces.

Information is arranged on the recording surfaces as shown in Figure 21. Each of the zones contains 128 tracks. The motion of the heads across the recording surface determines which of the tracks is used. Each track is made up of a number of sectors, 128 for tracks in the two outer zones and 100 for the tracks in the two inner zones. A sector holds 320 bits of information, and so

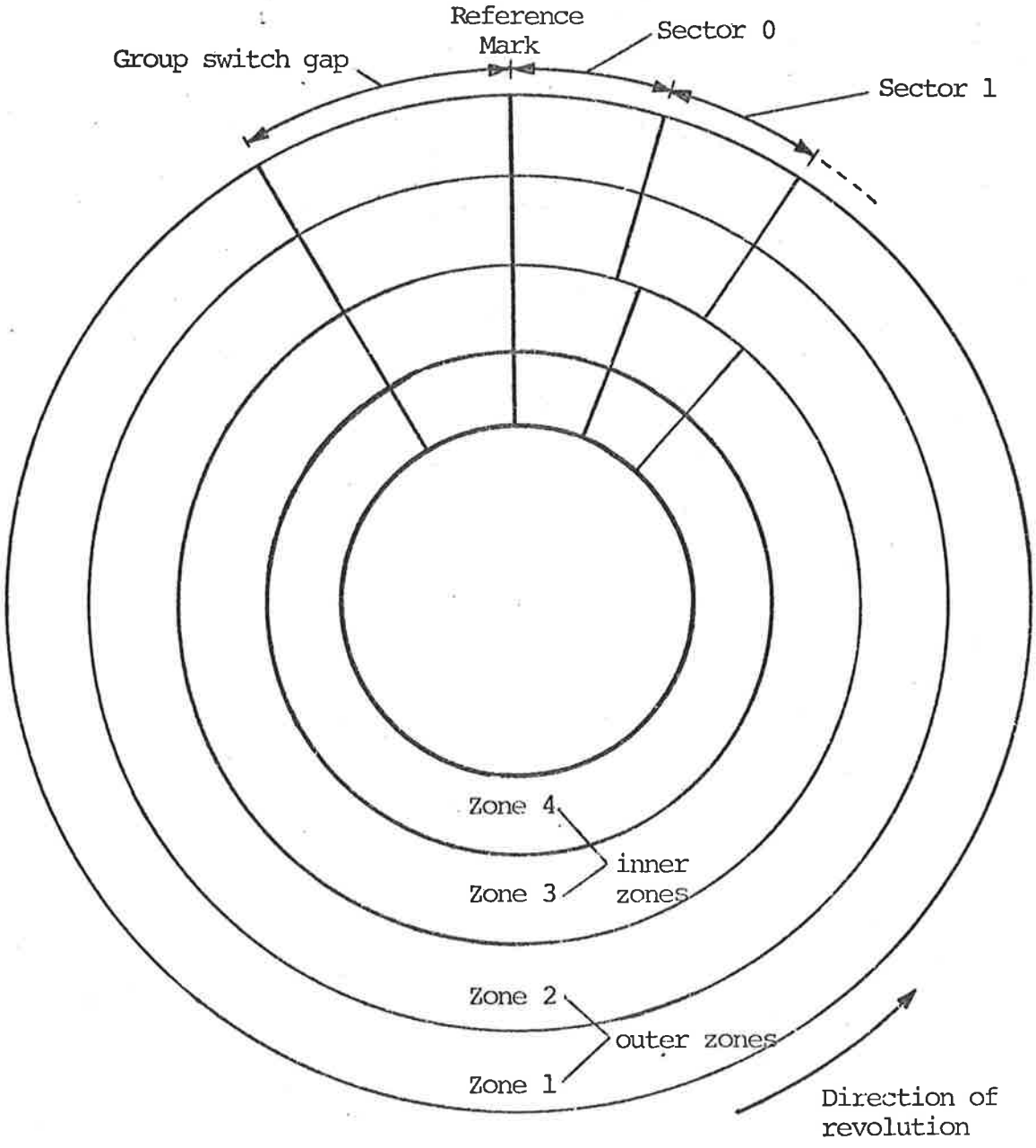


Figure 21 : A recording suface of the 6603 disc.

the 12 surfaces in a head group together hold 320 PPU words. In addition, each sector contains a number of extra words that hold positioning information. These will be referred to as "control words". They are accessible only to the PPU that controls the disc, and they can be used to assist the disc control program, as will be explained later (6.3.3). The most important of the control words contains the number of the track to which the sector belongs. A sector is the smallest part of the disc that can be accessed individually.

The disc has a reference mark at the beginning of sector 0 and there are pulse marks at sectors 24, 64 and 88. These are used by the circuitry of the disc itself for timing purposes ; they are not directly accessible to the controlling program.

6.3.2 Access to the Disc

Before information is read from or written on the disc, procedures must be carried out to ensure that the correct part of the disc is used. Three selections need to be made : selection of a head group, selection of a track within that head group, and selection of a sector within the track.

Head group selection is performed electronically, and takes about a millisecond. However, a change of head group can only be made when the heads are over the group switch gap (see Figure 21). If a command to change head groups is issued at some other point in a revolution, the issuing program must wait until the group

switch gap reaches the heads before it can continue. Thus a wait of up to a revolution is involved. Modifications made to the disc have eliminated this delay in some circumstances ; these are described in 6.3.3.

Track selection involves mechanical motion of the heads over the disc surface. The process can only be initiated when the heads are over the group switch gap, so that a delay of up to one revolution can occur (as for head group switching), depending on the position of the disc when the track selection command is issued. Once initiated, the movement takes three revolutions to complete. Track selection thus involves a wait of three to four revolutions, that is, using the revolution time given in Figure 20, 201 to 268 milliseconds. Again, modifications to the disc have changed the characteristics of this process, and these are described in paragraph 6.3.3.

Once the heads are positioned over the correct track, and the right head group has been selected, it is necessary to wait until the required sector comes under the heads. This waiting time depends on the rotation speed of the disc.

When reading or writing is underway, it takes roughly half a millisecond to transmit one sector. In practice, only one sector is ever transmitted at a time because the disc control programs use the technique of "half-tracking". Alternate sectors are used in sequence so that while the intermediate sectors are under the heads, processing of the data transmitted can take place.

The hardware of the disc can generate a status word that can be read at any time by the controlling program. This status word is not recorded on the disc surface, but is generated when required. The information contained in this word is shown in Figure 22. It includes the number of the sector currently under the heads, and bits that tell if head motion or head switching are in progress. This status word should not be confused with the control words, which are recorded on the disc surface and which can only be read when they are under the heads.

6.3.3 Disc Modifications

A number of modifications have been made to the 6603 disc since it was first designed. These enable it to work much faster, but they make its operations more complex and more difficult to simulate. As these modifications have a considerable effect on the timing characteristics of the disc, it was necessary to include them in the simulation.

The first modification concerns head group switching. Its effect is to allow switching at any point during a rotation, except when the switching is from an inner zone to an outer zone (or vice versa). Thus in most cases switching takes only about a millisecond, and it is not necessary to wait until the group switch gap is under the heads.

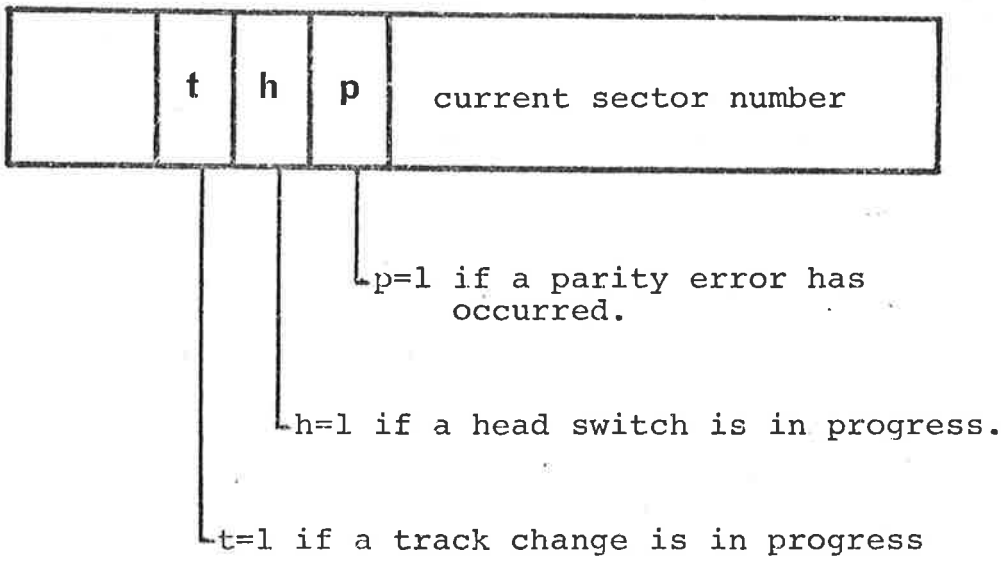


Figure 22 : The format of the disc status word.

The second modification reduces the time required for track selection. Instead of taking three revolutions plus the time to reach the switch gap initially, track selection now takes eleven timing pulses. A timing pulse occurs whenever a pulse mark (at sectors 0, 24, 64 and 88) passes the heads. Thus instead of taking 3 to 4 revolutions, the process takes $2\frac{1}{2}$ to 2 3/4 revolutions.

The two modifications mentioned so far are introduced at the same time, and a disc so modified is referred to as a 6603-I disc.

A third modification further reduces the effective time for a track change. The disc is altered so that reading and writing can take place while a track change is in progress. One of the control words of each sector, as mentioned in 6.3.1, contains the number of the track to which the sector belongs. With this modification, the program controlling the disc can therefore read the current track number while the heads are moving. When this number remains constant at the number of the wanted track, it is assumed that movement of the heads has finished, and information transfer can take place, despite the fact that the track change may not be logically complete. Logical completion, which is signalled by a change in the appropriate bit of the status word (Figure 22), still takes eleven timing pulses as explained above. The track change must be logically complete before another track change can be initiated. It is only the data transfer that can proceed sooner. A disc with this modification is called a 6603-II disc, and this is the type of disc being simulated.

Time for one revolution :	R
Number of sectors in one track (N)	
inner zone :	100
outer zone :	128
Time for n sectors to pass the heads	
T(n) :	$\frac{nr}{N+3}$
Time for switching head groups	
intra-zone :	H
inter-zone :	T(N-s+3)
Time for heads to move from one track to another :	Q
Time for logical completion of a track change (during which another track change command cannot be received) :	
$0 < s < 24$:	T(24-5) + 2R + T(64)
$25 < s < 64$:	T(64-5) + 2R + T(N-61)
$65 < s < 88$:	T(88-5) + 2R + T(N-61)
$89 < s < N$:	T(N+3-s) + 2R + T(64)

Figure 23 : Operation times for the 6603-11 Disc.

These modifications lead to a different table of times for disc operations (Figure 23). Note that the determination of the times is now a much more complex operation. Whereas previously (Figure 20), all the access times depended only on the revolution time, R, and the initial sector, s, there are now two more quantities to consider : Q (the time taken by the heads to move and stabilize) and H (the time required to switch head groups within a zone).

A most interesting aspect of these modifications is that they were only discovered by the measurements of the disc described below (6.4). The documentation originally available did not include them, although the effects of the third change (reading during track selection) on the disc control program had been observed during detailed simulation of the program. Thus the experiments to measure the performance of the disc not only improved the accuracy of quantities already known ; they also supplied further vital information about the disc's behaviour.

6.3.4 An Example of a Disc Access

The processes described above are perhaps best illustrated by an example. Suppose that the disc is positioned at track 56, head group 1, sector 74, and data are to be written on track 32, head group 6, sector 67. The operations involved in this example are illustrated in Figure 24, which shows events plotted along a time axis. This diagram will be referred to frequently in the discussion that follows.

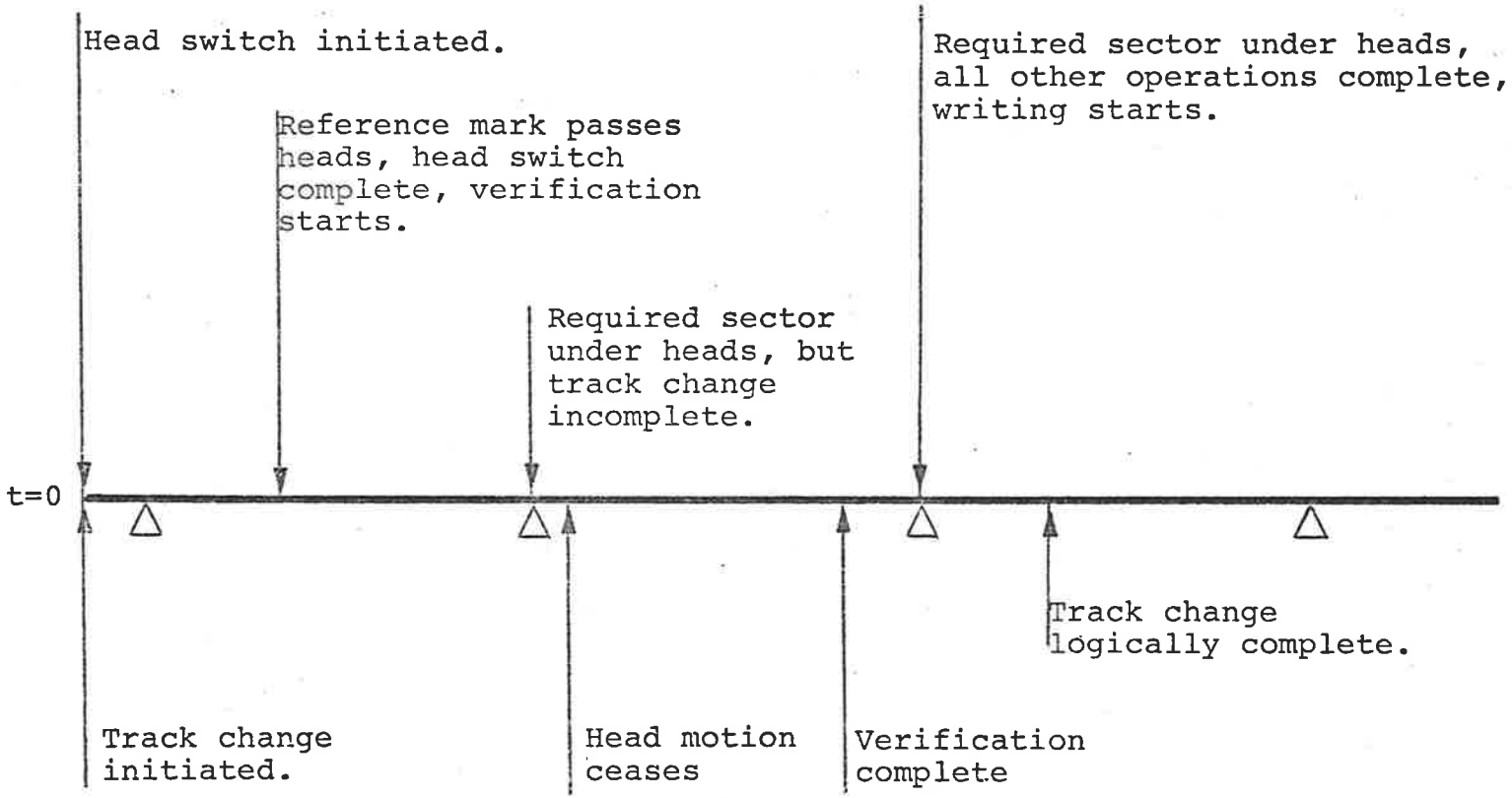


Figure 24 : The processes associated with the disc access discussed in the text.

Suppose that the disc access starts at time t=0. The first change to be initiated is the selection of the required track (number 32). A command is sent to the disc to initiate movement of the heads from track 56 to track 32. This is shown below the time axis in Figure 24. Completion of this request will take a time Q, which is unknown at present.

Immediately after sending the track change command, the program controlling the disc sends another command to switch from head group 1 to head group 6. This change is from the outer zone to the inner zone, and so can only take place when the group switch gap is under the heads. The change will be complete by the time the reference mark reaches the heads. Using the formulae of Figure 23, it can be seen that revolution from sector 74 to the reference mark, in the outer zone, takes T(N-74-3), which is .433R, where R is the disc revolution time. During this period, which is shown above the time axis in Figure 24, no further disc operation can take place.

Once the head switch is complete, at t=0.433R, the disc control program can read control words from the disc, to check for completion of the track change initiated at t=0. In order for the track change to be considered complete, the correct track number (32) must be read from 36 successive sectors. Because of the half-tracking, a successful test takes a time of T(72). The disc is now switched to the inner zone, so that this time (using Figure 23 again) is .700R. Once the heads have steadied at t=Q, therefore, a further .700R must

elapse before the track change is considered complete.

If Q < .433R, the heads will have steadied by the time checking starts, so that checking will succeed immediately. The total time required for the track change is therefore, in this case, .433R (the time at which checking starts) plus .700R (the time required for checking) or 1.133R. If, on the other hand, Q > .433R, checking will not succeed immediately, and the track change will not be considered complete until t=Q + .700R. This second situation is the one shown in Figure 24, below the time axis. In Figure 24, Q is about 1.2R.

Track selection and head switching have now been dealt with, and only sector selection remains to be considered. The sector to be written on is number 67, which will pass under the heads at a time T(67) after the reference mark. For the inner zone, T(67) is .718R. In this example, it has been shown that the reference mark first passes the heads at t=0.433R, so sector 67 will first pass them at t=.433R + .718R or 1.151R. This will happen again at intervals of one revolution, that is at t=2.151R, T=3.151R and so on. These are shown in Figure 24 by arrows below the time axis. At any of these instants, and only at these instants, writing on the required sector can commence. Note that the disc is also in the correct position at t=0.151R, but writing cannot possibly start then because the head switch is still pending.

Sector selection requires that writing starts at one of t=1.151R, t=2.151R, t=3.151R, etc. Which of these instants is the one depends on the value of Q. Writing will start at the first such instant that occurs after the track change is considered complete, that is, after t=Q + .700R. It can therefore be seen that, depending on the value of Q, writing will start at the following times :

If $< Q < .451R$, $t = 1.151R$

If $.451R < Q < 1.451R$, $t = 2.151R$

If $1.451R < Q < 2.451R$, $t = 3.151R$

If $2.451R < Q < 2.486R$, $t = 4.151R$

Note that Q cannot be greater than 2.486R. This is the time required for logical completion of the track change (i.e. for 11 pulse marks to pass the heads). Head motion will always be completed by this time. In Figure 24, Q falls within the second of the intervals above.

This example will be discussed further in 6.4.4, after the measurements of the disc response times are described.

6.4 MEASUREMENT OF 6603-11 RESPONSE TIMES

Examination of the formulae of Figure 23 shows that the time required for any disc access is determined by five quantities :

R, the revolution time of the disc

N, the number of sectors in the current zone

s, the sector currently under the heads

H, the time required for an intra-zone head switch

Q, the time required for the heads to move from one track to another.

In order to simulate a disc access properly, all these quantities need to be known accurately. Two of them, N and s, can be calculated. N is readily determined from the current zone, which is always the zone selected in the last disc access. Determination of s is more complex. Provided that the value of s is known to be s_o at some initial time t_o, then s can be calculated for any subsequent time t using the expression :

$$s = \left(s_o + \frac{(t - t_o)(N+3)}{R}\right) \bmod (N+3)$$

This expression is simply the number of sectors that have passed the heads since t_o, reduced by the modulus operation to allow for the revolution of the disc. If s > N, then the group switch gap is under the heads.

Three quantities, R, H and Q, remain. These cannot be calculated, and, if a better estimate of their values is required than that given by the manufacturer, they must be measured directly. The methods used to measure each of these quantities will now be described.

6.4.1 Measurement of Disc Revolution Time

The time taken by the disc for one revolution was the first and most important quantity measured. This quantity is used in every calculation of disc access time, so its accurate determination is vital.

In order to measure R, a PPU program was written so that direct communication with the disc was possible. Since new PPU programs cannot be introduced to the system during normal operations, for reasons given in 2.2.2, testing and running of this program was confined to after-hours periods when the whole system could be taken over. The program, in fact, required little testing at this stage, because its correct operation had been assured by running it with Simulator A, as described in 5.2.6, and watching its behaviour.

The operation of the program is shown in Figure 25. The position of the disc was obtained at frequent intervals by inspecting the disc status word (Figure 22). At the same time a microsecond clock was maintained, as for the submonitor described in 5.3.2. Whenever the sector number in the disc status word became zero (indicating that the reference mark had just passed the heads), the time was recorded in a buffer in central memory. After 2000 revolutions had been timed in this way, the PPU program finished and started a CPU program which analysed the results. This was repeated a number of times.

The results of one run of this program are shown in Figure 26. From the operation of the measuring program (Figure 25), it can be seen that there is a delay between when the reference mark passes the heads and when the program detects that it has done so. This delay can be as long as one inner loop of the program (in the event that the reference mark passes the heads immediately after the sector number is read). This maximum delay is about 50 microseconds, and

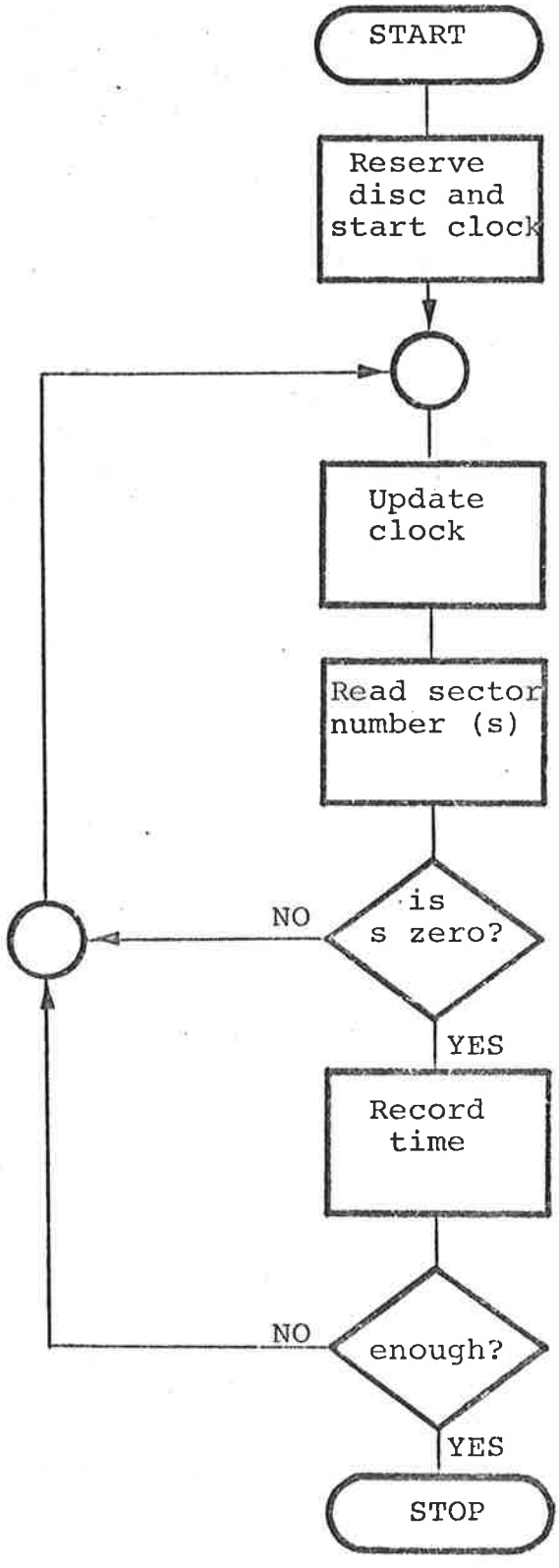


Figure 25 : The operation of the program to measure the revolution time of the disc.

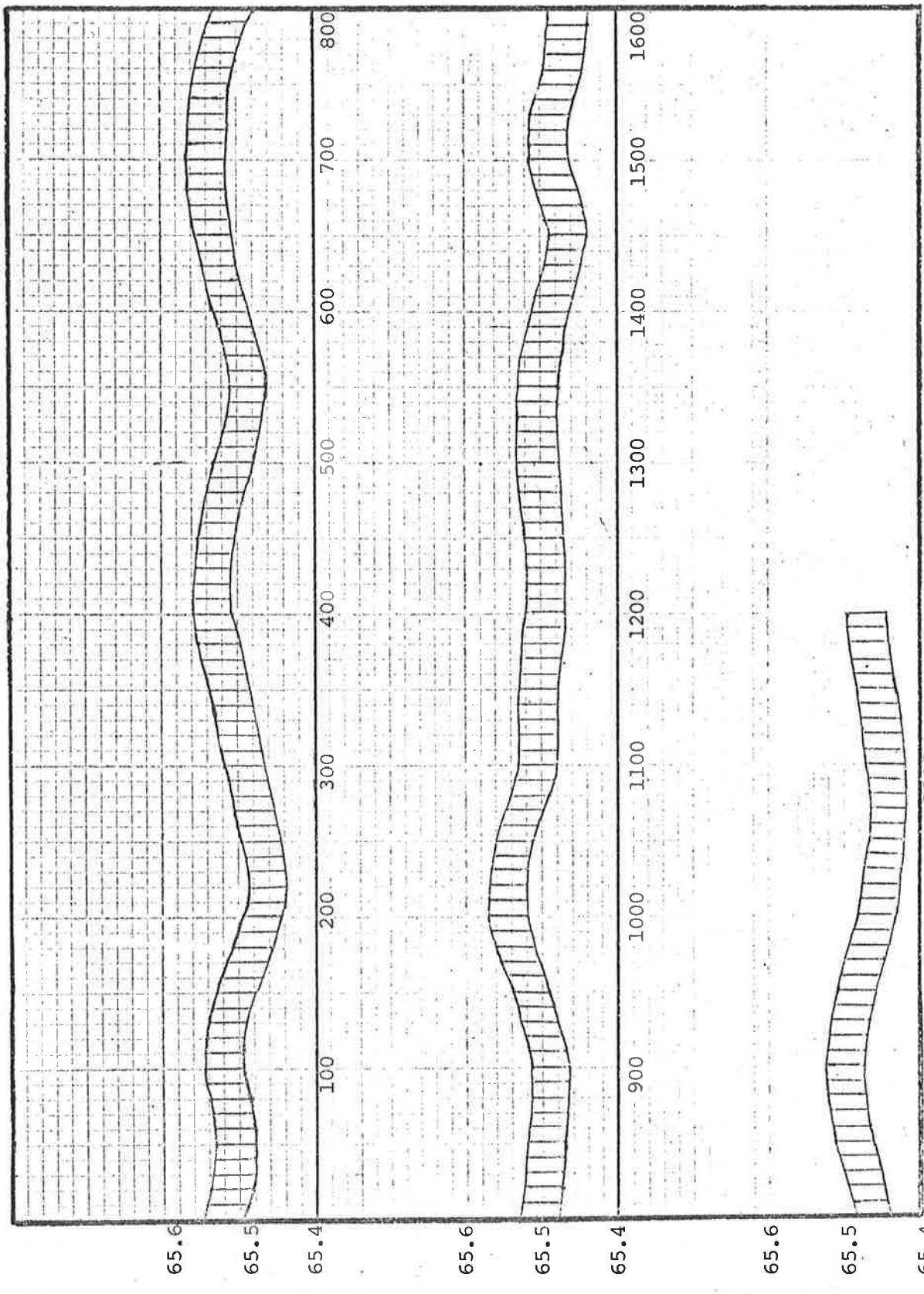


Figure 26 : Disc revolution times for 2000 consecutive revolutions

the recorded times are thus spread into a band about 50 microseconds wide, which is shown in Figure 26. It can be seen that, during the 2000 revolutions measured, the band moved little from its mean position (the maximum deviation was 40 microseconds). This indicates that the disc revolution speed is, for all practical purposes, constant. It is important that this fact was verified by the experiment, because the constancy of R is assumed by the formulae of Figure 23. The mean value obtained for R was :

$$R = 65.50 \pm 0.02 \text{ milliseconds.}$$

Note the difference of 1.5 milliseconds from the value given in Figure 20.

6.4.2 Head Switching Time

Measurement of H, the time required to switch from one head group to another within the same zone, was effected by a program similar to that used for measuring R. This program similarly maintained a microsecond clock. It issued a request for a head switch, and then continually inspected the disc status word (Figure 22) until the flag indicating that the switch was in progress had cleared. The elapsed time was recorded for later analysis. About 200 such measurements produced an average value for H of about 1.105 milliseconds, with a maximum deviation of .003 milliseconds. These results had to be adjusted to allow for the delay between the completion of the group switch, and the detection of this completion by the program. This delay had a possible range of 0 to 10 microseconds. The final value obtained was thus :

$$H = 1.100 \pm 0.005 \text{ milliseconds.}$$

As switching from one head group to another in the same zone is a wholly electronic process, one would expect is duration to be almost constant. This is ineed shown by the above result.

6.4.3 Measurement of Track-Change Times

The only information available about Q, the time required for the heads to move from one track to another, was that it depends on the distance that the heads must travel, and that it varies between 60 and 160 milliseconds.

In order to measure Q, a program was written to initiate a track change, and then to check for its completion by the same method as used by the disc control program in the operating system. A counter was set at 36. Sectors were read from the disc continually while the heads were moving. After each sector was read, the control word containing the track number of the sector was compared with the number of the destination track. If they were different, it was reset to 36. When the counter reached zero, the track change was regarded as being complete. Figure 27 shows the operation of the program in flowchart form.

The program measured the time between the initiation of the head movement and the completion of the track change. The corresponding value of Q was this time less the time required for the verification, T(72). The program obtained a value of Q for changes from track a to track b, where a and b each took the values 0, 8,

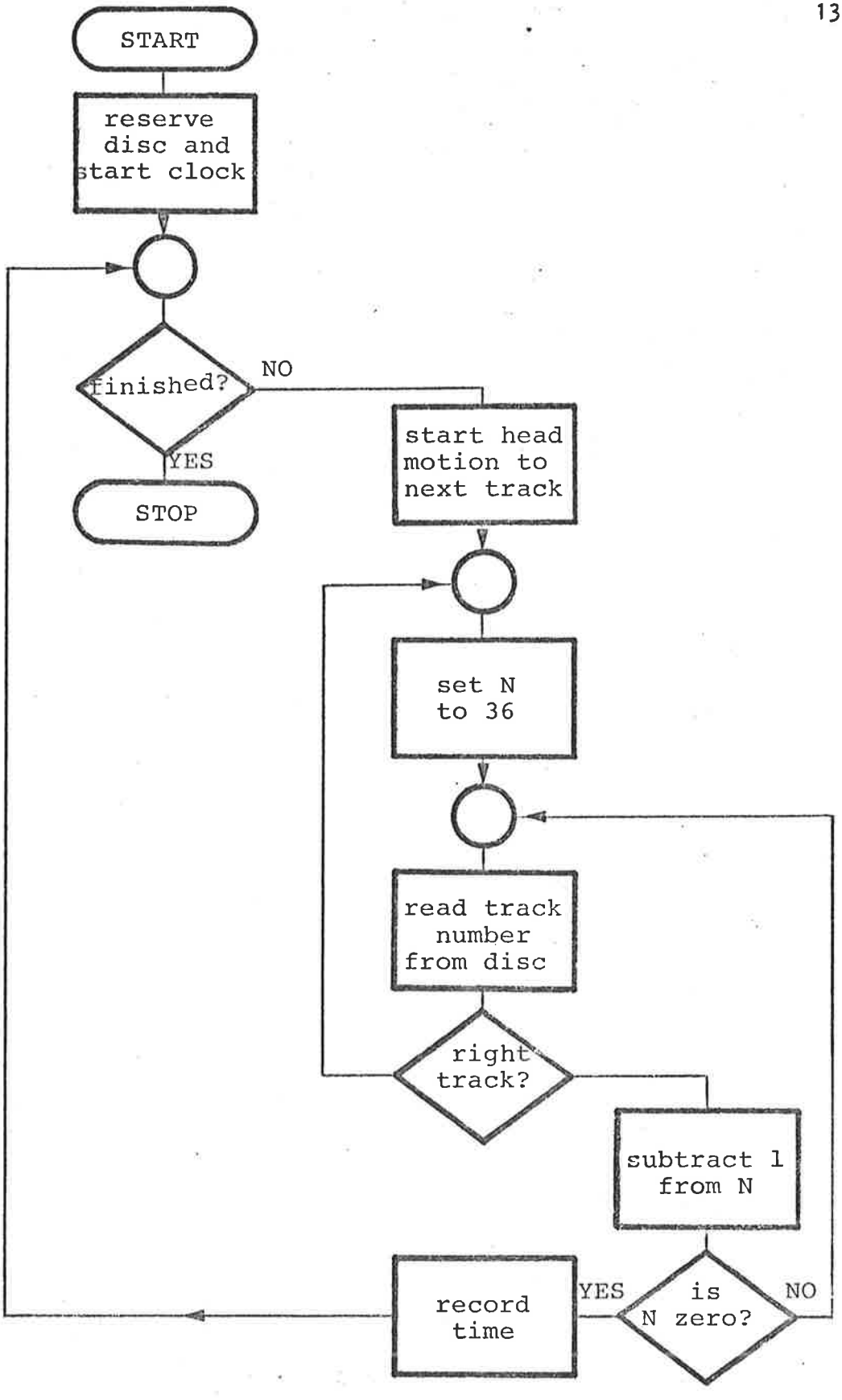


Figure 27 : The operation of the program to measure the time required for a track change.

16, ..., 120. The results of two sets of measurements are shown in Figure 28.

The most obvious characteristic of the results of Figure 28 is that they vary widely from one track change to another. Even two changes with the same start and destination tracks sometimes show considerable variation (e.g. 52 and 62 milliseconds for changes from track 60_8 to track 50_8). There is no obvious relationship between the distance the heads travel and the time taken. The only obvious factor in the results is the difference between changes that pass from one half of a zone to the other (across the boundary between tracks 77_8 and 100_8) and those that do not. Changes in the former category take a considerably longer time then those in the latter.

Regression analysis was applied to the results of Figure 28 to determine if there was any relationship between the distance between tracks and the time for head movement. The distance between tracks was defined as the difference between the track numbers, and will be referred to as x. For track changes within a half-zone, the following result was obtained :

$$Q = 67.64 - 0.16x + .015x^2 \text{ (S.D. = 9.1) milliseconds.}$$

For track changes crossing the boundary between tracks 77_8 and 100_8, the following result was obtained :

$$Q = 139.24 \text{ (S.D. = 9.0) milliseconds.}$$

The distributions of times for the two class of track changes are approximately normal.

START TRACK	DESTINATION															
	000	010	020	030	040	050	060	070	100	110	120	130	140	150	160	170
000	0	56	58	68	78	90	111	125	138	133	130	131	130	136	141	140
	0	52	62	66	79	91	102	113	127	140	131	130	147	132	138	137
010	83	0	86	58	81	80	91	112	130	125	136	138	131	136	135	138
	83	0	91	63	79	80	100	96	133	140	127	134	130	137	136	136
020	64	58	0	55	78	65	78	84	135	130	128	136	127	129	136	145
	65	58	0	52	79	68	81	89	143	138	117	125	130	129	134	129
030	78	62	76	0	80	79	79	78	130	139	130	129	136	143	142	137
	78	64	77	0	77	80	79	79	137	132	120	117	127	132	129	130
040	90	86	81	86	0	55	60	64	136	142	143	136	135	137	139	131
	88	86	83	87	0	53	63	66	141	136	136	130	129	136	133	127
050	94	79	81	83	74	0	83	61	138	140	140	138	127	138	139	144
	96	81	82	85	75	0	78	64	142	130	137	134	129	133	129	130
060	94	79	73	85	62	52	0	56	135	138	130	138	139	133	137	138
	92	80	81	85	62	62	0	53	145	130	126	133	130	130	116	126
070	97	82	86	73	71	61	72	0	136	143	140	139	141	135	131	134
	98	81	86	76	72	62	74	0	149	130	131	125	129	130	117	120
100	145	151	143	150	147	148	145	151	0	56	58	68	70	83	111	114
	147	151	146	150	147	148	148	152	0	52	63	57	70	83	109	112
110	151	144	154	143	150	145	150	149	72	0	80	59	81	70	101	109
	153	146	156	147	148	147	148	147	75	0	73	62	76	70	102	112
120	147	151	143	150	145	150	145	148	62	59	0	54	80	70	69	84
	148	153	145	150	147	150	146	150	64	52	0	52	80	69	80	91
130	151	147	150	144	148	144	149	144	70	64	72	0	82	80	77	71
	152	149	151	144	147	147	149	145	71	64	71	0	80	81	76	75
140	157	157	157	156	144	149	144	150	81	85	82	85	0	55	57	65
	151	156	151	157	146	149	144	152	81	85	83	85	0	53	59	64
150	159	155	158	154	150	144	151	143	78	71	80	79	69	0	76	57
	158	150	159	152	148	147	151	144	79	73	80	81	70	0	71	59
160	159	157	154	156	145	150	144	149	81	77	73	80	60	51	0	54
	153	156	149	155	146	150	144	148	82	77	75	83	62	52	0	54
170	156	159	151	154	148	145	147	147	86	84	75	67	62	67	69	0
	157	155	154	150	150	145	149	142	101	89	86	70	69	59	68	0

Figure 28 : Track change times for various start and destination tracks. Two measurements are shown for each change. The times are in milliseconds, and do not include the time required for verification. The track numbers are in octal.

6.4.4 The Significance of the Disc Response Times

The values of R and H determined above (6.4.1 and 6.4.2) show such small variations from one measurement to the next that they can, for all practical purposes, be regarded as constants. The values of Q, on the other hand, show appreciable random fluctuation. Q is a stochastic variable and is, in fact, the only significantly stochastic quantity involved in the operations of either the processors of the 6400 or the disc. It is the only quantity in the system discussed so far that cannot be accurately predicted.

The fact that Q is subject to random fluctuations does not mean that every disc access involves unpredictable delays. A considerable number of disc accesses do not involve track changes, and are thus independent of Q. Even for accesses that do involve a track change, the possible variation in Q is often unimportant, because of the need to access a particular sector. The sector to be read or written is under the heads only at definitely fixed instants, one revolution apart. Although the track change has a random duration, Q, the disc access as a whole cannot proceed until one of these instants, which are not random (because R is not random). The uncertainty in Q becomes an uncertainty about which of two exactly predictable delays occur.

This effect can be seen in the example discussed above (6.3.4 and Figure 24). The results of 6.3.4 were expressed in terms of R. If the value of R, which is now known, is substituted, the following possibilities can be calculated for the time taken for the disc

access :

If $Q < 29.54$, then $t = 75.39$

If $29.54 < Q < 95.04$, then $t = 140.89$

If $95.04 < Q < 159.23$, then $t = 206.39$

If $159.23 < Q < 162.83$, then $t = 271.89$

All the times are now in milliseconds. From the results of 6.4.3, it can be found that for a change from track 56 to track 32 (the change involved in the example), the expected value of Q is 76.66 milliseconds, which is within the second of the intervals above. Assuming that Q is normally distributed, with a standard deviation of 9.1, the following probabilities will apply to the possible times for the disc access :

A probability of $\sim 10^{-4}\%$ that $t = 75.39$

A probability of 97.7% that $t = 140.89$

A probability of 2.3% that $t = 206.39$

A probability of $\sim 10^{-10}\%$ that $t = 271.89$

The only values with any appreciable probability are the second and third. The continuous variation of Q has thus been reduced to a choice between two deterministic possibilities.

The concepts considered in this section assumed great importance in the simulation of disc accesses (described in 8.3.1) and in the approaches taken to validation of the simulator (8.4). They are discussed again in Chapter 8, in connection with these topics.

6.5 SUMMARY

The results obtained from Simulator A are accurate to within about 10 microseconds. In order to maintain a proper balance, the response times of peripheral devices should be known with comparable accuracy, where they are significant in the timing of system operations.

The only available source of information about equipment response times is the manufacturer's documentation. This cannot be expected to be completely accurate or definitive for a particular item of equipment. However, many devices, such as those controlled by Janus, are so shielded from the rest of the system, from the timing point of view, that the values given by the manufacturer are adequate.

The only device for which it was necessary to undertake extensive experimental measurement was the 6603-11 disc. These measurements were necessary because the information available about the disc was not sufficiently accurate, considering the vital role it plays in the operation of the Scope system.

Measurement of the disc response times produced a number of significant results. Firstly it made known such quantities as the revolution time and head switching time with a much greater degree of accuracy than would otherwise have been available. Secondly, it revealed the details and unexpected complexities introduced by the modifications described in 6.3.3. It also showed that some response times (revolution time and head switching time) could be predicted with an accuracy comparable with that already obtained for program execution times (from Simulator A), whereas track changes have stochastic response times that can only be predicted

within a certain range. Finally, analysis of the processes involved in a disc access showed how this stochastic effect is greatly modified by the need to access a particular sector.

These results had a significant effect on the development of the simulation model. The methods chosen from the validation of the model were, to a large extent, determined on the basis of the experimental results presented in this chapter.

CHAPTER 7

DESIGN AND CONSTRUCTION OF THE EVENT SIMULATOR

The culmination of the experiments and measurements described in the last three chapters is the development of the event simulator (Simulator B). The information about system software obtained from simulator A and the information about peripheral devices discussed in Chapter 6 are brought together, as shown in Figure 4, to form Simulator B. This chapter describes the overall design of Simulator B and the techniques used in writing it, and discusses some of the problems that were encountered and their solution.

7.1 BASIC DESIGN CONSIDERATIONS

In order to understand the basic requirements for the design of Simulator B, some knowledge of its intended purpose is necessary. The simulator is intended for practical studies of the Scope operating system, as it behaves on the CDC 6400 computer. The main requirements are that it be reasonably fast, that means of specifying information to it (about the system and about the job stream) be simple but flexible, and that the program itself be written so that it is easy to understand and easy to modify.

The speed of the simulator needs to be such that an appreciable amount of processing can be simulated at reasonable cost. Considering the high level of detail at which the event simulator works, a speed of about 30 times the real system (i.e. simulating half an hour of normal processing

in about a minute of simulator execution time) was considered a reasonable goal. At the installation being simulated, the longest period of interest is the normal operating day, about 12 hours, which would thus involve a 24 minute simulation run. In most instances, a much shorter run would suffice to show the effects of a given modification.

The ease with which the simuator is used depends to a large extent on the methods by which information is introduced to it. It is necessary to specify to the simulator what kinds of jobs are being submitted to the simulated system, and what pieces of equipment and software are present in the computer configuration being simulated. The design of the method used to specify this information requires careful thought.

The effect of a change in the logic of the operating system would normally be reflected in the simulator by a corresponding change in the simulator program. Such a change would be necessary in either of two cases : if it was desired to investigate the effects of a proposed change, or if it was desired to update the simulator to reflect a later version of Scope. In the former case, the change would usually be temporary ; in the latter case it would be permanent. To facilitate these changes, the simulation program needs to be written in such a way that it is easily understandable, and so that the relevant section can be readily located and altered without having to search through large sections of irrelevant program. This suggests a modular approach to the writing of the simulator.

7.1.1 Choice of Language

The simulation languages available at the time of writing Simulator B were Simscript (45) and Simula (17). Simscript was not considered as a candidate for reasons of efficiency. The choice was thus between Simula, and the most efficient non-simulation language on the 6400, Fortran.

Simula would provide a natural and concise means of describing the processes being simulated. When Simulator B was being designed, however, the Simula compiler available was new and unreliable. Documentation of the language and of the compiler was inadequate. On the other hand, Fortran, while not a language in which simulation programs can be written very naturally, was very well known, and had a thoroughly reliable compiler. It was decided that the advantages of Fortran outweighed the inconvenience of using it, so that Fortran was the language chosen for Simulator B.

In order to minimize the inconvenience of writing simulation procedures in Fortran, the simulation program used a package of subroutines to do the work of scheduling, queue management and other simulation tasks. In this way, the disadvantages of using a non-simulation language were largely offset. The simulation package, which is further described in Section 7.2, enabled Fortran to be used, in effect, as a simulation language.

7.1.2 Choice of Techniques

There are a number of techniques that can be used as the basis for a simulation project. The most popular described in (5) are event-based, process-based and entity-based simulations. If a simulation language is used, the technique used is normally pre-determined (for example, Simula uses the process-based approach). Since Fortran had been selected for this project, however, the choice of technique was open.

The event-based approach was chosen because the concepts of the system acquired from the experiments previously described were largely event-based. For example, the network in Figure 13 is made up of events linked by inter-event periods of known duration. Implementation of this kind of structure as an event-based simulation model is conceptually a relatively simple process.

7.1.3 The External Point of View

A most important aspect of the design of the simulator is its interface with the user, that is, the methods used to specify information to it and to obtain the corresponding results. Input to the simulator is in two sections. The simpler of these contains information needed to control a particular simulation run. This input takes the form of commands to the simulator, each of which takes effect at a particular moment of simulated time. A typical sequence of such commands can be seen in Figure 29, which will be examined in detail below (7.3.1), where the range of commands available and a description

Time	Command	
	TRACE INPUT TRACE JOB FLOW TRACE JANUS PPLIB	
1RA 0011 045 03 * 1EJ 0011 050 14 1SP 0007 010 12 * END		PPLIB definitions
	CPLIB	
REWIND 0015 024 02 .0055 1 C10 02 ** .005 COPY 0015 026 42 .1 2 C10 00 ** .075 .01 C10 01 ** .075 .01 END		CPLIB definitions
10. 15. 60.	DISPLAY UNTRACE JANUS TRACE STACK TRACE DISC END OF SIMULATION	

Figure 29 : An example of the simulation control stream containing definitions for the CPU and PPU program libraries.

of their effects are presented.

The other input section is for specification of the jobs being fed into the simulated system. Design of this section of the input involved a job specification language, and is discussed more fully in paragraph 7.3.3. Briefly, jobs are described by data cards resembling Scope control cards. These specify the type of processing required (compilation, loading, execution, copying, etc.), the files concerned with the process, and some indication of the amount of processing involved.

Output from the simulator is of three kinds : traces, displays and reports. Tracing gives information at an intricate level, so that each segment of processing can be studied. Displays present an overall picture of everything going on in the system at a particular instant. Reports list accumulated statistics about the performance of the system. These results are described further, with examples, below (7.3.4 and 7.3.5).

Flexibility is an important consideration in the design of the input and output facilities of the simulator. The specification of jobs and the reporting of their processing can both be done at widely varying levels of detail.

7.2 SIMULATION SUBROUTINES

As stated above (7.1.1), a large proportion of the bookkeeping tasks associated with the simulation was managed by a package of subroutines. This section describes these subroutines and shows how they enable Fortran to be used as a simulation language, though not, of course, with the same power of flexibility of a real simulation language such as Simula.

The use of packages of subroutines to "extend" Fortran in this way is a common technique and has been applied to simulation (11, 37). The subroutines described here were developed to answer the needs of this particular simulation project. They are not restricted to computer simulation, however ; they can be (and have been) used for many different types of discrete event simulation.

7.2.1 The Event List and the Sequencer

The timing aspect of the simulation is controlled by a list of scheduled events. The method is essentially the same as that described by Macdougall (43). Each entry on the event list has three attributes : the number of the event scheduled, the job with which the event is associated, and the time at which the event is to be simulated.

A simulation proceeds, events are added to the event list by the scheduler as described below. There must always be at least one event on the list, or simulation cannot continue. The simulation process is controlled by the sequencer, which is responsible for

initiating each event at the time for which it is scheduled. The sequencer operates as shown in Figure 30. The event list is scanned and the earliest scheduled event found. The simulated time is set equal to the time for which this event is scheduled, and the event is removed from the list. A call is made to a subroutine EVENT, which, given the event number and the associated job number, performs the processing necessary for actually simulating the event. When simulation of the event is completed, EVENT returns control to the sequencer, and the process is repeated. Simulation is terminated by an input command to the simulator, as described in paragraph 7.3.1.

Note that the subroutine EVENT is the place where the operations of the processes to be simulated are described. It is not one of the subroutines of the simulation package, which are independent of what is being simulated.

7.2.2 Event Scheduling

An event is scheduled by adding it to the event list. This is done by a subroutine SCHEDULE, which accepts the three attributes of the event (event number, job and time) and places them in the list. A check is made to ensure that the event is not being scheduled to occur at a time in the (simulated) past, in which case it could not be simulated at the correct time. In a correct simulation program, this would never happen, but the check was found useful during the development of the simulator.

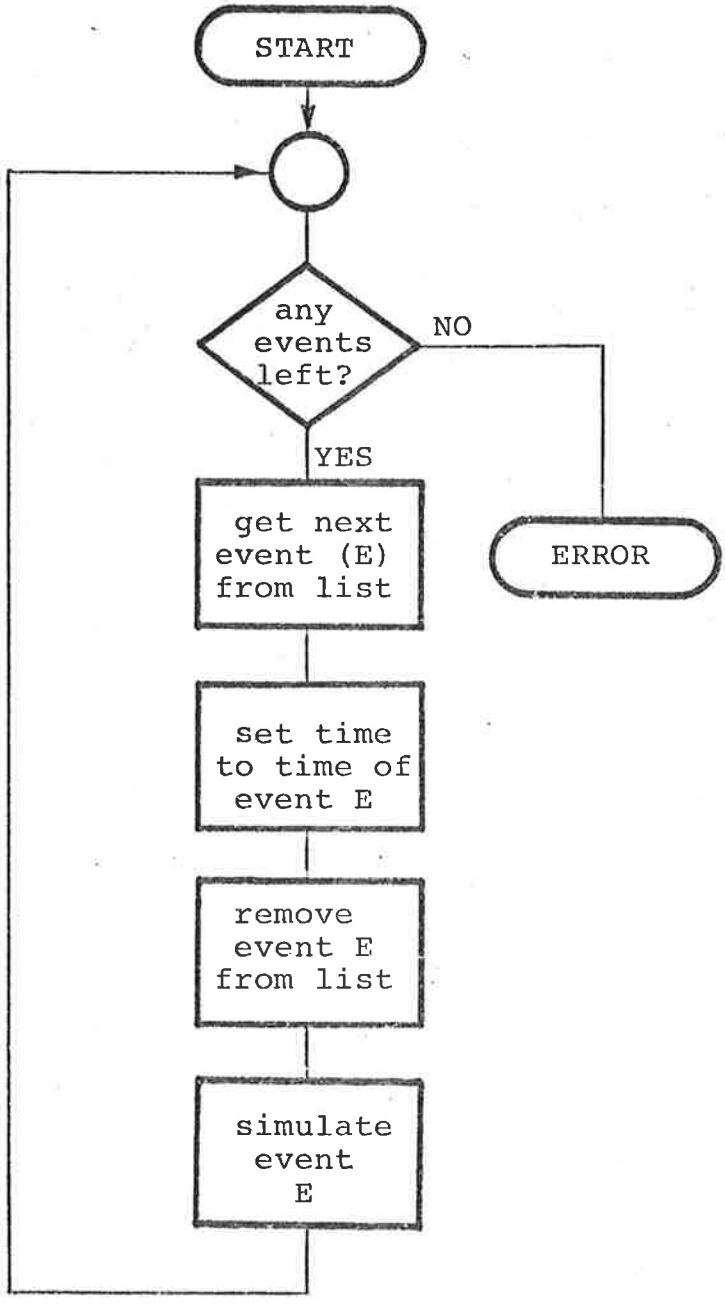


Figure 30 : The operation of the event sequencer.

Simulation is started by scheduling the first event, and then transferring control to the sequencer. Thereafter, events are normally scheduled by other events as they are simulated. For example, simulation of event number 4, to assign a PPU to a job, will schedule whatever event represents the processing to be done by the PPU when it is assigned. The event will be scheduled at the current time plus the time required to assign the PPU and to load the required program into it. This process is described in more detail below (7.4).

7.2.3 Queue Management

Management of queues is an essential part of simulating an operating system. The queuing subroutines in the simulation package provides the necessary processing to inspect the items on a queue, to add an item, to remove a selected item, to retrieve the length of a queue and to maintain statistics about the distribution (with regard to time) of the length of a queue. This processing is available for up to 10 queues.

An example will serve to illustrate the operation of the queuing subroutine. Suppose that item x (which will usually be a job number) is to be added to queue k, which at present has N entries on it. A call of the form :

```
CALL QUEUE (2,k,x)
```

is made, where the "2" is a key specifying that an item is to be added to the queue. Item x is added to queue k, the queue length is

incremented (to N+1) and the time T_1, is recorded. Suppose at some later time T_2, an item y is to be removed from queue k according to some priority scheme. Each item in the queue is inspected so that the appropriate one can be selected. The ith item is obtained by a call :

```
CALL QUEUE (1,k,z,i)
```

where the item is returned in z. When y has been chosen from all such items z, it is deleted by the call :

```
CALL QUEUE (3,k,y)
```

Item y is removed from the queue and the queue length is decremented (to N). The time at which the queue length last changed (T_1) is subtracted from the current time (T_2) and the difference is added to the accumulated time for which the queue has had length N+1. Finally, T_2 is recorded as the time of the queue length last changed.

It would have been possible to further "automate" the queuing process by implementing various common queuing policies and building them in to the queuing subroutine. The queuing policies used by Scope are not usually simple, however, and the same policy is seldom repeated in different parts of the system. The process of selecting items from queues was therefore included in the simulation proper, not in the queuing subroutine.

Queues are used in the simulation to represent a number of entities in the Scope system. These include the input and output

queues, jobs waiting from the CPU or a PPU, jobs waiting for an increase in memory allocation, the disc request stack, and jobs awaiting the allocation of a device or a channel. Each of these represented by a separate queue.

7.2.4 Resource Management

Another important and frequently used subroutine manages the allocation of resources. Its main task is to keep note of which resources (processes, input-output equipment, channels and control points) are assigned to which jobs. It maintains statistics concerning the length of time for which each resource is assigned to a job. The subroutine is called either to assign or to release a resource. It performs a number of checks on the processes it is asked to perform, and detects erroneous situations such as the release of an unassigned resource, or the assignment of one that is already assigned to another job.

7.2.5 Error Diagnostics

During the development of the simulation program, it is desirable for the program itself to be able to detect errors in itself and to diagnose them in a sensible way. Several such errors have already been mentioned : an empty event list, attempting to release an unassigned resource, etc. The event simulation procedures (in subroutine EVENT) can also detect errors. For example, if a job on the output queue issues a request for the CPU, something is wrong, and this can be detected by the program segment that simulates CPU

```
EVENT LIST OVERFLOW
EMPTY EVENT LIST
EVENT SCHEDULED IN THE PAST ***
INVALID QUEUE OPERATION *
OVERFLOW ON QUEUE NUMBER **
RESOURCE ASSIGNMENT LIST HAS OVERFLOWED
RELEASE OF UNASSIGNED RESOURCE ****
UNRECOGNIZABLE COMMAND *******
INVALID TRACE PARAMETER *******
TOO MUCH INFORMATION ABOUT JOB *******
END OF FILE WITHIN JOB *******
INVALID EVENT NUMBER **
UNDEFINED PPU PROGRAM CALL ***
DISC RESERVATION CONFLICT ****
CPU ASSIGNMENT CONFLICT
INPUT QUEUE CONFLICT
MISSING INPUT FILE *******
UNRECOGNIZABLE CONTROL CARD *******
OUTPUT FILE CONFLICT *******
NO JANUS ENTRY FOR JOB *******
```

Figure 31 : Some of the error messages provided by the simulator. Asterisks indicate variable fields.

allocation.

A subroutine ERROR is invoked whenever incorrect simulation is detected. This prints a suitable diagnostic message, and determines whether the error is sufficiently serious to stop simulation. Some indication of the range and kind of errors detectable is shown in Figure 31.

When the simulation program is running correctly, subroutine ERROR should never be invoked, except for errors in the input to the simulator. All the diagnostic facilities are retained in the production version of the simulator, however, since as previously described, using the simulator can involve changing the program, in which case errors could be introduced.

7.3 INPUT AND OUTPUT

The methods used by the simulator to communicate with the user were introduced above (7.1.3). This section describes in greater detail the facilities available.

7.3.1 Control of the Simulation

One of the input streams read by the simulator is a set of commands controlling the simulation run. Each command comprises an invocation time, a keyword, and optional parameters. If the invocation time is omitted, the time of the previous command is used. The command is obeyed when the simulated time reaches the invocation time. The commands must appear in order of increasing invocation time.

The keyword in each command describes the action to be performed. The recognized keywords are :

PPLIB — indicates that alterations to the PPU program library of the simulated system follow. The use of this command is described in 7.3.2 below.

CPLIB — indicates that alterations to the CPU program library follow. This command is also discussed in detail in 7.3.2.

EQUIPMENT — defines the equipment configuration. The parameters define the number of available printers, readers, etc.

SPEEDS — defines the speeds of the input-output devices. The parameters include the disc revolution speed, card reading rates, and so on.

DISC — defines the position of the 6603-11 disc at the start of the simulation run. The track, head group, and rotational position can be specified.

The commands listed so far would normally be used only at the beginning of a simulation run (unless, for example, it was desired to investigate the effects of, say, adding another line printer to the running system, in which case an EQUIPMENT command would be used part-way through a run). The following commands are used at any time during the run :

TRACE — start tracing a particular aspect of simulated processing. The tracing facilities are described in 7.3.4 below.

UNTRACE — has the opposite effect to TRACE.

DISPLAY — produces a display of the system, or indicates that displays are to be produced at regular intervals.

READ — defines the number of jobs available to be read. This enables the reading of jobs into the simulated system to be batched, as it is in the real system. For example, READ 6 indicates that another 6 job decks are present in the reader hopper to be read when Janus is ready. Subsequent jobs in the input stream do not become available until another READ command is encountered. This technique simulates the operators collecting a batch of jobs from the input counter and loading them into the hopper.

END — indicates that the simulation run is to stop.

Figure 29 shows a typical sequence of control cards for a short simulation run. Initially tracing is switched on for several aspects (job flow, input and Janus). The PPU and CPU libraries have entries added to them (as described in the next paragraph) and simulation begins. At 10 seconds, a display is requested, and at 15 seconds some alterations are made to the trace flags. Simulation then continues until 60 seconds when the END command is encountered.

7.3.2 Definition of Simulated Libraries

The operation of the relatively few PPU programs in the Scope system has been built into the program of Simulator B, as will be described below (7.4). This takes care of the operation and timing of the PPU programs in the system. The only other information the simulator requires about the PPU programs is where each resides, so that when one has to be loaded into a PPU, the correct procedure can be simulated. For each PPU program in the system, the simulator needs the length, the positon on the disc, and whether the program is also kept in central memory. This information is specified through the PPLIB command in the simulation control stream. For example, the first entry after the PPLIB command in Figure 29 refers to the PPU program IRA. The copy of IRA stored on the disc starts at half-track 11, sector 45. These two numbers form the disc address of IRA. The next number of the card defines the length of IRA to be 3 sectors or about 200 central memory words long. The asterisk denotes that IRA is resident in central memory as well as being on the disc. When the loading of IRA is simulated, the simulator can extract this information from the library and schedule the appropriate loading sequence.

CPU programs are not individually catered for in the simulator program. Instead the simulator uses the same processes to process all CPU programs, both those in the system and those from users.

This is possible because a CPU program, when running, appears to the system as a sequence of requests from the program to the system, separated by periods of execution. The effect of the program on the system is determined solely by the nature of these requests and the times at which they are made. This information, which is available from Simulator A, needs to be specified for each CPU program in the library of the simulated system, as well as the information required for the PPU programs.

For example, the first CPU program defined in the CPLIB of Figure 29 is REWIND. This program simply issues a single call to the PPU program CIO to rewind the file, and then ends. The timing involved in this process was found from Simulator A. Looking at Figure 29, the first card of the REWIND definition gives its position on the disc (half-track 15, sector 24), and its length (2 sectors). REWIND is not resident in central memory (no asterisk) and so must be loaded from the disc. The last two quantities (.0055 and 1) are the total (default) execution time and the number of monitor requests the program makes during execution. The monitor requests themselves are defined on subsequent cards. The card following the REWIND card specifies a call to CIO, to perform a rewind operation (code 2) on whatever file is named as the parameter of the REWIND program when its execution is simulated. This parameter is variable and is signified by the ** field. The CIO request is to be made .005 seconds after REWIND starts executing.

The second CPU program in Figure 29, COPY, contains two examples of a repeated request. The COPY card itself is set up the same way as the REWIND card above, but with the specification that two requests follow. The first of the request cards define a call to CIO to read (code 0) from the file named as the first parameter of COPY. This request is to be first issued at .005 seconds after the start of execution, and then at intervals of .01 seconds until the program stops execution. The second request (to write on the file named as the second parameter) is also repeated. The number of requests actually issued when the execution of COPY is simulated will depend on its execution time.

The method by which these definitions of CPU programs are used to simulate CPU program execution is described in 7.5.1.

7.3.3 Job Specification

Besides the command stream, the other source of input for the simulator is the descriptions of the jobs in the input stream. These descriptions are written in a specially contrived job specification language, which resembles the format of Scope control cards.

As explained in 2.2.7, the processing required from the Scope system for a job is described by a set of control cards. These can call for the execution of any of the Scope library programs, for the execution of programs stored on a user's file, or for a change in resource allocation. The job descriptions for the input to the

```
JOBNAME 0100 040000 100
RUN(INPUT,OUTPUT,LGO)
LOAD(LGO)
EXECUTE.
END

QQQJOB1 0010 040000 050
RUN(INPUT,X,LGO)
RFL(20000)
LOAD(LGO)
EXECUTE.
EXIT.
RFL(5000)
REWIND(X)
COPY(X.OUTPUT)
END

JOBNAME 0010 004000 1000
COPY(INPUT,X)
REWIND(X)
COPY(X,OUTPUT)
REWIND(X)
COPY(X,OUTPUT)
END
```

```
JOBNAME,T100,CM40000.
RUN(S,,,INPUT,OUTPUT,LGO)
LOAD(LGO)
EXECUTE.
<END OF RECORD CARD>

QQQJOB1,T10,CM40000.
RUN(S,,,INPUT,X,LGO)
RFL(20000)
LOAD(LGO)
EXECUTE.
EXIT(S)
RFL(5000)
REWIND(X)
COPY(X,OUTPUT)
<END OF RECORD CARD>

JOBNAME,T10,CM4000.
COPY(INPUT,X)
REWIND(X)
COPY(X,OUTPUT)
REWIND(X)
COPY(X,OUTPUT)
<END OF RECORD CARD>
```

Figure 32 : Input descriptions from the simulation job stream (left) and the corresponding Scope control cards (right). All (normally omitted) default parameters have been included in the Scope control cards to illustrate the correspondence better.

simulated system take the form of "control cards" that correspond closely in appearance and meaning to the Scope control cards. The correspondence is shown in the examples of Figure 32.

The first card of each simulated job description contains the job name, the job's initial memory requirements and its CPU time limit. In the real Scope system these quantities also appear on the first card in the deck, the job card. The first card of the simulated specification also contains the number of cards in the simulated job deck, so that the correct size of input file can be computed. The latter consideration is necessary for accurate simulation of disc allocation, as explained in 7.5.2. As an example of this first card, the first job in Figure 32 has a time limit of 100 seconds, and requires 40000 words of central memory. There are 100 cards in its (simulated) deck.

The subsequent cards of each job specification describe the processes to be simulated for the job. There are three types of process that can be specified, corresponding to the three types of control card listed above.

The simplest type of control card is that specifying a resource request or other similar operation. These operations do not involve the execution of a special CPU or PPU program ; in the real system they are processed directly by the program that interprets the control cards (IAJ). The requests of this type that are recognized by the simulator include :

RFL(n) requests the allocation of n words of central memory to the job. This card can be used to change the memory allocation specified on the simulated job card.

PAUSE requests that the system suspends processing while the job waits for some action from the operator.

EXIT provides an "exit path" in the event of a normally fatal error. If EXIT is encountered in the normal sequence of control cards, processing of the job stops. If a fatal error occurs before the EXIT is encountered, processing continues with the cards following the EXIT. If the job contains no EXIT card, a fatal error stops its processing.

Because each of these special system request cards involved specialized processing, new cards can be introduced to the simulated system only by changing the simulator program. The processing for new cards of this type should be added to the event that interprets control cards.

The second type of process that can be specified by a control card is the execution of an absolute central processor program from the system library. The programs contained in the library are defined by the CPLIB card described in Paragraph 7.3.1. The control card itself contains the program name, parameters (file names) to be used during the execution, and the period for which the program is to

execute. The system library will normally contain programs for file handling, compilers, and a loader for the relocatable programs generated by the compilers. Each of these was studied using Simulator A, and the details of the requests they made to the system were recorded for inclusion in the simulated library table of Simulator B.
The programs in the standard simulated library include :

REWIND(X)	positions file X at the beginning of information.
COPY(X,Y)	copies file X on to file Y.
EVICT(X)	releases space assigned to file X. This is done automatically at the end of the job for all but the output file but it is sometimes convenient to use it during processing.
RUN(X,Y,Z)	standard Fortran compilation, with input from file X, compiled program written on file Z and listing on file Y.
COMPILE(X,Y,Z)	a general compiler whose request characteristics are a compromise of other compilers than Fortran. The parameters have the same significance as for RUN.
LOAD(X)	load the relocatable program from file X. Also load required subroutines from the system library.

This standard library can be altered or extended by use of the CPLIB card in the simulation control streams, as described above (7.3.2).

The third and last kind of control card calls for execution of a user's program. The program must already have been loaded by a LOAD(X) command. The EXECUTE card specifies that the loaded program be executed. The card lists all the requests that the program will make to the operating system, in the same way as the CPLIB cards define the requests that system programs make (see 7.3.2).

Finally, the last card in the specification of each job is an END card signifying that another job follows.

Specification of a typical sequence of jobs is shown in Figure 32. The Scope control cards that would be used for the real job appear also. It can be seen that the two specifications are very similar. This feature of the design was deliberate, in order to simplify the process of job specification for users of the simulator, who would presumably be familiar with the Scope formats.

7.3.4 Tracing

Tracing is the most detailed level at which Simulator B produces information about the processes it is simulating. For tracing purposes, the processes simulated are divided into about 15 disjoint sets, each representing a different aspect of the system's behaviour. Typical examples are disc accessing, overall job flow, memory management, file operations, etc. Each item of the trace output

```
30.1864  JANUS    .STACK REQUEST TO READ FROM FILE INPUT
30.1865  JANUS    .PROCESSING STACK REQUEST
30.1964  JANUS    .DISC CHANGED TO HEAD GROUP 2
30.2052  SYSTEM   .JOB QAZWBAA SELECTED FROM INPUT QUEUE
30.2102  QAZWBAA  .REQUEST FOR 040000 WORDS OF CM
30.2111  QAZWBAA  .FIELD LENGTH INCREASED TO 040000
30.2123  QAZWBAA  .JOB BROUGHT TO CONTROL POINT 2
30.2212  SYSTEM   .JOB QAZWBBB SELECTED FROM INPUT QUEUE
30.2235  QAZWBAA  .READING CONTROL CARDS
30.2237  QAZWBAA  .STACK REQUEST TO READ FROM FILE INPUT
30.2261  QAZWBBB  .REQUEST FOR 040000 WORDS OF CM
30.2270  SYSTEM   .CONTROL POINT 3 MOVED TO 055700
30.2271  QAZWBBB  .FIELD LENGTH INCREASED TO 040000
30.2283  QAZWBBB  .JOB BROUGHT TO CONTROL POINT 3
30.2372  SYSTEM   .JOB QAZWBCC SELECTED FROM INPUT QUEUE
30.2396  QAZWBBB  .READING CONTROL CARDS
30.2398  QAZWBBB  .STACK REQUEST TO READ FROM FILE INPUT
30.2422  QAZWBCC  .REQUEST FOR 040000 WORDS OF CM
30.2431  SYSTEM   .CONTROL POINT 4 MOVED TO 115700
30.2432  QAZWBCC  .FIELD LENGTH INCREASED TO 040000
30.2443  QAZWBCC  .JOB BROUGHT TO CONTROL POINT 4
30.2556  QAZWBCC  .READING CONTROL CARDS
30.2558  QAZWBCC  .STACK REQUEST TO READ FROM FILE INPUT
30.3375  QAZWBAA  .PROCESSING STACK REQUEST
30.3472  JANUS    .JANUS REACTIVATED BY IIQ
30.3490  JANUS    .STACK REQUEST TO READ FROM FILE SYSTEM
30.4638  QAZWBAA  .DISC CHANGED TO TRACK 001
30.5000  +++++++  .TRACE EVENTS
30.5000  +++++++  .TRACE ASSIGNMENTS
30.5238  $$$$$$$  .EVENT 57 FOR JOB 8
30.5239  QAZWBAA  .TRANSFERRING 01 PRUS OF FILE INPUT
30.5248  $$$$$$$  .EVENT 8 FOR JOB 1
30.5248  ********  .PP 2 RELEASED FROM JOB 8
30.5248  ********  .PP 2 ASSIGNED TO JOB   9
30.5248  QAZWBBB  .PROCESSING STACK REQUEST
30.5248  $$$$$$$  .EVENT 30 FOR JOB 8
30.5248  $$$$$$$  .EVENT 54 FOR JOB 9
30.5248  $$$$$$$  .EVENT 31 FOR JOB 8
30.5248  $$$$$$$  .EVENT 32 FOR JOB 8
30.5248  QAZWBAA  .COMPILE.
30.5248  $$$$$$$  .EVENT 29 FOR JOB 8
30.5248  QAZWBAA  .LOADING CPU OVERLAY
30.5300  +++++++  .UNTRACE EVENTS
30.5300  +++++++  .UNTRACE ASSIGNMENTS
30.5300  +++++++  .UNTRACE DISC
30.5300  +++++++  .UNTRACE STACK
30.5903  QAZWBBB  .COMPILE.
30.6558  QAZWBCC  .COMPILE.
30.7100  JANUS    .END OF INPUT STREAM
31.0152  QAZWBAA  .LOAD.
```

Figure 33 : A section of a trace from Simulator B

comprises the simulated time at which the event of interest occured, the name of the job involved, and a brief message describing the event. A typical segment of trace output is shown in Figure 33. Note the use of three dummy "job names" : +++++++ (for commands from the simulator control stream), ****** (for resource management) and $$$$$$$ (for events simulated). Figure 33 also shows how various levels of tracing can be invoked by using the TRACE and UNTRACE commands.

7.3.5 Displays and Reports

Displays give a picture of the state of the simulated system at a given moment in time. They are designed to resemble the console displays generated continually by Scope in the real computer in order to keep the human operators informed of what is going on in the system. They have the same purpose in the simulated system, except that they cannot be generated continually. In the simulated system, the console is merely an abstraction, so that instead of appearing on the console screen, the displays are printed as part of the simulation output. The information shown in a display includes a list of all jobs in the system with their current status, a display of the activities of the jobs occupying control points, a table of what each peripheral processor is doing, and an account of all allocations of space on the 6603-11 disc. A sample display is shown in Figure 34.

```
                                DISPLAY AT  200 SECONDS

                                    JOBS IN THE SYSTEM
NO     NAME          PRIOR     STATUS          TIME LIM    MEMORY
 7     DDDDD01       1501      CONTR PT 2        1000      060000
11     DDDDD05       1501      INPUT Q           1000      060000
12     BBBB522       4361      INPUT Q           0040      050000
13     AAAAA29       4467      CONTR PT 5        0005      040000
15     AAAAA09       4467      PRINTING          0005      040000
22     AAAAA19       4467      OUTPUT Q          0005      040000
23     BBBB520       4361      INPUT Q           0040      050000
26     AAAAA24       4467      PRINTING          0005      040000
27     AAAAA25       4467      CONTR PT 4        0005      040000

                                    CONTROL POINTS
1.JANUS    P 7777 L 7777 CP    0 PP 129 RA 012100 FL 003600    1-------
2.DDDDD01  P 1501 L 1000 CP   69 PP  45 RA 015700 FL 060000 A --------
3.NEXT     P 0000 L 0000 CP    0 PP   0 RA 075700 FL 000000    --------
4.AAAAA25  P 4467 L 0005 CP    2 PP   6 RA 075700 FL 040000 X --------
5.AAAAA29  P 4467 L 0005 CP    1 PP   6 RA 135700 FL 040000 X -2------
6.NEXT     P 0000 L 0000 CP    0 PP   0 RA 177700 FL 000000    --------
7.NEXT     P 0000 L 0000 CP    0 PP   0 RA 177700 FL 000000    --------

                                    PPU ASSIGNMENTS
                                    0  MTR  SYSTEM
                                    1  1IR  JANUS
                                    2  1SP  AAAAA29
                                    9  DSD  SYSTEM
```

Figure 34 : A display of the simulated system.

```
==========================================
                  REPORT
==========================================

 SIMULATED TIME (SEC):              60.051
 REAL CPU TIME (SEC):                9.454
 SIMULATOR:COMPUTER TIME RATIO:      6.352
 NUMBER OF JOBS READ:                7
 NUMBER OF JOBS COMPLETED:           4
 TURN-AROUND TIMES (SEC):
       AVERAGE:                     34.796
       MINIMUM:                     27.906
       MAXIMUM:                     41.676
 CPU UTILIZATION (P.C.):            58.288
 PPU UTILIZATION (P.C.):            21.466
 CORE UTILIZATION (P.C.):           44.671

==========================================
```

Figure 35 : A report generated by Simulator B.

Reports are intended to summarize information from which the overall performance of the system can be judged. They include an account of how queue lengths are distributed in time, and various quantities on which an estimate of system efficiency can be based, such as job turn-around times, through-put, CPU usage and so on. Using these quantities, a rapid comparison can be made between two systems, so that the effect of a change from one to the other can be assessed. A typical report is shown in Figure 35.

7.4 SIMULATING SCOPE WITH THE EVENT SIMULATOR

This section describes the method by which the information about Scope obtained from Simulator A was incorporated into Simulator B, using the auxiliary subroutines described in Section 7.2.

7.4.1 Implementation of Results from Simulator A

The results of the work with Simulator A, as previously explained, took the form of precedence-decision networks of the kind shown in Figure 13. The nodes represent events and the arcs represent delays between events. Some arcs have conditions associated with them, and these conditions must be satisfied before the arc is followed. Other arcs have no conditions and are followed in all situations. Suppose an event E is represented by node N. After event E occurs, processing continues along each unconditional arc leaving N and also along each conditional arc leaving N for which the condition is satisfied. A number of possibilities arise from this rule, some examples of which are :

(1) If there is no arc that can be followed from N, as with node 4 in Figure 13, then no subsequent process is to be initiated after event E.

(2) If there is one unconditional arc leaving N, as with node 1 in Figure 13, then this is the only possible path that can be taken, and the event at the other end of the arc must be scheduled with a delay equal to that associated with the arc.

(3) In many instances, there will be more than one arc leaving N, but the conditions associated with them will be such that only one is ever followed in any particular instance. The arc chosen will be chosen according to the conditions applying in the system when E occurs. Once the decision is made, the procedure is the same as in case (2) above. In Figure 13 nodes 2, 3, and 5 are of this type.

(4) It is possible for more than one arc to be followed from a node at the same time. This implies that another process has been initiated. For example, in node 9 of Figure 13, there are two unconditional exit paths. One is the continuation of processing by IRA (arc 9 - 3) and one is the assignment of a PPU to a new process, the execution of IAJ (arc 9 - 10). Both must be followed, so that simulation of the event at node 9 must include

scheduling the events at both node 3 and node 10, each with the appropriate delay.

Once a particular event is simulated, if the network segment associated with it is known, the scheduling of its successor events is thus easily arranged. Each event in the operation of the system is scheduled appropriately by its predecessor in the network, and will thus be simulated at the appointed time. It remains only to initiate simulation by scheduling the one event that has no predecessor, the dead-start of the system.

The only situation not covered by these considerations is the case where a node has more than one predecessor. Suppose, for example, that the network contains a node N, with predecessors A and B. When the events at A and B are simulated, each will schedule the event at A, probably at different times. The network structure implies that the event at N cannot be simulated until the delays associated with both these scheduling operations have expired. That is, event N should be simulated at whichever of its scheduled times is the later. In the simulator program, this is arranged by programming each such event N so that it must be invoked twice before its simulation actually takes place. There are very few instances of this situation in the system. An example is the calling of a CPU program from the system library. Execution of the program cannot be simulated until the CPU is assigned to the job and the loading process (done by the stack processor) is complete.

It should be noted that Simulator B does not explicitly use the network structure itself. While it may have provided a neat method of programming to have included the network as a data structure from which the program was driven, it was decided to build the network implicitly into the structure of the program. This was done in order to make the program easier to follow by its intended users, who would find its structure similar to the structure of the system programs being simulated, which do not use an explicit network. Use of a network-driven program would simplify the overall structure of the simulator, but the structure of an event, taken in isolation, would be harder to understand.

7.4.2 An Example

The process of converting the network from Simulator A into a program segment for Simulator B can be illustrated by the example in Figure 36. This shows the networks for the process of requesting, assigning and loading a PPU. The event of interest is the request for the PPU, corresponding to node 1 in each network. The program segment for the event is shown in Figure 37.

When a PPU is requested, it is necessary firstly to see if any PPU is available for allocation. If there is none, the requesting job is put on the PPU queue and no further events are scheduled (node 2). If a PPU is available, it is assigned. Processing thereafter depends on the residence of the PPU program requested. If it resides in central memory, the time required to find and load it

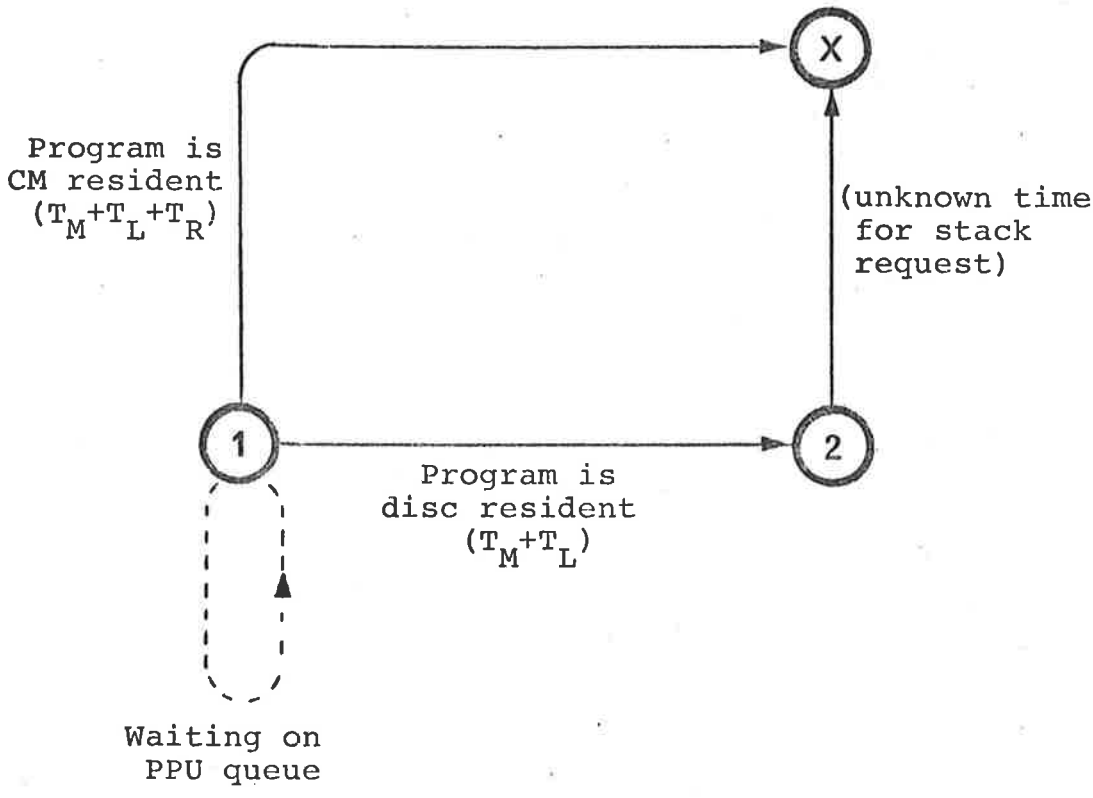


Figure 36 : Precedence-decision network for the request and assignment of a PPU.

Node 1: Request assignment of PPU.
Node 2: Issue stack request to read the PPU program from disc.
Node 3: First event in the execution of the PPU program.

```
C     EVENT 4 - REQUEST PERIPHERAL PROCESSOR ASSIGNMENT.
C     JOB IS THE NUMBER OF THE JOB REQUESTING A PPU.
C     NM IS THE NAME OF THE PPU PROGRAM REQUESTED.
C
C     SEE IF THERE IS A PPU FREE.  IF NOT, PUT JOB ON THE PPU QUEUE.
C     IF ONE IS FREE, GO TO STATEMENT 42.
   40 DO 41 I = 1,NPP
   41 IF (IPP(I).EQ.0) GO TO 42
      CALL QUEUE (2,3,JOB)
      RETURN
C
C     A FREE PPU EXISTS.  ASSIGN IT TO THE REQUESTING JOB.  LOOK FOR THE
C     REQUESTED PROGRAM IN THE LIBRARY TABLES (LIBPP).  IF IT IS NOT THERE,
C     CALL SUBROUTINE ERROR.  IF IT IS THERE, GO TO STATEMENT 44.
   42 CALL ASSIGN (IPP(I),JOB)
      DO 43 I = 1,NPROG
   43 IF (LIBPP(1,I).EQ.NM) GO TO 44
      CALL ERROR (26HUNDEFINED PPU PROGRAM CALL, 1,1,NM,2HA3)
C
C     THE REQUIRED PROGRAM HAS BEEN FOUND IN THE LIBRARY.  CHECK ITS
C     RESIDENCE.  IF IT IS IN CENTRAL MEMORY, SCHEDULE THE NEXT EVENT
C     AFTER THE TIME REQUIRED TO FIND AND READ THE PROGRAM.  IF IT IS ON
C     THE DISC, GO TO STATEMENT 45.
   44 IF (IBYTE(LIBPP(2,I),5).EQ.0) GO TO 45
      CALL SHEDULE (KU(JOB),JOB,TMTR+TLIB+TRCM)
      RETURN
C
C     THE PROGRAM IS ON THE DISC, AND A STACK REQUEST MUST BE ISSUED TO
C     GET IT.  SHEDULE A STACK REQUEST (EVENT 7), PASSING ON THE INFORMATION
C     ABOUT THE LOCATION OF THE PROGRAM IN LIBPP(2,I).
   45 CALL SHEDULE (7,JOB+LIBPP(2,I),TMTR+TLIB)
      RETURN
```

Figure 37 : The program text for the request and assignment of a PPU

can be calculated immediately, and the next event for the requesting job can be scheduled with this delay. This next event (node x) is variable, and will depend on the PPU program being requested. The event that scheduled the PPU request would have made this information available to the PPU assignment event. If the PPU program is disc resident, a stack request is scheduled. There is no way of telling in advance just how long the stack request will take, so the next event after the PPU assignment cannot be scheduled. It becomes the next event to be scheduled after the stack request is processed, and will automatically be scheduled by the events simulating the disc access.

Releasing a PPU from a job A cancels the assignment. The PPU queue is inspected to see if any jobs are waiting to use a PPU. If there are such jobs, one is selected, say B, and removed from the queue. The PPU assignment event is scheduled for B. No further event is scheduled for A, since no event for job A depends upon the PPU being released.

7.5 SOME DIFFICULT AREAS

Although most events in the system could be programmed in a relatively straightforward manner (as described in the example of 7.4.2), a few were more difficult, mainly because of the complexity of the programs being simulated. This section describes briefly some of these difficult areas, and outlines the methods used to deal with them.

7.5.1 CPU Program Execution

If a CPU program made no requests to the system, simulating it could be done simply by obtaining its execution time, T, (which is defined either in the job stream or in the simulated system library table) and scheduling the next event after execution with a delay of T. T would be added to the accumulated CPU time of the job.

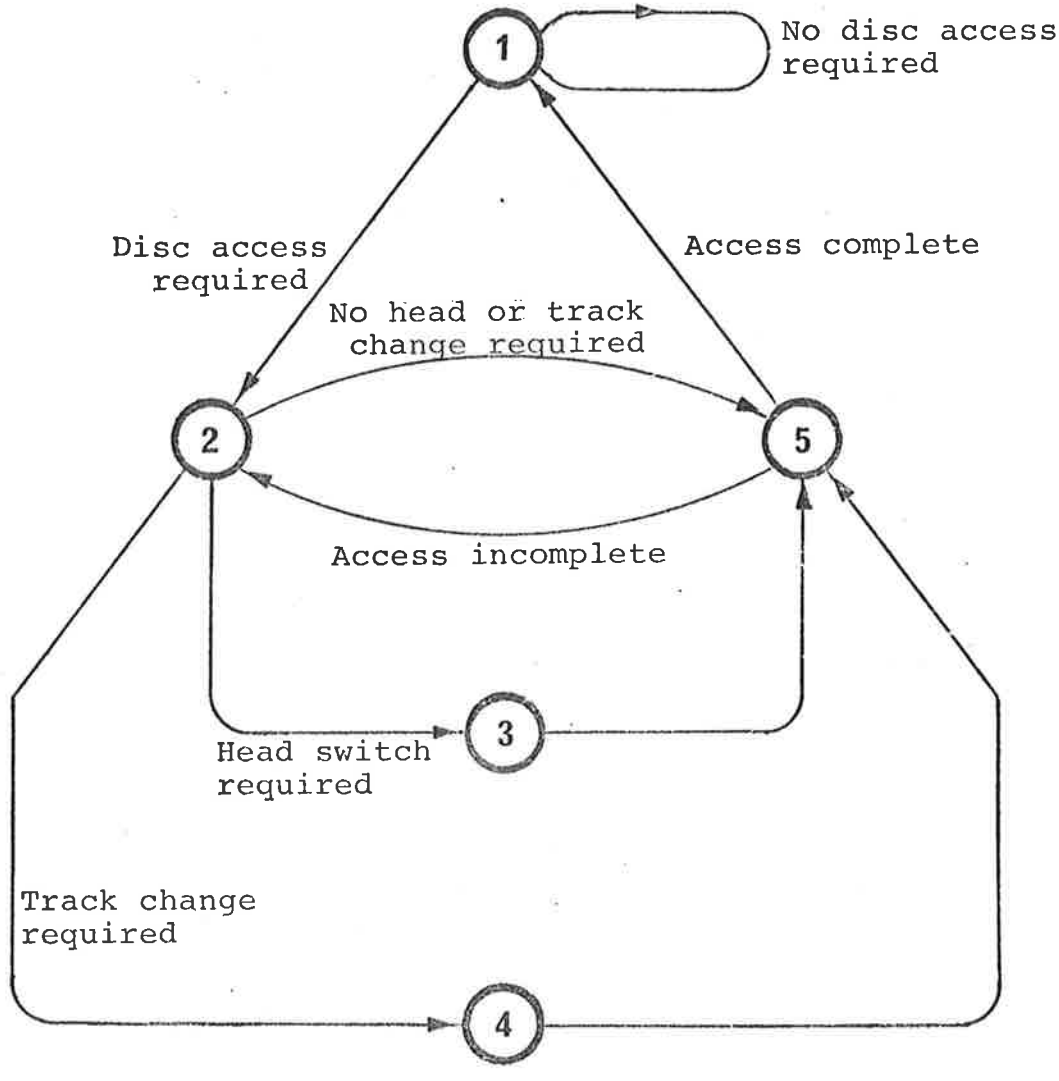
CPU programs do make requests to the system, though, and the input to the simulator (see 7.3.2) enables the user of the simulator to specify just what requests are made and when. As shown in the last example of 7.3.2, a request can be defined as a repeated request, to be simulated when the program has been executing for times, t, t+x, t+2x, ... until execution stops, where t and x are defined in the program specification. A repeated request may also be defined starting from the end of execution, to occur at times ... T-t-2x, T-t-x, T-t. Repeated requests are useful for defining copying and other file-processing programs, where input and output requests are repeated in this manner. The event that simulates a segment of CPU processing must be capable of identifying the next request, however it has been defined, and computing the time at which it will occur. The request corresponding to the event must then be scheduled at the appropriate time.

This is done by keeping a list of future requests and their invocation times for each job executing a CPU program. When the CPU is assigned to a job, the next request is identified. If it is a single request, it is removed from the list ; if it is a repeating

request, it is left in the list with the repetition time, x added to the invocation time. The type of request to be simulated is determined, and the event to initiate it is scheduled with a delay corresponding to the invocation time. Simulation of the request may or may not involve releasing the CPU ; this depends on the nature of the request. If the CPU is released, it can be assigned to another job while the request is being processed, or it can remain idle. When the request to the system has been processed (unless it is the end of program request) the job concerned requests the CPU again. When the CPU is assigned, the whole process is repeated until the CPU program ends.

7.5.2 Stack Processing

Stack processing is difficult to simulate because of the complexity of the operation and because of the amount of information that must be maintained in order to do it properly. The network for a disc access is shown in Figure 38. Simulation of the process involves 5 events corresponding to the 5 nodes in the diagram. At event 1, the stack processor starts inspecting the stack and choosing which request to process next. Some requests (such as rewinding or evicting a file) can be processed simply by changing tables, and do not require a disc access. These result in a direct return to event 1, after a suitable delay (which will depend on the number of entries in the stack and the nature of the request).



Node 1: Start processing stack request
Node 2: Start accessing next half-track
Node 3: Initiate head switch
Node 4: Initiate track change
Node 5: Initiate transfer of information

Figure 38 : The network for processing a stack request. Durations are not shown.

If a disc access is required, tests must be made to see what processes are necessary to simulate it (event 2). In order to take the correct path(s) from node 2, the simulator must know both the current position of the disc and the position of the disc of the data it is required to access. This allows the correct sequence of head switches, track selections and sector selections to be carried out. The current position of the disc is determined from the last access (which defines the track and head group) and the time (which determines the rotational position). The position on the disc of each file in the system is determined when it is written, using the same algorithm as the real system. Thus during subsequent accessing of a file, the corresponding area of the disc will be known and so the appropriate events will be simulated. The only exception is the system library file, which contains the programs in the operating system. This is set up at dead-start and occupies the lowest addresses on the disc.

It was found convenient to be able to issue a stack request by disc address (specifying the physical address on the disc at which it was desired to read or write) or by file name (where the current position of the file was retrieved from the file table, and converted to a disc address). This flexibility also contributed to the complexity of the stack processing events, but greatly simplified many other areas of the simulator where stack requests were set up and scheduled.

7.5.3 Memory Management

When a job requests an increase in the amount of central memory allocated to it, whether at its initiation or during its processing, the monitor uses a relatively complex algorithm to compute the optimal plan for obtaining the required memory. The criterion for optimization is the minimization of the memory to be moved. In Figure 39, a number of jobs are shown occupying central memory. If job E has to increase its memory allocation slightly, this is obviously best achieved by extending E's allocated area downwards. This would involve no movement of memory. If job A wanted an increase, it would be better to move A up, giving it the resulting gap, than it would be to move B, C, D and E down. A request for an increase by job C would lead to a more complex situation. The algorithm used by the monitor is capable of investigating all these possibilities.

In the real system, when the monitor decides that it is necessary to move the memory associated with a control point, all activity at that control must cease before the move can take place. The monitor then assigns the CPU to a storage move program. When the move is complete, the CPU is reassigned, and any suspended activities are restarted.

Simulation of this process involves firstly implementation of the algorithm to find the best move strategy. For each control point whose storage is to be moved, the simulator must then delay for as long as it would take to stop the control point's activities.

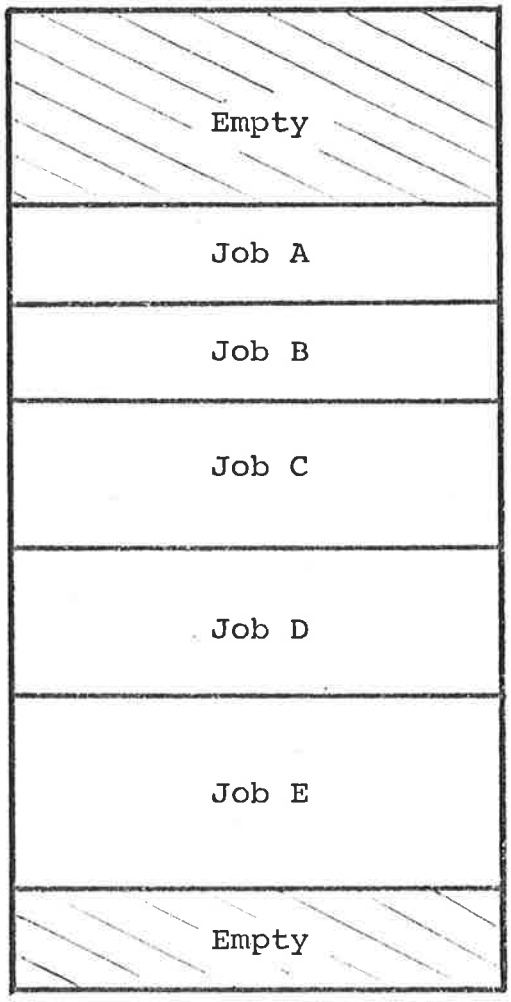


Figure 39 : A number of jobs occupying space in central memory.

This time is obtained by consideration of the activities currently going on at the control point, which are known to the simulator. The CPU is assigned to the memory move program for the required time (which is computed from the amount of memory to be moved). The pointers that define the memory of each control point are then updated, the CPU is re-assigned, and the control point is reactivated. This procedure is repeated for as many control points as need to be moved.

If a job requests more memory than is currently available, its field length is automatically reduced to 100_8 words (the minimum required to sustain a running job) and the job is put on a queue. Whenever a job releases memory, a check is made of the requests on this queue to see if any can be accommodated now that more memory is available. If such a queue entry does exist, its request for memory is re-scheduled.

Accurate simulation of the processing concerning central memory is important because central memory is one of the resources in the system that is in short supply. System performance would thus be expected to be more sensitive to errors in this area of the simulation than in many others.

7.6 SUMMARY

The design and construction of Simulator B brought together information from many different areas. The results from Simulator A, the equipment responses from Chapter 6, the requirements for the external character-

istics of the program, and the restrictions of the language in which the program is written all had a vital part to play in the design and implementation of the simulator at all levels.

Once Fortran had been chosen as the language in which to write Simulator B, the first task was to make Fortran more like a simulation language by writing a set of subroutines for managing time, event scheduling, queues and resources. A few such relatively simple subroutines made the programming of Simulator B much easier, and made the program text considerably easier to follow.

The design of the inner workings of the simulator was largely predetermined from the network structure of the system (mapped out using Simulator A) and the facilities provided by simulation subroutines. The external interfaces of the program also needed careful consideration, and were designed for maximum flexibility. Input to the simulator could be used to define the equipment configuration, software, and the job stream as well as to define parameters to control the simulation run. Output could be produced at almost any level of detail, from a listing of every event as it was simulated to nothing but a statistical summary at the end of the run.

Certain areas of Scope processing proved difficult because of their complexity or because of the quantity of information required for adequate simulation. These problems were not fundamental ones, however, and were solved by careful programming and many trial runs.

Once the simulator was written, and the obvious errors eliminated from it, the next stage was its validation, which is described in Chapter 8. In Chapter 9, some examples serve to illustrate the uses to which the simulator can be put, and the results that can be obtained from it.

CHAPTER 8

VALIDATION OF THE EVENT SIMULATOR

The process by which information from Simulator A was transformed into the program of Simulator B was described in Chapter 7. The information concerned was verified by virtue of the fact that Simulator A was successfully validated, as described in Chapter 5. There was ample opportunity, however, for errors to be introduced during the translation process described in Chapter 7, and it was therefore necessary to validate Simulator B when this process had been completed. This validation checks the validity of all the logic of the Simulator B program, and is shown in Figure 4 as "deterministic validation".

It was then necessary to introduce the stochastic effects (which, as has already been noted, were few in number) and to examine their effect on the accuracy of the simulated system. Because the system had become stochastic, a different method of validation was used, and this is shown in Figure 4 as "stochastic validation". The two phases of validation are described in this chapter.

8.1 METHODS OF VALIDATION

There are two approaches to validation, stochastic and deterministic that were considered in the validation of Simulator B. It may be useful to review the general techniques of each and to discuss the significance that can be attached to their results.

8.1.1 Stochastic Validation

In a stochastic validation, the system being simulated usually contains some unavoidable random effect. The result of this effect in the real system will often not be the same as that of the same effect, under the same conditions, in the simulated system. For example, consider the case of a computer operator responding to some request from the system. The time taken by the operator in the real system is, say, x. In the simulated system, the response time would be generated from some random distribution, and could possibly be x but is more likely to be some different value y. The difference in the response times x and y will probably lead to further differences between the two systems in later processing. The results obtained from the two systems, running the same jobs under the same conditions, will be different, even if the simulation is a good one. The validation thus becomes a statistical problem. One must determine the probability of the two different results coming from the same system, given the probable stochastic variation permitted. Much work has been done investigating suitable methods for such an analysis, for example by Fishman and Kiviat (24). To achieve an acceptable level of reliability, a statistical analysis needs a large number of observations (i.e. a large number of runs on the real and simulated systems). When such analysis reveals a discrepancy between the two systems, it usually cannot pinpoint its cause, so that finding the error could be difficult.

8.1.2 Deterministic Validation

No real system is ever completely free of noise, and so no system can strictly be described as deterministic. Often, however, a process can be predicted with such accuracy that, for all practical purposes, it is noise free. This is the sense in which the word "deterministic" is used here.

In order to carry out a deterministic validation, all random effects in both the real and simulated systems must be eliminated. One way of doing this is to set up the system in such a way that there are no random processes in it. Alternatively, a run can be made on the real system, with random effects included, and the actual values taken by each random variable measured. In the latter case, when the run is repeated in the simulated system, the values measured are supplied to the simulator so that they can be used in place of a random selection when the stochastic effects are simulated. The stochastic effects are thus replaced by known effects for the particular run.

If all random effects can be removed, all that is necessary for a deterministic validation is to run the two systems under the same conditions and to gather suitable information about each run. Because the processes concerned are known in detail, and all quantities involved have definitely known values, the results from the two systems should correspond exactly, provided that the validation experiment is correctly set up. If there is no one-for-

one correspondence between the events in the two systems, then there is an error in the simulator. The difference observed could not have arisen because of some stochastic effect. The place in the runs at which the difference is first observed often indicates the section of the model where the error will be found.

8.1.3 Comparison of the Two Methods

Although it is conceptually a much simpler process than stochastic validation, deterministic validation gives much more definite results. With a statistical validation, one can only say that the simulated system mirrors the real system at some confidence level, whereas with a deterministic validation the only limit on the precision of the testing is the accuracy with which quantities can be measured in the two systems. The disadvantage of deterministic validation, however, is the difficulty of fulfilling the conditions it requires - elimination of all stochastic effects. In many systems, the stochastic effects are so many and so frequently encountered that their elimination is impracticable. In some cases, it might be possible to remove stochastic effects, but this might change the system beyond recognition, so that the model being validated bore little relation to the original system. In the case of a computer simulation, however, it is more likely that a deterministic simulation is viable, since so many computer operations involve discrete logical steps with definitely known behaviour. The Scope 3.2 system, for example, contains very few stochastic operations. These can be removed with little effect on the design of the system.

It was therefore considered most desirable for a deterministic validation to be carried out, so that the non-stochastic parts of the system could be checked thoroughly.

8.2 DETERMINISTIC VALIDATION OF SIMULATOR B

In the case of Simulator B, it was possible to arrange a deterministic validation by eliminating all significantly random effects from the system. As has already been noted, the only frequent stochastic operation in the system is the movement of the heads of the 6603-11 disc from one track to another. All other operations necessary for running the system can be validated deterministically, provided that the track changes can be eliminated from both real and simulated systems.

8.2.1 Elimination of Track Changes

It was a trivial matter to ensure that track changes were eliminated from the simulated system. A call to the error subroutine was inserted in the program segment responsible for simulating the event "select a new track". If no error message was printed by the simulator, then no track change had been attempted. Changing tracks was avoided by ensuring that the system only ever wanted to access one track, and was managed in the simulator in the same way as in the real system, as next described.

It was more difficult to eliminate track changes from the real system. The system had to be set up in such a way that access to only one track was necessary or possible. When it is realized what normal operation of the system is like, with track changes going on

continually (usually more than one a second), this is a fairly rigorous constraint. Nevertheless, it was found that a workable version of the system could be set up using only one track. All the system library and all the files associated with jobs had to fit on one track, which, when half-tracking and head switching are considered, amounts to 16 half-tracks (the unit of the disc so far as assignment is concerned) each of which has 58368 central memory words. It was found possible to reduce the space taken by the system library to three half-tracks by deleting all programs not actually used by jobs run during the validation. Thus many PPU programs could be deleted, and all the CPU programs except one. Since the simulator uses the same logic to simulate all CPU programs, as explained in Chapter 7, simulation of any one program is sufficient to test the logic for the execution of all CPU programs.

Reduction of the system library to three half-tracks left 13 half-tracks for use by the jobs being run, which was sufficient to have a number of jobs executing at once, actively competing for resources and thus using the more complex parts of the system - those used to resolve conflicts.

There is one other piece of information that is normally stochastic that had to be made fixed, and that is the rotational position of the disc at the start of the validation run. This can be readily defined in the simulator by use of the DISC card in the simulation control stream. In the real system it was fixed by temporarily blocking all jobs from execution until the reference

mark passed under the heads. This was done by setting up all the control points in such a way that the job initiation program (IRA) bypassed them. When the reference mark reached the heads, the control points were unlocked and the waiting jobs were free to be selected by the system for execution. The time at which this happened was used as the origin from which all subsequent events were measured.

8.2.2 The Validation Experiment

The deterministic validation experiment consisted of setting up a dead-start tape for the reduced, one-track operating system. The computer was then dead-started using this tape, and the control points were locked from assignment to jobs. The jobs to be executed during the validation run were then read in. A specially written PPU program was run to unlock the control points immediately after the reference mark on the disc passed under the heads. This PPU program then behaved in exactly the same way as the sub-monitor, described more fully in Paragraph 5.3.2. As in the validation of Simulator A, the sub-monitor, while executing, produced detailed timing information about the processes going on elsewhere in the system. These processes included the jobs previously read in, since, with the control points unlocked, these were now free to execute. When these jobs had finished, the information that the sub-monitor had produced about them was recorded on punched cards for later analysis.

The choice of punched cards, by the way, gives some insight into the restrictions posed by the need to keep to one track of the disc. All other media (e.g. magnetic tape or permanent disc files) require special software in the operating system to support them. Since the operating system for this validation had to be kept as small as possible, all such special software had to be excised from it. The card punch is controlled by Janus, which had to be included in the system anyway to control the card reader and line printer.

A corresponding run was set up in the simulated system to reflect the restrictions imposed on the real system (i.e. only one track being used, and all the control points locked until the disc was appropriately positioned). Job descriptions corresponding to the jobs run on the real computer were supplied to the simulator. On completion of the simualation run, the events (and corresponding times) corresponding to the real system's monitor requests were identified. The times for these events in the two systems could then be compared.

8.2.3 Results

As explained above, the advantages of deterministic validation is that it should give a one-for-one correspondence between real and simulated events. There is no need for statistical tests to be performed so that some measure of confidence in the results can be obtained. The two systems either correspond, or they do not.

The first time the deterministic validation was attempted, four identical jobs were run on the two systems. Each job called for execution of the system's one CPU program, which read from the input file (requiring the execution of the PPU program CIO and a stack request) and did some processing (about 1/3 second) not requiring PPU assistance. It then wrote on the output file and finished. Each job required 40000_8 words of central memory, about 3.5 seconds of CPU time, execution of about 6 PPU programs and 10 stack requests. Four such jobs trying to run simultaneously result in competition for these resources. Insistence that the simulated system manage this competition in exactly the same way as the real system, with exactly the same timing, is a stringent test of the accuracy of the simulator logic.

The results from the first run of the four jobs described above is shown in Figure 40. It can be seen that there is a vague correspondence between the event times in the real and simulated systems, particularly at the beginning of the run, but the results are not nearly close enough to be satisfactory.

By considering the nature of the deviation between the two systems shown in Figure 40, it was possible to identify the errors in the simulator that produced the deviations. For example, the most obvious inconsistency in Figure 40 is that job 3 ran much earlier in the simulated system than in the real system. This was found to be due to an error in setting up the simulated system. Instead of having the correct 120000_8 words of central memory as

EVENT	Job 1	Job 2	Job 3	Job 4
Control point and memory assigned	0.089 *0.089*	0.102 *0.103*	1.657 *0.124*	1.669 *2.191*
Stack request issued to read control cards	0.102 *0.101*	0.115 *0.115*	1.669 *0.135*	1/681 *2.274*
Control cards read	0.136 *0.128*	0.201 *0.193*	1.708 *0.265*	1.773 *2.491*
Loading of CPU program initiated	0.145 *0.137*	0.210 *0.200*	1.718 *0.272*	1.783 *2.498*
CIO requests access to file name table	0.231 *0.298*	0.296 *0.363*	1.803 *0.435*	1.869 *2.662*
CIO issues stack request to read input file	0.234 *0.302*	0.300 *0.367*	1.806 *0.439*	1.872 *2.666*
CIO requests access to file name table	0.669 *0.781*	0.997 *1.030*	2.241 *1.432*	2.571 -
CIO issues stack request to write output file	0.679 *0.785*	1.007 *1.107*	2.250 *1.436*	2.580 -
Stack request to load PPU program IEJ	0.998 *1.428*	1.191 *1.430*	2.570 *1.714*	2.698 -
Termination of output file	1.126 *1.640*	1.388 *1.902*	2.633 *2.105*	2.829 -
Memory released	1.252 *1.964*	1.515 *2.162*	2.759 *2.299*	2.955 -

Figure 40. (continued overleaf).

EVENT	Job 1	Job 2	Job 3	Job 4
Write dayfile on output file	1.260 *1.971*	1.523 *2.168*	2.766 *2.306*	2.962 -
Evict input file	1.450 *2.228*	1.582 *2.359*	2.891 *2.563*	3.022 -
IEJ (and therefore the job) finished	1.453 *2.231*	1.585 *2.363*	2.894 *2.566*	3.024 -
CPU time for job	0.330 *.337*	0.330 *.337*	0.332 *.337*	0.331 -
PPU time for job	0.186 *.428*	0.313 *.639*	0.189 *.554*	0.318 -

Figure 40: Times in the real (top) and simulated (bottom) systems for various events during the running of four equivalent jobs. Run number 1.

(continued from previous page)

in the real system, it had 160000 words, so that three jobs could fit in at once instead of two, as in the real system. Correcting this error and performing the simulation run again gave the results shown in Figure 41. These show a marked improvement over the results of Figure 40, but the agreement in some quantities is still not good enough. Some investigation showed that the remaining deviations were due to errors in the simulation of the 6603-11 disc (it will be observed that the discrepancies of Figure 41 are approximately an integral number of disc revolutions, and this clue directed attention immediately to the disc simulation as the likely source of the error).

After a series of five such corrections and re-runs, the results shown in Figure 42 were obtained. Here the deviations between the real and simulated times are about as good as can be expected from the measuring methods used. It is interesting to note that the accuracy shown in Figure 42 for Simulator B is comparable to that shown in Figure 17 for Simulator A (discussed in Chapter 5). This is a very significant result, for it shows that careful design of a simulation model can result in reasonable efficiency without sacrificing the accuracy obtained from a much more detailed simulation.

8.3 SIMULATION OF RANDOM EFFECTS

When the deterministic validation was completed, the simulator could be regarded as an accurate model of the Scope 3.2 system, except that it

EVENT	Job 1	Job 2	Job 3	Job 4
Control point and memory assigned	0.089 *0.090*	0.102 *0.104*	1.657 *1.468*	1.669 *1.739*
Stack request issued to read control of cards	0.102 *0.101*	0.115 *0.155*	1.669 *1.453*	1.681 *1.821*
Control cards read	0.136 *0.134*	0.201 *0.199*	1.708 *1.640*	1.773 *1.836*
Loading of CPU program initiated	0.145 *0.141*	0.210 *0.206*	1.718 *1.648*	1.783 *1.843*
CIO requests access to file name table	0.231 *0.238*	0.296 *0.304*	1.803 *1.829*	1.869 -
CIO issues stack request to read input file	0.234 *0.243*	0.300 *0.308*	1.806 -	1.872 -
CIO requests access to file name table	0.669 *0.656*	0.997 *0.978*	2.241 -	2.571 -
CIO issues stack request to write output file	0.679 *0.666*	1.007 *0.988*	2.250 -	2.580 -
Stack request to load PPU program IEJ	0.998 *0.981*	1.191 *1.254*	2.570 -	2.698 -
Termination of output file	1.126 *1.122*	1.388 *1.450*	2.633 -	2.829 -
Memory released	1.252 *1.315*	1.515 *1.578*	2.759 -	2.955 -

Figure 41. (continued overleaf).

EVENT	Job 1	Job 2	Job 3	Job 4
Write dayfile on output file	1.260	1.523	2.766	2.962
	1.323	*1.585*	-	-
Evict inout file	1.450	1.582	2.891	3.022
	1.514	*1.710*	-	-
IEJ (and therefore the job) finished	1.453	1.585	2.894	3.024
	1.517	*1.714*	-	-
CPU time for job	0.330	0.330	0.332	0.331
	0.337	*0.337*	-	-
PPU time for job	0.186	0.313	0.189	0.318
	0.371	*0.579*	-	-

Figure 41: Times in the real (top) and simulated (bottom) systems for various events during the running of four equivalent jobs. Run number 2.

(continued from previous page)

EVENT	Job 1	Job 2	Job 3	Job 4
Control point and memory assigned	0.089 *.088*	0.102 *.103*	1.657 *1.656*	1.669 *1.670*
Stack request issued to read control cards	0.102 *.101*	0.115 *.115*	1.669 *1.668*	1.681 *1.682*
Control cards read	0.136 *.136*	0.201 *.202*	1.708 *1.708*	1.773 *1.774*
Loading of CPU program initiated	0.145 *.144*	0.210 *.210*	1.718 *1.716*	1.783 *1.782*
CIO requests access to file name table	0.231 *.230*	0.296 *.296*	1.803 *1.802*	1.869 *1.868*
CIO issues stack request to read input file	0.234 *.234*	0.300 *.300*	1.806 *1.806*	1.872 *1.871*
CIO requests access to file name table	0.669 *.664*	0.997 *.991*	2.241 *2.235*	2.571 *2.563*
CIO issues stack request to write output file	0.679 *.674*	1.007 *.995*	2.250 *2.245*	2.580 *2.573*
Stack request to load PPU program IEJ	0.998 *.994*	1.191 *1.192*	2.570 *2.566*	2.698 *2.698*
Termination of output file	1.126 *1.125*	1.388 *1.387*	2.633 *2.631*	2.829 *2.828*
Memory released	1.252 *1.251*	1.515 *1.513*	2.759 *2.756*	2.955 *2.953*

Figure 42. (continued overleaf).

EVENT	Job 1	Job 2	Job 3	Job 4
Write dayfile on output file	1.260	1.523	2.766	2.962
	1.258	*1.520*	*2.763*	*2.960*
Evict input file	1.450	1.582	2.891	3.022
	1.448	*1.580*	*2.888*	*3.020*
IEJ (and therefore the job) finished	1.453	1.585	2.894	3.024
	1.451	*1.583*	*2.891*	*3.023*
CPU time for job	0.330	0.330	0.332	0.331
PPU time for job	0.186	0.313	0.189	0.318

Figure 42: Times in the real (top) and simulated (bottom) systems for various events during the running of four equivalent jobs. Run number 5.

(continued from previous page)

did not include stochastic operations. There are two different types of stochastic effect that need to be considered : track changes on the disc and responses by the operator.

8.3.1 Track Changes on the Disc

As has explained, all operations involving the 6603 disc are frequent, and therefore need to be simulated carefully. Measurement of the times involved in changing tracks on the disc was described in Chapter 6. It was also mentioned (6.4.4) that only in some cases is the stochastic effect of such a track change important. The sector to be read or written is under the heads only at definitely fixed instants, one revolution apart. Although the track change has a random duration, the disc access as a whole cannot proceed until one of these instants, which are not random. In many cases, the range of stochastic variation possible in the track change lies completely within the interval between two such instants, and so the stochastic process cannot affect the overall behaviour of the system. In some instances, however, the random duration of the track change may result in uncertainty as to whether the data transfer to or from the disc will occur at one such instant or the next.

This phenomenon is shown diagramatically in Figure 43, where the events are plotted along a time axis. In each of the cases illustrated, a track change is initiated at T_o, on the left-hand side of the diagram. Its completion time is represented by the

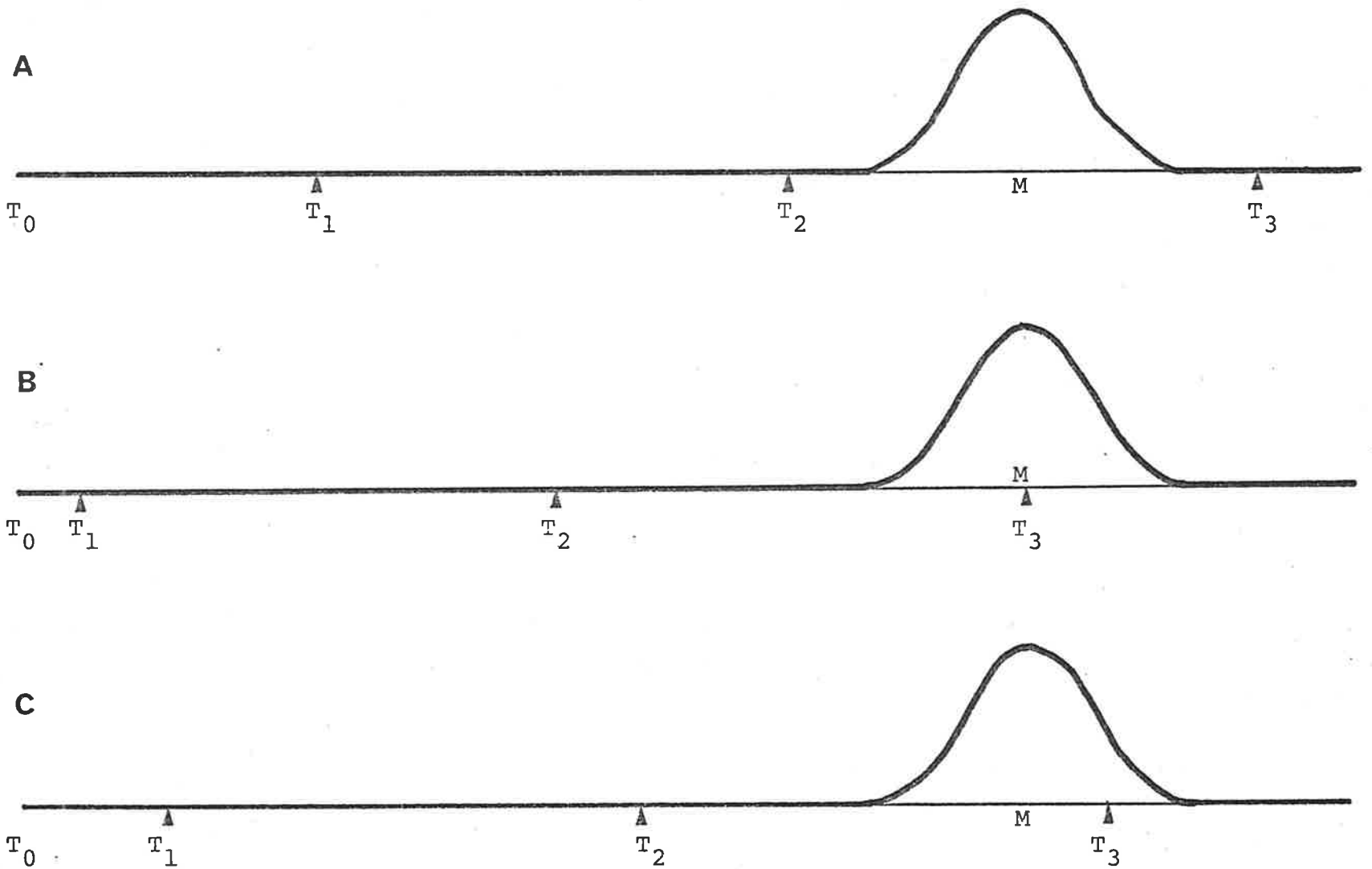


Figure 43 : Three cases showing how a disc access may or may not be affected by the stochastic effect of a track change. See the text for explanation.

probability distribution curve on the right-hand side, which has a mean of 139 milliseconds and a standard deviation of 9. These values are the results of the measurements described in Chapter 6, and are derived in 6.4.3. The instants, T_1, T_2, etc. represent the successive instants at which the required sector is under the heads. In case A, the times at which the track change could possibly finish are wholly between T_2 and T_3. The disc will not be ready to transmit data at T_2, but it will be ready at T_3. Information transfer will therefore start at T_3. In case B, there is a probability of about 0.5 that the track change will finish before T_3. The data transfer could therefore start at T_3 or T_4 with about equal probability. In case C, it will most likely start at T_3, but could possibly be delayed until T_4, with some small probability.

Simulation of a track change involves, firstly, calculation of the mean time for completion of the track change, M. The first T_i greater than M is then found. This T_i is the most likely starting point for the data transfer, and is the one chosen by the simulator. The simulator than estimates the probability, P, of the alternative course being taken in the real system. This is done by find the closest T_j to M. P is then the integral of the probability distribution function between T_j and the nearer asymptote. If $T_j > M$ (in which case i-j), then there is a probability of P that the real system will be one revolution slower. If $T_j < M$, then P is the probability that the real system will be one revolution faster. Since the simulator always selects the most likely course

of action, P is always less than or equal to 0.5. P is estimated by a simple interpolation on values of the normal distribution. This is fairly crude, since only an estimate of the probability is required. The value of P, and whether it is for a slower or faster access, are printed in the simulation trace.

8.3.2 Operator Responses

Operator responses differ greatly from track change times in two respects. They are required much less frequently (on average, about once every 250 seconds as against about once second for track changes), and the times involved are much less certain and vary more widely.

In order that operator responses could be simulated, it was first necessary to measure the time such a response could be expected to take. This could be done readily by analysing the dayfile (the accounting file generated by Scope which lists various events in the system). A dayfile was examined and the times of messages calling for operator action were recorded. For each such request, the message corresponding to the operator's completing the required action was located, and the response time was taken to be the time between the two events. For example, suppose that there is a request for a tape to be loaded at time T_1. The assignment of the tape by the operator occurs at time T_2. The corresponding response time is thus $T_2 - T_1$.

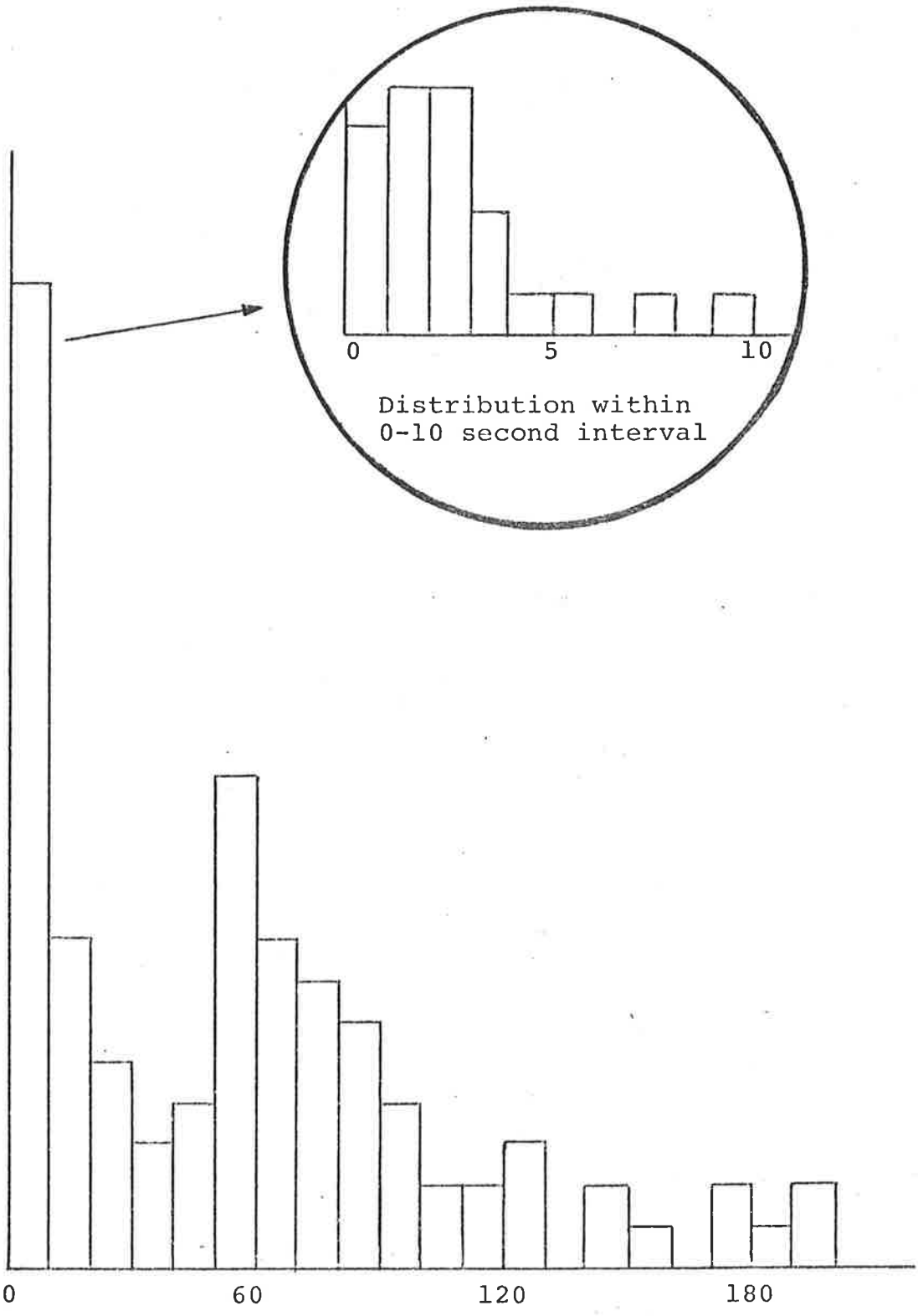


Figure 44 : The distribution of operator response times (in seconds.)

Study of a large number of operator requests in this manner resulted in the distribution of response times shown in Figure 44. The graph is composed of two separate distributions of similar shape, with peaks at 2 and 55 seconds. The requests peaking at 2 seconds are those requiring no action from the operator other than an entry at the console. The distribution for these requests when plotted separately, declines rapidly and is almost zero by the time 60 seconds is reached. It is most unusual for an operator to ignore a simple request as long as this. The requests peaking at 55 seconds are those requiring some additional action (loading a tape, renewing paper in a printer, etc.). These decline more slowly, approaching zero at about 3 minutes.

Simulation of an operator response can be done in one of two ways. The precise time the operator is to take for a particular request can be specified in the job description (the PAUSE command described in 7.3.3 is an example of this). The simulator then uses the specified time. Alternatively, if no time is given, the simulator generates a random time from the distribution in Figure 44.

8.4 PROPOSED METHOD FOR STOCHASTIC VALIDATION

The scheme proposed for the stochastic validation of Simulator B was essentially the same as for the deterministic validation, except that track changes on the disc were permitted. The normal system library could be simulated and a considerable number of jobs could be present in

the system at one time. If a stream of jobs was run using the real system and the same stream run through the simulator, and a set of times for various events obtained for each run, then the two sets could be compared. Such a procedure would enable the simulator to be validated over a longer period of processing than was possible with the one-track system used for the deterministic validation.

With the addition of the stochastic track changes, however, the process of comparing the two sets of data becomes more involved. As previously explained, differences between them could be due either to legitimate stochastic effects or to errors in the simulator. Various statistical techniques are available to assist in comparing two stochastic time series in order to determine (with a given level of confidence) whether they characterise the same system. Naylor and Finger (47) list a total of eight. These statistical techniques assume that the system being modelled is entirely stochastic, in the sense that every inter-event time is subject to random fluctuations. In Simulator B, however, the system being studied is almost wholly deterministic. It is only occasionally (on average about once a second) that a process involving a stochastic delay (i.e. a track change) occurs. For many of these processes, the random effect is cancelled by a later deterministic effect (the accessing of a particular sector). When the random effect is important, it determines whether a disc access occurs on one revolution or the next. That is, the choice is between two deterministic sequences of events, rather than from a continuum of different possibilities.

The operation of Simulator B is thus made up of deterministic processes interrupted at comparatively infrequent intervals by a random choice between two possibilities. Statistical techniques designed for completely stochastic systems were not considered suitable for validation of Simulator B. It was decided instead to extend the deterministic validation techniques to accommodate the stochastic choices, as explained below.

8.4.1 Design of the Experiment

The proposed stochastic validation first required the generation of a significantly large (say 20) batch of jobs having the same distribution of characteristics as the everyday 6400 input stream. (This problem of generating jobs is discussed in Chapter 9). This batch would be run with the real system and with Simulator B, and corresponding event times recorded for the two runs. The two sets of times could then be compared as for the deterministic validation, except that at some stage a track change might occur that caused the two systems to take different paths thereafter. In one system, a track change might be completed just in time for the required sector to be accessed. In the other, because of the random effect, the track change could take longer, and the system would have to wait for another revolution to perform the access. This would delay all subsequent disc accessing by one revolution, and would interfere with all subsequent processing in the system. The comparison of the two event sequences, therefore,

could only be made as far as the first disagreement.

Once the first such disagreement was identified, another run could be made on the simulated system set up so that the simulator took the same course at that point as the real system had. This would enable the comparison to be continued to the next disagreement. The process would be repeated until the whole run had been validated. Thus, in a piecewise way, the whole system could be validated deterministically.

The proportion of track changes leading to disagreements would be fairly small, since the simulator always takes the more likely of the two possible courses. Often the probability of the less possible course is very low, so the chances of the real system behaving differently are not great. At each point where the disagreement actually occured, a check would be made on the estimated probability for this happening (the quantity P, calculated by the simulator as described in 8.3.1). A disagreement where P was very low would give reason for suspicion of the probability estimation.

8.4.2 The Head Crash

After the validation procedure described above was planned, but before it could be put into operation, a "head crash" occured on the 6603-11 disc at the installation being used. This happens when the heads of the disc come in contact with the moving recording surface, damaging the magnetic material. This head crash was

severe, and the damage was considerable, so the Computing Centre decided not to attempt repairing the disc, but to continue using the new 844 disc packs, which by that time had taken over much of the work of the 6603 anyway.

The head crash, unfortunately, meant that the planned validation could not be done, since it was impossible to run the job streams on a real version of the system being simulated (i.e. a 6400 computer with a 6603 as the main storage device). Even if the disc had been repaired, the validity of the planned procedure would have been in doubt, because the characteristics of the disc would probably have changed as a result of the accident.

With no possibility of proceeding with the stochastic validation as planned, other methods for checking the simulator had to be investigated.

8.5 STUDY OF THE STOCHASTIC MODEL

A number of methods for completing the stochastic validation in spite of the head crash were investigated. None of these methods, which are described below, was both feasible and satisfactory. It was desirable, then, to determine how important a part the stochastic events played in the operation of the system - that is, to find how often the probability of a discrepancy between real and simulated systems was significant.

8.5.1 Methods Considered for Stochastic Validation

After the head crash prevented any further runs on the real system, two methods for performing the stochastic validation were considered.

Although jobs could no longer be run on the real version of the simulated system, information was available about how such jobs behaved in the form of accounting files. A limited supply of these was available from the time before the head crash. These were examined to see if the information they contained was sufficient for them to be substituted for the system itself. The accounting files did contain enough information about what happened to jobs as they ran through the system, but there was insufficient information about what sort of jobs they were. Because of this, the same set of jobs could not be defined for running through the simulated system. For example, the accounting files give no indication of the frequency and types of PPU requests made by a user's program. Thus an effective validation could not be performed for lack of information about the jobs being run.

A second approach was to substitute the 844 disc packs for the 6603-11 in the simulated system. This would make the simulated system equivalent to the real system once again. Unfortunately, the information available about the operation of the 844 disc was inadequate. For example, the access time is quoted as 10 to 37 milliseconds, with no indication of the distribution. Incorporation of the 844 in the simulation model would have required a complete

sequence of measurements as described in Chapter 6 for the 6603. This would have been time consuming and costly. Because of increased system loads and longer running hours, it was less feasible to take over the whole system in order to carry out the measurements. In view of these difficulties, together with the relatively small gain at stake (validation over a longer time span than had hitherto been possible), it was decided not to attempt such a procedure.

Both schemes for a stochastic validation were, then, found wanting. The simulator has not, therefore, been validated over a long time span. The shorter validations, however, showed that its performance in almost all areas was extremely close to that of the real system. There is no reason to believe that this accuracy would deteriorate with time. The only aspect of the simulation that has not been validated is the probability estimates for discrepancies due to track changes. These estimates, based directly on physical measurements of the disc, are only informative, and their precision is not crucial.

8.5.2 Frequency of Stochastic Events

The operation of Simulator B can be regarded as a Markov chain as shown in Figure 45. At certain times, there is a random choice between two courses of action, shown by the branches in the chain. The probabilities associated with each branch can be computed (as described in 8.3.1), and the simulator always selects the branch with the higher probability (1-P). It prints the probability

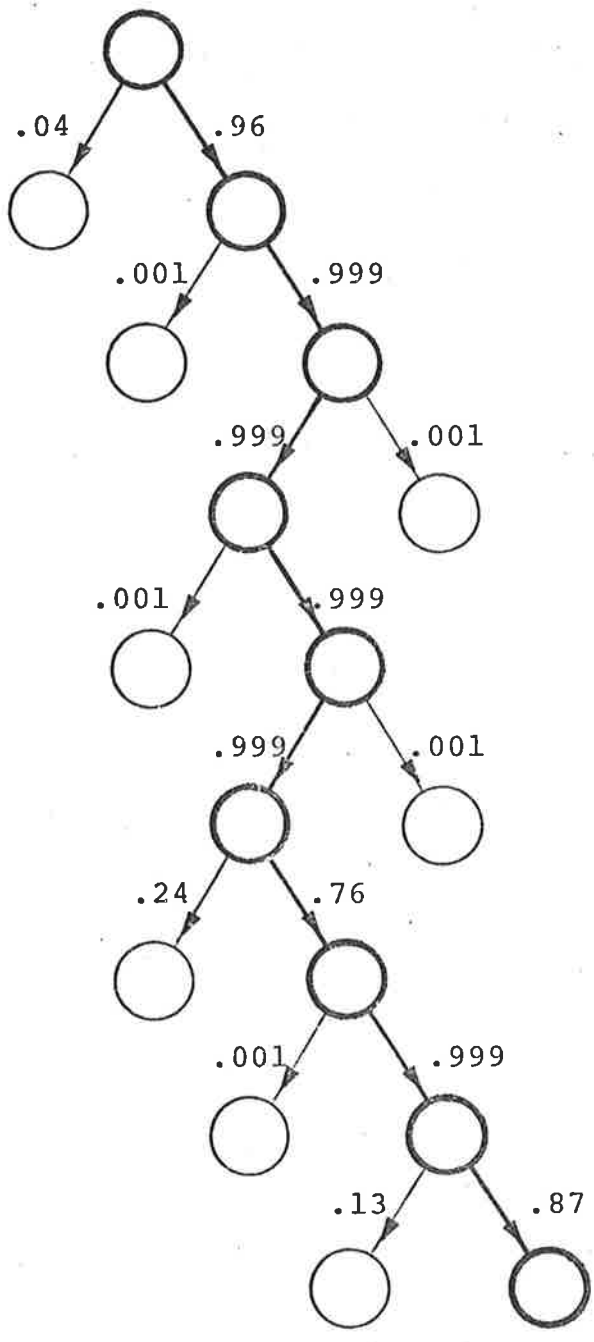


Figure 45 : A segment of the operation of Simulator B represented by a Markov chain. The heavy line indicates the path taken by the simulator Branches to the left indicate possible delays of one revolution. Branches to the right indicate possible advances.

associated with the alternate branch (P) for the information of the user.

Because of the head crash, the values of P could not be compared with the occasions on which the real system actually took the alternative branch. However, a study was made of the frequency with which P is significantly large. It will be seen from Figure 45 that P (i.e. the probability of taking the less likely branch) is often negligible (.001). In two instances, it is fairly high (.13 and .24). A much larger number of simulated disc accesses was studied to determine just how P is distributed within its possible range of 0 to 0.5. The result is shown in Figure 46, which shows, for a given threshold probability P_t, the percentage of decision points for which $P > P_t$. Thus, if a probability of 0.1 for a discrepancy is regarded as significant, then about 26% of track changes will give appreciable possibility of a discrepancy occuring.

P is the probability that, at a given branch, the simulator will choose a path different from that taken in the real system. From this it can be seen that the probability of the simulator taking the correct branch in each of n successive track changes is

$$\prod_{i=1}^{n} (1 - P_i)$$

where P_i is the value of P at the ith decision point. In Figure 45, the probability of the simulator's whole path being correct is 0.63.

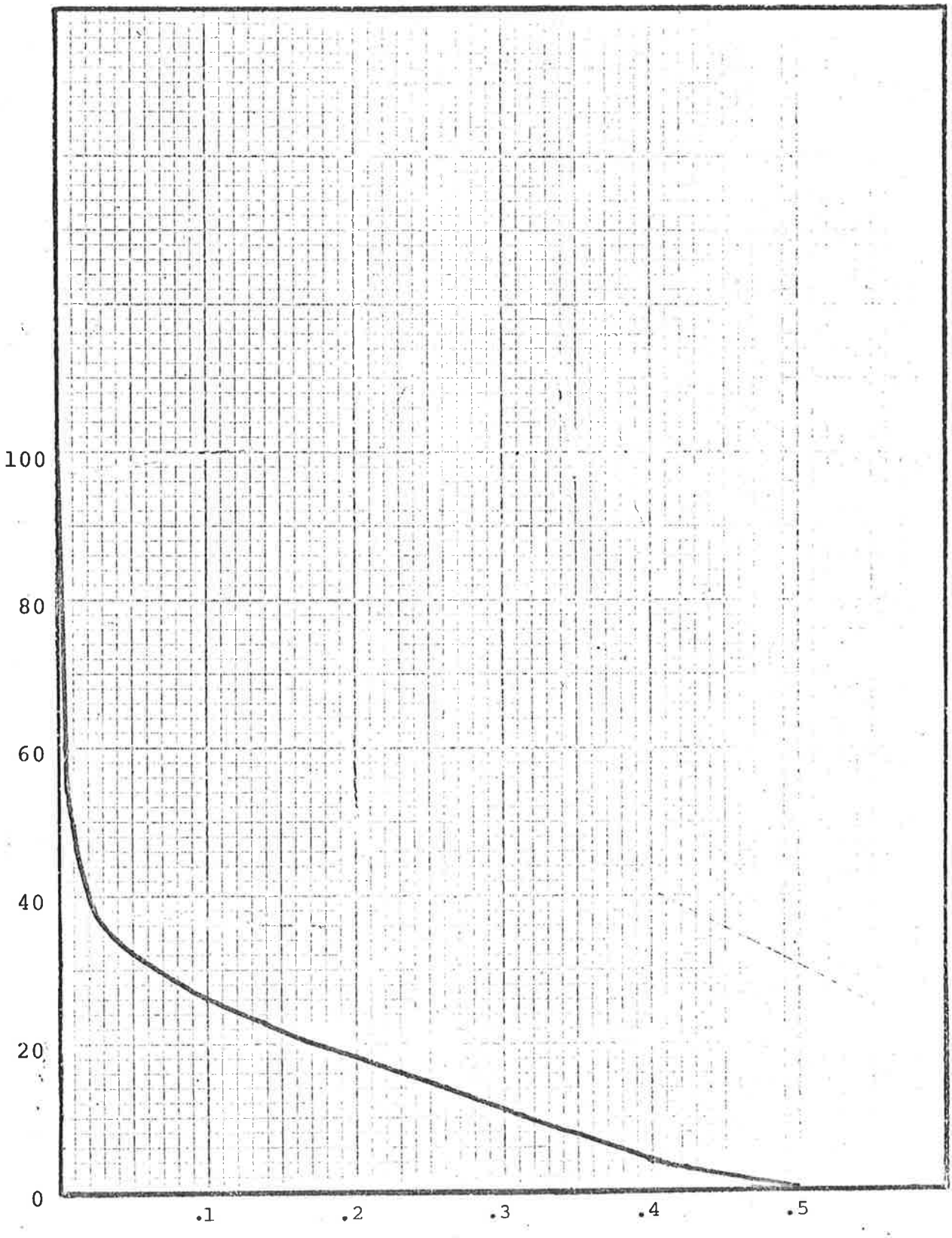


Figure 46 : The cumulative distribution of P, the probability of a track change causing a discrepancy between real and simulated systems.

The calculation of P is dependent on the standard deviation of the distribution of track change times, which was 9 milliseconds. It was intended to carry out more exhaustive measurements of track change times to see if a better mechanism could be found for simulating them than the regression formulae derived in 6.4.3. It was hoped that such a study would enable the times to be predicted from a narrower distribution, which would have the effect of lowering the standard deviation. Unfortunately the head crash on the 6603 disc prevented such measurements, so the standard deviation of 9 milliseconds had to be kept.

It should be noted that the decision points where P is significant will, on average, be evenly divided between advances and delays of equal magnitude. Over a long period the real system, even though it takes a different (and less likely) path than the simulated system, would behave in a similar way. For long periods, where the detail of what happens at any instant is not of interest, the possibility of discrepancies due to track changes can therefore be disregarded.

8.6 SUMMARY

Validation of Simulator B was split into two phases. Firstly, the simulation of the intricacies of Scope logic was validated. The processes involved in this were all deterministic, and so the validation could be based on an exact, event-for-event comparison between the real and simulated systems. The validation proved difficult in practice, since it required

that the Scope system be set up in such a way as to use only one track of the 6603 disc. Despite the difficulties, the validation was successfully carried out, and the comparison obtained was as good as that obtained in the validation of Simulator A. This shows that a great deal of detail can be removed from a simulation model (as in the transition from Simulator A to Simulator B) without adversely affecting its accuracy, so long as care is taken about just how detail is removed. Simulation at the more detailed level is necessary in order to find where detail is not critical.

After the deterministic validation, the stochastic track changes were implemented. These do not make the system wholly stochastic, but require that it make, at times, a random choice between two deterministic courses of action.

Validation of this process was precluded by the unforseen demise of the 6603 disc in the real system. Methods of bypassing the problem were considered and found wanting. All that could be done, therefore, was to determine how important a part the stochastic effects played in the system, and how often they could be expected to cause deviation between the real and simulated systems. Over long periods of time, these discrepancies cancel one another.

Except for its prediction of discrepancies due to track changes (which in any case are only informative and do not affect the operation of the model), Simulator B can be regarded as validated. The results of the validation show that its behaviour is very close to that of the real system in all respects.

CHAPTER 9

SOME EXAMPLES OF THE USE OF THE SIMULATOR

For the purposes of illustration, a number of simple investigations were carried out using the simulator. These enabled the simulator to be evaluated in a situation closer to real life than in the validation runs. Ease of specifying the details of jobs and of the operating system could be estimated for the sorts of problems the simulator was designed to study.

Briefly, the investigations carried out were :-

1. examination of the effects on the system of moving the PPU program CIO from central memory to the disc.
2. identification and improvement of bottlenecks in the system.
3. examination of the effects of different priority schemes on the performance of the system.

These investigations are typical of the sort of problems considered by those interested in predicting the performance of a computer system.

9.1 PRELIMINARY CONSIDERATIONS

There are two problems that must be considered in almost any investigation of the performance of a computer system. The first is the determination of the kind and mix of jobs the system will be processing. The second is to decide that measures will be used to judge the operation of the system. Both these problems are difficult. The

difficulties, and the assumptions made in this study, are now described.

9.1.1 Job Classes

The types of job submitted to a computer and the way in which these types are mixed are subject to complex fluctuations. The job mix will vary with the type of computer and with the nature of the installation. Even with one particular system, there will generally be fluctuations depending on the time of year, day of the week, and time of day. In most installations there will be a growth trend as well.

The main characteristics of a job, so far as its effect on the system is concerned, are its size (i.e. the amount of processing it requires) and the type of processing needed (i.e. dominated by central processing or by file accessing). With jobs run on the 6400 under Scope, a quantitative measure of both these characteristics can readily be derived from the CPU and PPU processing times. As a measure of job size, the system time can be used, where :

System Time (ST) = 4 x CP Time + PP Time.

The weighting of CPU time by four is somewhat arbitrary, but has some foundation in that this factor is used in the 6400 accounting programs, indicating that a unit of CPU time is generally thought to be "worth" four units of PPU time. As a measure of how much file accessing, as opposed to central processing, a job does, the processing ratio can be used, where :

Processing Ratio (PR) = PP time/CP time.

As an indication of what this quantity means, consider the fact that jobs involving long compilations or assemblies usually have a PR very close to one. This type of job would therefore involve "equal" amounts of file accessing and central processing. Jobs doing trivial processing (e.g. copying) on lengthy files would have a higher PR, while jobs involving much processing with little file accessing would have a lower PR. Because of the PPU overheads involved in initializing and terminating a job, the PR for small jobs is generally much higher than it is for large jobs doing the same sort of processing. In other words, System time and Processing Ratio are not completely independent.

A sequence of about 200 successive jobs that were submitted to the 6400 on a typical working day were examined, and their system time and processing ratios calculated. A scatter diagram of these jobs, plotted against the two measures, is shown in Figure 47. It can be seen that the jobs fall into several clusters, shown in Figure 47 by the boundaries tabulated in Figure 48. These clusters will be called job classes.

Defining the class of a job purely in terms of its processing times has the disadvantage that it bears no relation to the origin of the job. Since different users would be expected to submit different classes of jobs, it was decided to look at the orginators of the jobs in the various classes, to see if there was any definite relationship between the two. The jobs plotted in Figure 47 were

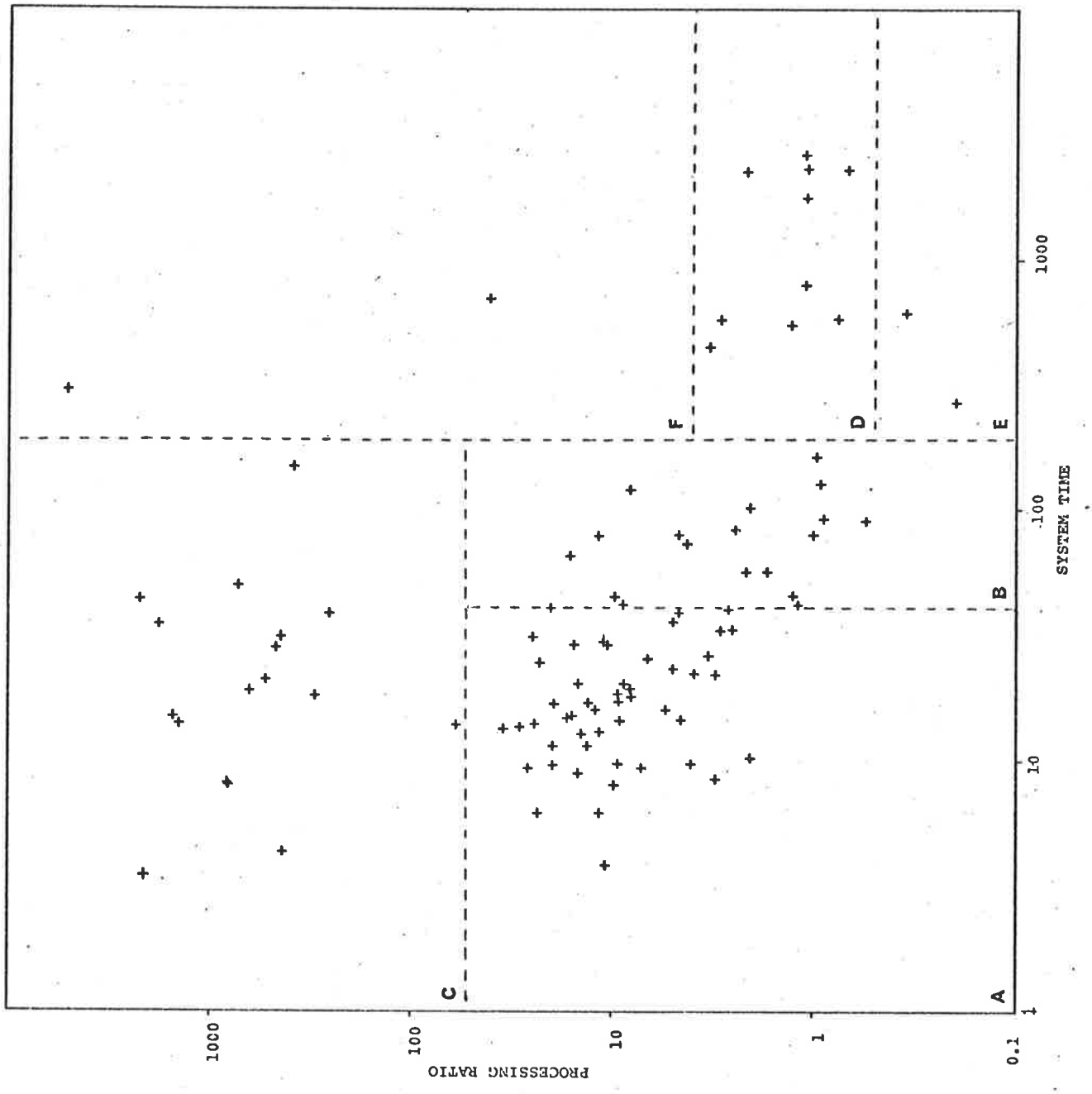


Figure 47 : Scatter plot of about 250 jobs submitted to the 6400. Note that the scales are logarithmic. The division of the jobs into classes is shown.

Class A	54.7%	ST < 40	PR < 50
Class B	20.9%	40 < ST < 200	PR < 50
Class C	16.2%	ST > 200	PR > 50
Class D	5.1%	ST > 200	.5 < PR < 4
Class E	1.3%	ST > 200	PR < .5
Class F	1.7%	ST > 200	PR > 4

Figure48 : The proportions in which jobs of various classes are mixed in the 6400 input stream during term time (from a sample of 250 jobs).

found to originate as follows :-

Class A : Simple compilation-execution jobs submitted by students learning a programming language for the first time.

Class B : Jobs submitted by more advanced students and research workers requiring more (and more sophisticated) resources than the jobs in Class A. Most of the runs described in this chapter, for example, were Class B jobs.

Class C : Jobs restricted almost entirely to file handling (copying, listing and permanent file manipulations). These jobs are run by more sophisticated users to set up files for later jobs.

Class D : Long jobs using similar amounts of PPU and CPU time (e.g. long compilations and assemblies). These usually come from systems programming staff.

Class E : Long jobs made up largely of central processing. These are rare, and usually involve mathematical applications.

Class F : Long jobs with appreciable central processing and very large amounts of PPU processing. These jobs come from such users as the university administration, the library, etc. They are usually rare but can dominate at some times of the year (e.g. during the processing of enrolments).

There were very few disagreements between classification by ST and PR and classification by origin. Only about 25% of jobs fell into different classes under the two different schemes. The class divisions used would therefore seem to be natural and consistent ones.

9.1.2 Job Mix for Simulation Runs

The distribution of jobs among the various classes in Figure 47 is tabulated in Figure 48. As was mentioned at the beginning of this section, the distribution varies with time. For example, during university vacations, there are far fewer Class A jobs. During term, especially when a large class has a programming exercise due, Class A jobs dominate. Other classes of jobs are also subject to periodic fluctuations, although not usually as dramatic as those of Class A. The distribution shown in Figure 48 is from a typical day during the university term.

Simulator B is designed to predict the effect on the system of changes made to it. Because of the variation in job mix, it is necessary to make such predictions over a range of job mixes, and to make a final judgement only in the light of the overall result.

For the purposes of the studies described in this chapter, the distribution shown in Figure 48 (a typical term-time job mix) is of greatest interest, and job streams selected from this distribution will dominate in the experiments that follow.

In order to simplify the specification of jobs to the simulator, a job generating program was written. This program, given the distribution of the jobs among the classes, generates a random sequence of jobs conforming with this distribution. This stream is in a form suitable for feeding to the simulator, as described in 7.3.3. For example, the job generator, asked to produce a job stream where the distribution A:B:C:D:E:F was 3:2:1:1:0:0, produces the sequence of job descriptions shown in Figure 49. The jobs shown have a simpler structure than most real jobs would, but the amount and type of processing they require are similar to those of real jobs. If, for a particular job, a more exact specification is required, the job generator can be bypassed, and the job description given explicitly.

For most of the runs to be described, a stream of thirty jobs were used with the classes represented in the ratio 17:6:5:2:0:0. When this stream was run with the simulator, it produced the distribution shown by the arrows in Figure 50. The distribution of real jobs from Figure 47 is also shown for comparison. It can be seen that the thirty jobs generated fit fairly well into the overall distribution.

9.1.3 Criteria for Evaluation

It is sometimes difficult, given two versions of an operating system, to say which is better. It is even difficult to say what "better" means. The definition varies, depending on whose interests are at stake.

```
DDDDD01 1000 060000    19
CREATE(Q)                              1.
REWIND(Q)
COMPILE(Q,LGO,OUTPUT)                172.0
END
CCCCC02 0005 020000    16
CREATE(X)                               .01
REWIND(X)
WAIT                                  17.3
COPY(X,Y)                               .01
REWIND(X)
COPY(X,Z)                               .01
END
AAAAA03 0005 040000    46
COMPILE(INPUT,LGO,OUTPUT)
REWIND(LGO)
LOAD(LGO)
EXECUTE                                 .6
END
BBBBB05 0040 050000    51
COMPILE(INPUT,LGO,OUTPUT)              3.2
LOAD(LGO)
EXECUTE.                               6.9
END
AAAAA06 0005 040000    36
COMPILE(INPUT,LGO,OUTPUT)
REWIND(LGO)
LOAD(LGO)
EXECUTE.                                .9
END
AAAAA07 0005 040000    51
COMPILE(INPUT,LGO,OUTPUT)
REWIND(LGO)
LOAD(LGO)
EXECUTE.                               1.1
END
```

Figure 49 : A job stream produced by the job generator.

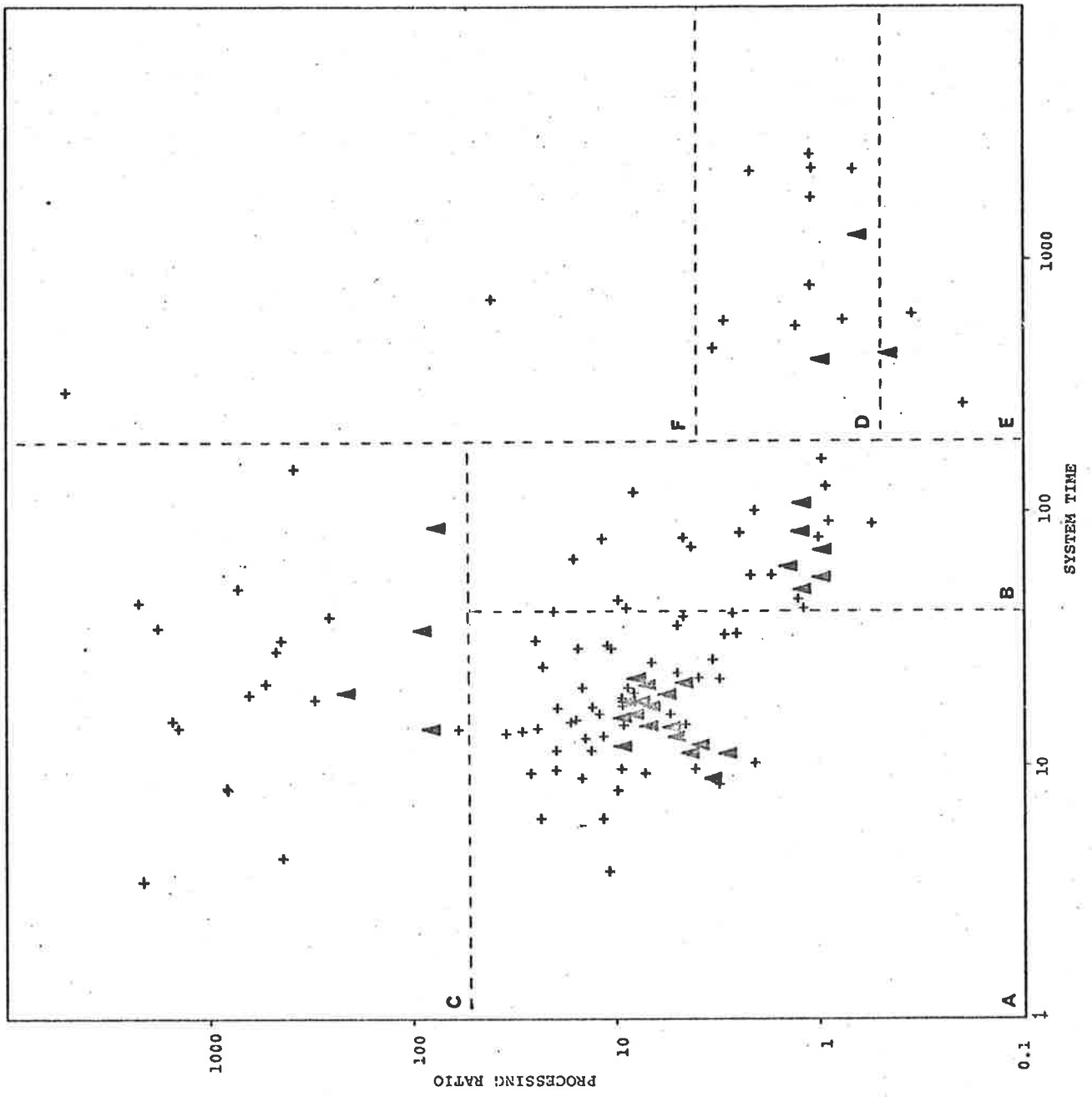


Figure 50 : A comparison between 250 real jobs (crosses) and 30 jobs produced by the job generator using the same distribution (triangles).

The user of a computer system will judge the performance of the system by the time he has to wait for a job to run through it, i.e. the "turn-around time". As the user sees it, turn-around time is the time between when his input deck is submitted for running, and when the output becomes available. Simulator B does not simulate the processes of collecting card decks and distributing outputs. Turn-around time will therefore be defined as the time between the beginning of reading and the end of printing for a given job. Job collection and distribution are a function of the operators, not the operating system. For the user, the shorter the turn-around time the better, but this is not the only consideration. Distribution of turn-around time is also important, since it is desirable that the user be able to predict how long the job will take to run. This is difficult if turn-around times vary widely in unpredictable ways. Turn-around time needs to be considered for the various job classes separately, since optimising the system for one class will often slow the processing of the others.

The manager or owner of the system is usually more interested in obtaining the maximum usage from it. The measure for this is "throughput", the number of jobs run in a given time. Turn-around time is of little importance, except in so far as it keeps the users happy.

The designer of an operating system often has little idea of what kind of jobs will ultimately be submitted to it. His criteria for optimization must therefore be independent of the jobs. The

criteria for optimizing the Scope 3.2 operating system was, largely, usage of the CPU. Usage of other valuable resources, such as central memory, is also an important measure of system efficiency.

In judging the performance of an operating system, then, several factors need to be considered. The most important are turn-around time and its distribution, throughput and usage of valuable resources. The report printed by Simulator B (described in 7.3.5 and Figure 35) contains information about all these factors.

9.2 CHANGING THE RESIDENCE OF CIO

The process of loading a PPU program is described in detail in 7.4.2. The amount of processing involved in this operation depends a great deal on the residence of the PPU program, i.e. whether it is stored in central memory or on the disc. If the program is in central memory, it is simply read into the PPU. If it is on the disc, the PPU must issue a stack request for the disc to be accessed. This section describes a simulation experiment involving the residence of a PPU program.

9.2.1 The Nature of the Problem

The most frequently used PPU program in the Scope system is CIO, which processes all requests by CPU programs to access any file. Because of the high frequency with which it is used, CIO is nearly always stored in central memory. Using Simulator B, the effects of moving it to the disc were predicted. Of course, for

a program such as CIO, there would never by any question but that it should reside in central memory. It was chosen for this experiment simply to show how a trivial change in the operating system can produce a dramatic effect in its performance. For programs used less frequently than CIO, it is often difficult to decide which is worth more - the efficiency of having the program in central memory, or the extra memory that would be available if the program were on the disc. Simulator B can be used to study this problem for any of the PPU programs it simulates.

9.2.2 Setting up the Experiment

The job generator (described in 9.1.2) was used to generate a stream of thirty jobs with the same class distribution as that in Figure 48. This stream was run using Simulator B, simulating the "standard" system, with CIO in central memory. The results from this run, which are discussed below, are shown in Figure 51.

The simulator was then set up with CIO disc resident. This was achieved by adding two cards to the simulation control stream, one redefining the residence of CIO and giving its position on the disc, and the other redefining the amount of central memory available to jobs (this was 250 words more because CIO was no longer in central memory). The stream of thirty jobs was run through the altered system, and the results tabulated as before, in Figure 52.

9.2.3 Results

Since this is the first time the Simulator has been described working with life-like jobs, the results of Figure 51 may be of interest in themselves. Central memory requirements played an important part in the behaviour of the various job classes. In this system there were about 140000_8 words of central memory available for allocation to users' jobs. The jobs in the various classes used in the study had the following central memory requirements (in octal) :

Class A	40000
Class B	50000
Class C	20000
Class D	60000

Because of their low memory requirements, jobs in class C received better treatment than their priority (based on estimated CP time and memory) warranted, since often they could be fitted in when other jobs (with higher priority) could not. This accounts for their low average turn-around time (1½ minutes). The turn-around times for the other classes were about as expected. The difference between the average and maximum turn-around times is not great, indicating that, if the class of a job is known then a fairly reliable prediction can be made, intuitively, about how long it will take to run.

The results are also affected by the starting and running down of the simulated system. The 30 jobs simulated were read in

CLASS	A	B	C	D	ALL
NUMBER OF JOBS	17	6	4	3	30
AV. TURN-AROUND TIME (MIN)	2.59	4.19	1.50	9.80	3.49
MAX. TURN-AROUND TIME (MIN)	3.16	5.30	3.37	14.20	14.20
THROUGHPUT (JOBS/HOUR)					116
CPU UTILIZATION (P.C.)					59.80
PPU UTILIZATION (P.C.)					24.30
CENTRAL MEMORY UTILIZATION (P.C.)					76.20
AVERAGE LENGTH OF INPUT QUEUE					4.41
AVERAGE LENGTH OF STACK					.72

FIGURE 51. SYSTEM PERFORMANCE UNDER STANDARD CONDITIONS.

CLASS	A	B	C	D	ALL
NUMBER OF JOBS	17	6	4	3	30
AV. TURN-AROUND TIME (MIN)	3.49	6.02	2.28	14.31	4.92
MAX. TURN-AROUND TIME (MIN)	4.61	7.66	5.43	20.67	20.67
THROUGHPUT (JOBS/HOUR)					85
CPU UTILIZATION (P.C.)					34.80
PPU UTILIZATION (P.C.)					35.50
CENTRAL MEMORY UTILIZATION (P.C.)					82.80
AVERAGE LENGTH OF INPUT QUEUE					5.80
AVERAGE LENGTH OF STACK					1.40

FIGURE 52. SYSTEM PERFORMANCE WITH CIO DISC RESIDENT.

a continuous batch at the beginning of the run. The shorter (classes A and C, and to some extent B) jobs, having a high priority, ran first, leaving the longer jobs to be run later. By the time the longer jobs ran, therefore, there was no competition with shorter jobs, all of which had left the system.

Comparison of Figure 52 with Figure 51 shows that, in all respects, the system in Figure 52 (with C10 on the disc) is about 1.4 times less efficient than that of Figure 51. The only quantities not showing this degradation are the PPU utilization and the central memory utilization. The first increased because of the extra disc accessing involved, and the second because of the additional time for which jobs occupied control points.

9.3 INVESTIGATION OF BOTTLENECKS

One of the most important applications of a computer simulation is the identification of bottlenecks in the system, and the study of how they can be eliminated or improved. The informal investigation to be described now shows how the simulator could be used to carry out such an investigation.

9.3.1 The Nature of the Problem

Study of the behaviour of the simulated system during the run associated with Figure 51 (i.e. under standard conditions) showed that, of the many queues in the system, the only two on which jobs spent appreciable amounts of time waiting idle were the input queue (waiting for control points or central memory)

and the request stack (waiting for access to the disc). The average lengths of these queues with respect to time is shown in Figure 51.

The bottlenecks in the system are, then, access to the disc and availability of either central memory or control points. Control points were readily eliminated from consideration by noting that at no time during the simulation run of Figure 51 were all available control points being used. The reason jobs stayed on the input queue was lack of central memory.

Once the bottlenecks were identified, an experiment was planned to investigate the effects of removing them. The disc problem was relieved by hypothetically decreasing all access times on the disc by 50%, and the central memory problem was relieved by hypothetically increasing the amount of central memory in the system by 50%.

9.3.2 Setting up the Experiment

Increasing the speed of the disc required changes, one to the formulae for the track-change time, and one to the revolution time (from 65.5 to 43.7 milliseconds). The same batch of jobs as for the run of Figure 51 were run through the simulated system with this modification. The results, discussed below, are shown in Figure 53.

Increasing the amount of central memory in the system were achieved simply by placing an EQUIPMENT card in the simulation control stream, redefining the central memory to be 300000_8 words

long (in the standard system there are 200000_8 words). The same job stream was run through this system, with the results shown in Figure 54.

Finally, the same job stream was run again with both the faster disc and more memory, so that both bottlenecks were improved. The results from this run are shown in Figure 55.

9.3.3 Results

The effect of speeding up the disc can be seen by comparing Figure 53 with Figure 51. The performance of the system was improved by between 20% and 40%, depending on the measure chosen. The average length of the stack (the queue for access to the disc, the heavy use of which identified the disc as a bottleneck) was reduced by about 40%.

The effect of having more central memory is shown in Figure 54, which should also be compared with Figure 51. A surprising feature of these results is that the extra memory in fact makes the system worse so far as Class C and Class D jobs are concerned. This result was unexpected, but once it is seen, its cause can be found without much difficulty.

It was stated in 9.2.3 that in the standard system, class C jobs get faster turn-around than they really should because of their small memory requirements. This phenomenon depends on there being a shortage of central memory. When the storage is relieved, as in Figure 54, the advantage that class C jobs had over the others

is removed, and their turn-around times are more in accordance with their priority.

Class D jobs are unique in that most of their turn-around time is spent executing at a control point, rather than waiting on the input queue. This is because of the large amounts of processing they require. With more memory, the system has greater overheads (because more jobs are executing at any one time) and the queues concerned with such overheads get longer (notice, for example, how high the average stack length is in Figure 54). Thus, once it gets to a control point, a job will take longer to run in this system than in the standard system. The saving due to increased memory is limited to time spent on the input queue. Short jobs spend only a short time at a control point anyway, so that the effect of extra overheads is far outweighed by time saved on the input queue. For class D jobs, however, the opposite is the case, and their performance suffers.

What has in fact happened in the system of Figure 54 is that the performance of class A and B jobs has been improved at the expense of class C and D jobs. The simulation shows just what mechanisms operate to bring about this effect.

The results of Figure 54 illustrate two important facts about computer systems. Firstly, they are so complex that intuitive prediction of their behaviour is unreliable. Secondly, it is always necessary to consider many different factors that contribute to the balance of a system. If relieving the effect of one bottleneck

CLASS	A	B	C	D	ALL
NUMBER OF JOBS	17	6	4	3	30
AV. TURN-AROUND TIME (MIN)	1.94	3.27	1.22	8.19	2.73
MAX. TURN-AROUND TIME (MIN)	3.76	4.15	2.64	12.77	12.77
THROUGHPUT (JOBS/HOUR)					141
CPU UTILIZATION (P.C.)					69.60
PPU UTILIZATION (P.C.)					25.30
CENTRAL MEMORY UTILIZATION (P.C.)					77.30
AVERAGE LENGTH OF INPUT QUEUE					3.70
AVERAGE LENGTH OF STACK					.45

FIGURE 53. SYSTEM PERFORMANCE WITH A FASTER DISC.

CLASS	A	B	C	D	ALL
NUMBER OF JOBS	17	6	4	3	30
AV. TURN-AROUND TIME (MIN)	1.36	3.51	2.18	10.31	2.79
MAX. TURN-AROUND TIME (MIN)	1.84	4.32	4.98	14.36	14.36
THROUGHPUT (JOBS/HOUR)					119
CPU UTILIZATION (P.C.)					58.90
PPU UTILIZATION (P.C.)					24.80
CENTRAL MEMORY UTILIZATION (P.C.)					63.20
AVERAGE LENGTH OF INPUT QUEUE					2.10
AVERAGE LENGTH OF STACK					1.45

FIGURE 54. SYSTEM PERFORMANCE WITH 50 P.C. MORE CENTRAL MEMORY.

CLASS	A	B	C	D	ALL
NUMBER OF JOBS	17	6	4	3	30
AV. TURN-AROUND TIME (MIN)	.98	2.75	1.80	8.51	2.20
MAX. TURN-AROUND TIME (MIN)	1.54	3.60	3.95	12.11	12.11
THROUGHPUT (JOBS/HOUR)					140
CPU UTILIZATION (P.C.)					69.50
PPU UTILIZATION (P.C.)					28.00
CENTRAL MEMORY UTILIZATION (P.C.)					63.00
AVERAGE LENGTH OF INPUT QUEUE					1.58
AVERAGE LENGTH OF STACK					1.06

FIGURE 55. SYSTEM PERFORMANCE WITH A FASTER DISC AND MORE CENTRAL MEMORY.

increases the effect of another, (as in Figure 54, where removing central memory restrictions has aggravated the problem of access to the disc), it should perhaps be considered whether the effort might be better spent elsewhere, on relieving the aggravated bottleneck.

Finally, Figure 55 shows how the system performs with both the disc and central memory improved. All the jobs run faster except the class C jobs, which still run more slowly than in the standard system because they are so favoured in the latter. The increased stack length in Figure 55 shows that a greater increase than 50% is required in disc speed to properly balance the 50% increase in central memory. It is also interesting to note that in the run associated with Figure 55, jobs began to spend appreciable periods on the CPU queue, indicating that the two original bottlenecks had been improved to the extent that others were beginning to show.

9.4 INVESTIGATION OF PRIORITY SCHEMES

When a control point is free, the Scope 3.2 system selects from the input queue the highest priority job that will fit in the memory available. The priority of jobs on the input queue is thus an important factor in the behaviour of the system. This experiment examines the effect of a change in the scheme for allocating priorities to jobs on the input queue.

9.4.1 The Nature of the Problem

In the standard system (i.e. that of Figure 51), when a job is read in and put on the input queue, its priority is calculated using the formula :

$$\text{Priority} = 4500_8 - \frac{M*T}{20000_8}$$

where M is the number of words of central memory the job requires and T is its CPU time limit in seconds. By this formula, jobs requiring less memory and CPU usage are given higher priority. Jobs in the various classes considered previously, for example, receive initial priorities in octal as follows :

Class A	4467
Class B	4461
Class C	4474
Class D	1501

In addition, an ageing mechanism exists which adds 1 to the priority of each job on the input queue about every seven seconds. This results in a "first come first served" discipline for jobs with otherwise equal characteristics.

The alternative priority scheme to be investigated is only slightly different from the standard one. Class D jobs are given an initial priority of 6000_8. Such a scheme may seem somewhat unrealistic at first sight. However, Class D jobs usually come from systems programmers, who often work to tight schedules. It is reasonable to envisage a system where such people are given a

high priority.

9.4.2 Setting up the Experiment

In order to give a wider appreciation of the effect of the different priority scheme, simulations were done using two different job mixes. One was the same as in previous runs, while the other had a greater proportion of longer jobs, with the classes represented in the ratio 9:9:8:4 for A, B, C and D respectively. The results given by the standard system running this job mix are shown in Figure 56 and are discussed below.

The simulator was then changed to reflect the new priority scheme. This change involved the addition of a single card to the simulator program. A simulation run was then performed for each of the two job streams. The results for the normal job mix are shown in Figure 57, and those for the second job mix are shown in Figure 58.

9.4.3 Results

Comparison of Figure 56 with Figure 51 shows how the system performs with a different job mix. The turn-around times of jobs in classes A, B and C are all improved, at the expense of the class D jobs. The throughput is considerably lower, as would be expected since more of the longer jobs are being processed. The difference in turn-around times is due to the fact that in Figure 51 the jobs arrived in such a way that a Class D job gained a control point early in the run, leaving enough memory for only one job from classes A

or B to run at the same time. This did not happen in Figure 56, where all the jobs of Classes A, B and C were almost completed before the first Class D job ran.

The effect of the changed priority scheme can be seen by comparing Figure 57 with Figure 51. The change in the characteristics of the system is dramatic, with much poorer turn-around for jobs in classes A and B. This is because class D jobs are favoured in the system, so that A and B jobs have to wait on the input queue until all the class D jobs had finished. Class C jobs were not greatly affected because of their small memory requirements. Note that, although the average turn-around time is much worse in Figure 57 than in Figure 51, the throughput is higher, indicating a more efficient use of resources, as is indeed shown by the CPU, PPU and memory utilization, which have risen appreciably.

Figure 58, which should be compared with Figure 56, shows the changed priority scheme with the second job mix. The only significant change the new priority scheme produces in this case is to force Class B jobs to wait until all the Class D jobs have run. Class A jobs, unlike those of Figure 57, all ran before the first Class D job. These phenomena illustrate an important implication of the memory bottleneck. When there is a shortage of memory, the completion of a job releases just enough memory for another job of that class to take its place. There is a tendency, therefore, for all the jobs of a particular class to run one after the other, irrespective of priority considerations, as in fact happened for

CLASS	A	B	C	D	ALL
NUMBER OF JOBS	9	9	8	4	30
AV. TURN-AROUND TIME (MIN)	1.21	2.62	1.36	14.19	3.40
MAX. TURN-AROUND TIME (MIN)	2.10	3.44	1.96	19.87	19.87
THROUGHPUT (JOBS/HOUR)					90
CPU UTILIZATION (P.C.)					62.70
PPU UTILIZATION (P.C.)					21.50
CENTRAL MEMORY UTILIZATION (P.C.)					75.80
AVERAGE LENGTH OF INPUT QUEUE					2.61
AVERAGE LENGTH OF STACK					.74

FIGURE 56. SYSTEM PERFORMANCE WITH A DIFFERENT JOB MIX.

CLASS	A	B	C	D	ALL
NUMBER OF JOBS	17	6	4	3	30
AV. TURN-AROUND TIME (MIN)	10.65	10.25	1.94	8.46	9.19
MAX. TURN-AROUND TIME (MIN)	10.05	12.61	4.66	11.63	12.61
THROUGHPUT (JOBS/HOUR)					133
CPU UTILIZATION (P.C.)					66.00
PPU UTILIZATION (P.C.)					30.00
CENTRAL MEMORY UTILIZATION (P.C.)					92.00
AVERAGE LENGTH OF INPUT QUEUE					17.50
AVERAGE LENGTH OF STACK					.80

FIGURE 57. SYSTEM PERFORMANCE WITH A DIFFERENT PRIORITY SCHEME.

CLASS	A	B	C	D	ALL
NUMBER OF JOBS	9	9	8	4	30
AV. TURN-AROUND TIME (MIN)	1.49	13.18	1.78	13.05	6.62
MAX. TURN-AROUND TIME (MIN)	2.00	18.03	2.50	18.82	18.82
THROUGHPUT (JOBS/HOUR)					91
CPU UTILIZATION (P.C.)					65.00
PPU UTILIZATION (P.C.)					22.00
CENTRAL MEMORY UTILIZATION (P.C.)					83.00
AVERAGE LENGTH OF INPUT QUEUE					7.30
AVERAGE LENGTH OF STACK					.73

FIGURE 58. SYSTEM PERFORMANCE WITH A DIFFERENT PRIORITY SCHEME AND JOB MIX.

all four classes in Figures 57 and 58. In Figure 58, for example, all the Class A jobs run before any of the Class D jobs, despite the fact that the Class D jobs, in this system, have a higher priority. This is because several Class A and Class C jobs were read in and assigned to control points before the first Class D job arrived. These class A and C jobs all finished at different times. Each time one finished, either 40000 or 20000 words of central memory was released (depending on whether a Class A or a Class C job finished). This released memory was enough for another class A (or C) job to be brought to a control point. Never at any one time was the whole 60000 words required by Class D jobs available, until no more Class A and C jobs remained on the input queue. At this stage, the available free memory remained unused until the next job finished, when there was enough for a Class D or Class B job to run. A Class D job, having the higher priority, was selected.

9.5 SUMMARY

Before Simulator B could be used for studying the behaviour of the system, some consideration had to be given to two problems : job mix and system evaluation. The job mix problem was approached by first devising measures by which jobs could be classified. A large number of jobs was taken from the accounting files of the real system and, using these measures, a classification scheme was formed. Using this scheme, a job mix could be specified to a job generation program which produced, at random, job streams for the simulator. When these job streams were run

through the simulated system, they showed similar characteristics to the real jobs that had been studied.

A number of measures were considered for the performance of the operating system, given a particular mix of jobs. It was realized that the appropriateness of each measure depended on what aspect of the system was being evaluated, so that each of the measures was studied in each of the subsequent investigations.

The first investigation carried out using the simulator was a very simple one - to study the effects of moving the PPU program CIO to the disc. It was found, as expected, that all aspects of the performance of the system were seriously degraded.

The second investigation was more realistic - the identification and improvement of bottlenecks. Identification was easily achieved by studying the various queues in the system during simulation. The major bottlenecks proved to be access to the disc and the availability of central memory. The effects of a faster disc and more memory were studied, both separately and together. The results were sometimes unexpected ; a change which one would intuitively expect to produce unqualified improvement, would, in some cases, produce poorer results.

The third investigation involved the priority scheme for jobs on the input queue. For this, two input streams with different job mixes were used. In the case of the first stream, the change that was introduced into the priority system produced dramatic changes in the performance of the system. The second stream was not affected as greatly. Once again, the results were not as might be intuitively expected.

The studies described in this chapter were not intended to be rigorous "production" simulation projects, but were designed to illustrate how the simulator could be used. The fact that such simple studies can evoke such unpredictable behaviour in the system illustrates the difficulties of studying computer systems, and the effectiveness of simulation as a tool for such study.

CHAPTER 10

CONCLUSION

The last few chapters have described in detail the design, development, validation and use of a simulator for Scope 3.2 on the 6400 computer system. This process first involved the accumulation of much information about the system being simulated. Measurement of the behaviour of system programs required the development of an independent simulator, working at the instruction level (Simulator A, Chapters 4 and 5). Information about the performance of the input-output equipment was obtained by physical experiments (Chapter 6). This information was then collected into an event-based simulation program (Simulator B, Chapter 7). Once written, this program was subjected to detailed validation (Chapter 8), which indicated that its behaviour accurately reflects that of the real 6400 system, both qualitatively and quantitatively.

10.1 AIMS AND ACHIEVEMENTS OF THE STUDY

The obvious practical achievement of this study is the development of an accurate simulation model. Its more significant result, however, is the knowledge gained about the processes of constructing and testing such a model.

In Chapter 1 a number of topics were introduced which were to be particularly considered during the development of the simulator. These will now be reviewed in the light of the successful completion of the model.

10.1.1 Levels of Detail

Fundamental to the whole approach to the development of this simulator was the idea of starting with information as detailed as possible, and eliminating detail only when it could be replaced by simpler processes without significant change in the behaviour of the simulated system. This development was in the opposite direction to that of most simulation studies, which start with a coarse model, which is refined when its results are found to be inadequate. The method of going from a detailed to a less detailed model, as in this study, is longer and more difficult, but it has considerable advantages. The information generally supplied about a computer is at a very detailed level (instruction execution times and so on). Starting the simulation at this level enables this precise information to be used explicitly by the simulator. At the start, the unit of processing for simulation is the instruction, whose characteristics are known exactly. As simulation proceeds, it supplies information about larger segments of processing, and this information is of comparable accuracy to the instruction times on which it is based. Once these larger units are known well enough, they become the units for the next level of simulation, which, although less detailed, retains a similar level of accuracy. The process of forming the simulation model is thus precise and well defined.

10.1.2 Validation

A major objective of this simulation study was rigorous validation, which is often neglected or glossed over (55).

The validations described in Chapters 5 and 7 show that it is possible to undertake detailed validation of a simulation model, although somewhat unusual methods may be needed to achieve this end, for example the one-track system described in 8.2.1. At all times, validation was based on proof that the processes in the real and simulated systems were the same, as well as on comparison of numerical results. If the processes being simulated are correct, then correct results follow automatically, so long as the correct data are available about the quantities involved in the mechanism. In a coarse simulation, when a process itself is not simulated, but is approximated by some means (for example by random sampling), validation can be based only on numerical results, and becomes very much a "black box" affair. Such models can only be validated by statistical methods, which require many more runs, and which can only show that the simulated results are statistically similar to the real results. The deterministic validations described in previous chapters, however, show, by one-for-one comparison, that the mechanism of the two systems is the same, both qualitatively and quantitatively. This seems a much more lucid way of performing a validation.

The fact that Simulator B produced results very close to those of the real system after relatively few validation runs

(8.2.3) is further evidence in favour of the approach taken to levels of detail. All the information in Simulator B had been obtained either from more detailed simulation or from explicit measurement of the real system, so it was an accurate model almost from the start.

10.1.3 Other Problems Described in the Literature

Other problems often mentioned in the literature, besides level of detail and validation, are choice of language, specification of jobs, formal description of operating system processes, and the choice of criteria to evaluate operating systems.

Choice of language was forced in this instance by Fortran being the only suitable language that had reliable software at the time the project began. The only other language seriously considered was Simula. Now that Simula has better software, and more comprehensible documentation (3), the decision may well be different, although Fortran still has the advantage of being much more widely known.

Specification of jobs was done by inventing a job description language resembling Scope control cards as described in 7.3.3. For programs not known to the simulator (i.e. users' programs), a definition of the characteristics of the programs was also necessary. Job definition was thus in a form that would be fairly familiar to prospective users of the simulator.

Some formal description of operating system processes was

found to be necessary to convert information from Simulator A into the program for Simulator B. It was found that precedence-decision networks, as used by Noe and Nutt (55, 56), but with durations associated with each arc, adequately filled this need.

The problem of evaluating an operating system has no real solution, and can only be approached by considering all the quantities that may be relevant. An important characteristic of the model developed here is that one can see the mechanisms that produced the quantities observed by looking at the simulation trace. Thus when unexpected results are obtained, as in 9.3.3, their cause can readily be identified.

One major problem which is not often discussed in the literature is the simulation of input-output equipment, particularly mass storage. Nielsen (51), however, describes how it is necessary to simulate paging drums accurately, as they are so frequently accessed in a paged memory system. Accurate simulation of such a device requires not only correct response times, but also a record of just where each piece of information is stored on the device. Nielsen forsees the need to do this for discs as well as drums, if the discs in a system are used frequently enough. In the 6400, the disc is the only mass storage device available. Although Scope 3.2 is not a paging system, access to the disc is still frequent enough to make exact simulation necessary.

A related problem, rarely mentioned, is the acquisition of reliable information about the response times of mass storage

equipment. The usual quantities given are the average and maximum access times. These are of little use when a device is being simulated as described above, with a record of where everything is stored on it. For an accurate simulation, the time for each operation in an access (latency, head switching and head movement) is needed. A large proportion of the work involved in this study was devoted to obtaining adequate information about these processes.

10.1.4 Other Achievements

A project for developing a simulation on the scale described here involves large amounts of programming. This particular project required extensive programming of three quite different types. Firstly, in Simulator A, the logic and arithmetic of the processors of the 6400 had to be simulated with complete, bit by bit accuracy. In Simulator B, a very large and complex program had to be designed in a highly modular way to simplify the modifications that may be necessary when investigating changes to the system. Finally, there was a great deal of PPU programming involving close interaction with peripheral devices. The programming in these very different areas, in itself, required considerable exploration of new or unusual techniques (for example, 5.1.2, 5.1.3, 5.3.2, 6.4, 7.2 and 8.2).

Simulator A is, in effect, a "hands on" copy of the entire system, with all registers, memory, etc. open to view. As such it has had a number of uses besides that of gathering data for

Simulator B. It has found considerable applications as a teaching aid, as it enables students to program PPUs and to try out their programs, which would be chaotic and impracticable using the real system. Systems programmers have also used Simulator A as a debugging aid. PPU programs are practically impossible to debug using the real system as it is so difficult to get information about what a PPU is doing. Simulator A, with its instruction by instruction trace of each PPU program being executed, provides an ideal answer to this problem, as described in 5.2.6.

10.2 FUTURE POSSIBILITIES

The research described in previous chapters leaves a number of interesting openings for further work in simulating the 6400, and points to several other areas where additional investigation would be useful.

10.2.1 Simulation of Scope 3.2

Some simulation projects with Scope 3.2 were described in Chapter 9. These, however, were not intended to be rigorous investigations of system performance under the conditions described ; they were simply demonstrations to illustrate the sort of investigations the simulator could perform.

There is a great deal of scope for further, more rigorous investigations of the sort described in Chapter 9. Sections 9.3 and 9.4 show how complex the performance of an operating system is. An extensive investigation of the kind described in 9.3, with many different job mixes being considered, would illustrate many facets

of the problem of system balance. The problem investigated in 9.3 was, virtually, given a certain amount to spend on improving the performance of a computer system, how best to spend it. The results of 9.3.3, while only a demonstration, show that this is not a trivial problem. Further investigation by simulation in this field could perhaps lead to some formalization of the mechanisms at work in the balance of the system, so that analytical optimization techniques could be brought to bear on the problem.

10.2.2 Further Development of the Simulator

There is no reason to suppose that the development of the simulation model ends with Simulator B. Just as Simulator A showed that large areas of processing could be treated as single units, so Simulator B can be used to find which of its processes can be simplified, with minimal effect on its behaviour and results. This would lead to a coarser and faster Simulator C. If the transistion from Simulator A to Simulator B is any guide, Simulator C would not necessarily give much less accurate results than Simulator B, but its ability to trace the mechanisms at work in the system would be less.

10.2.3 New Operating System

In Chapter 1 it was pointed out that Scope 3.2 was an important operating system because it is the most sophisticated 6400 system designed exclusively for batch jobs. Later systems place more emphasis on interactive work, with corresponding changes

in the philosophy of the system design.

The current operating system for the 6400 is Scope 3.4. In much of the detail, this system is very similar to Scope 3.2, except for the way jobs are allocated to control points. Under Scope 3.4, a job can be pre-empted from a control point before it has finished, if it has to wait for some action, such as a response from the operator, access to a file held by another job, or (and this is the main reason for the change) response from an interactive terminal. A sub-system of Scope 3.4 called the scheduler, comprising both CPU and PPU programs, controls the allocation and pre-emption of jobs, according to a complex priority scheme. The priority scheme divides jobs into several classes, and enables the installation to define various priority levels, aging rates, etc. for each class.

There is a wealth of questions about Scope 3.4 that could be investigated by simulation. The optimization of the priority parameters for a given job mix ("tuning the system") is very difficult, and simulation would greatly assist in doing it. The addition of interactive jobs also opens many avenues for investigation.

It is uncertain how difficult it would be to convert Simulator B to Scope 3.4. Such a project would certainly require more information about the 844 disc than is currently available, and the operation of the scheduler programs would have to be

investigated. Some consideration would have to be given to the simulation of interactive terminals, which now supply a significant part of the system's work. This latter consideration would interfere with the deterministic character of Simulator B. The processes of most interest in Scope 3.4 (scheduling and priority assignment) could probably be investigated with a relatively coare simulation about the level of Simulator C cited above. Such a project could incorporate a great deal of the information contained in the present Simulator B.

10.2.4 Analytic Models

Another application envisaged for Simulator B is as a standard against which analytic models can be validated. Some of the queueing processes in the Scope 3.2 system are subject to unusual rules and restrictions, and analytical simulation of these queues would be an interesting project. Because Simulator B works at a relatively detailed level, it should provide as good a standard for comparison as the real system would. It has the added advantage that processes are more easily observed in the simulated system, so that disagreements observed in such validations could be easily identified and eliminated.

10.2.5 Work in Other Areas

A problem that was encountered continually in the development and description of the simulator was the lack of a general, widely-known, formal representation for the sorts of entities being studied.

For example, the essence of a single process can be represented clearly by a flowchart. To adequately represent the interaction of two (or several) processes communicating with one another, some more sophisticated notation is required. Some genral framework for describing such entities, into which could be fitted the processes of a particular operating system (or many other sorts of system, for that matter) would be very useful. The networks of 5.2.4 are a move towards this ideal, but they do not really provide a clear indication of the processes, interactions and mechanisms at work.

10.3 SUMMARY

The development of a simulation model of the 6400 under Scope 3.2 provided an opportunity for exploring the processes involved in constructing and validating large computer simulations. One of its more important achievements was to show that starting simulation at a very detailed level, while slow and tedious, provides a more foolproof way to accurate simulation than the more conventional method of adding refinements to a coarse model. This approach also had considerable bearing on the methods used for validation. These methods enabled an exhaustive deterministic validation to be carried out on almost the whole simulated system. The close agreement that was obtained from the validation indicates how accurate a simulation can be attained using these techniques.

Work on the simulator opened up a number of avenues for further investigation, largely concerned with the characteristics of operating systems, and the way in which the mechanisms in an operating system interact. One of the problems about such studies is the lack of a scheme or language for fully describing interacting processes.

BIBLIOGRAPHY

1. BEAUMONT, W. P. and MACASKILL, J. L. C.

Studies in the Simulaton of Computers

Australian Computer Journal, Vol. 7, No. 1 pp.7-11 (1975)

2. BELL, T. E.

Objectives and Problems in Simulating Computers

AFIPS Fall Joint Computer Conference, Vol. 41, No. 1, pp. 287-297 (1972)

3. BIRTWHISTLE, G., DAHL, O.-J., MYHRHAUG, B. and NYGAARD, K.

Simula Begin

Studentlitteratur (1973)

4. BLATNY, J., CLARK, S. R. and ROURKE, T. A.

On the Optimization of Performance of Time-Sharing Systems by Simulation

Communications of the ACM, Vol. 15, No. 6, pp. 411-420 (1972)

5. BLUNDEN, G. P. and KRASNOW, H. S.

The Process Concept as a Basis for Simulation Modelling

Simulation, Vol. 9, No. 2, pp. 89-93 (1967)

6. BOOTE, W. P., CLARK, S. R. and ROURKE, T. A.

Simulation of a Paging Computer System

The Computer Journal, Vol. 15, No. 1, pp. 51-57 (1972)

7. BUCHOLZ, W.

A Synthetic Job for Measuring System Performance

IBM Systems Journal, Vol. 8, No. 4, pp. 309-318
(1969)

8. CALINGAERT, P.

System Performance Evaluation : Survey and Appraisal

Communications of the ACM, Vol. 10, No. 1, pp. 12-18
(1967)

9. CLARK, S. R., ROURKE, T. A. and WREN, J. M.

A Note on the Reproducibility of Discrete Event Simulation Studies

INFOR, Vol. 10, No. 2, pp. 194-200
(1972)

10. CLAYTON, B. B., DORFF, E. K. and FAGEN, R. E.

An Operating System and Programming Systems for the 6600

AFIPS Spring Joint Computer Conference, Vol. 25, pp. 41-57
(1964)

11. COHEN, C.

Simulation Modelling and Programming with SPURT/73

Vogelback Computing Center, Northwestern University
(1973)

12. CONTROL DATA CORPORATION

6000 Series Computer Systems : Peripheral Equipment Reference Manual

CDC Publication No. 60156100, Revision H
(1970)

13. CONTROL DATA CORPORATION

6000 Series Computer Systems : Reference Manual

CDC Publication No. 600100000, Revision M
(1971)

14. CONTROL DATA CORPORATION

6000 Series Computer Systems : Scope 3.2 Reference Manual

CDC Publication No. 60189400, Revision K
(1970)

15. CONTROL DATA CORPORATION

Scope System Programmers' Guide

Control Data Corporation
(1970)

16. COOKE, M. P.

Some Techniques for the Measurement of Computer Performance

Australian Computer Conference, Vol. 5, pp. 579-587
(1972)

17. DAHL, O.-J. and NYGAARD, K.

Simula - an Algol-based Simulation Language

Communications of the ACM, Vol. 9, No. 9, pp. 671-678
(1966)

18. DAY, F. A. G. and WATMOUGH, R.

A Simulation Study of WRE 7090/1401 Operations

Weapons Research Establishment, WRE Report 429(AP)
(1971)

19. DENISTON, W. R.

SIPE, a TSS/360 Software Measuring Technique

Proc. ACM National Conference, Vol. 24, pp. 229-239
(1969)

20. DENNING, P. J.

The Working Set Model for Program Behaviour

Communications of the ACM, Vol. 11, No. 5, pp. 323-333
(1968)

21. DEWAN, P. B., DONAGHEY, C. E. and WYATT, J. B.

A Generalized Simulation Model for Computer Systems

Proc. Australian Computer Conference, Vol. 5, pp. 543-551
(1972)

22. DUNN, P. A.

Procsim : a Processor Simulation Language

Proc. Australian Computer Conference, Vol. 6, pp. 553-563
(1974)

23. FIFE, D. W.

An Optimization Model for Time-Sharing

AFIPS Spring Joint Computer Conference, Vol. 28,
pp. 97-104
(1966)

24. FISHMAN, G. S. and KIVIAT, P. J.

The Statistics of Discrete Event Simulation

Simulation, Vol. 10, No. 4, pp. 185-195
(1968)

25. FOX, D. and KESSLER, J. L.

Experiments in Software Modelling

AFIPS Fall Joint Computer Conference, Vol. 31, pp. 429-436
(1967)

26. HAMLET, R. G.

Efficient Multiprogramming Resource Allocation and Accounting

Communications of the ACM, Vol. 16, No. 6, pp. 337-343
(1973)

27. HART, L. E.

The User's Guide to Evaluation Products

Datamation, Vol. 16, No. 17, pp. 32-35
(1970)

28. HARVEY, D. J. and NASH, B. L.

Tuning a Time-Sharing System

Proc. Australian Computer Conference, Vol. 6, pp. 160-171
(1974)

29. HERMAN, D. J. and IHRER, F. C.

The Use of a Computer to Evaluate Computers

AFIPS Spring Joint Computer Conference, Vol. 25, pp. 383-395
(1964)

30. HOBGOOD, W. S.

Evaluation of an Interactive-Batch System Network

IBM Systems Journal, Vol. 11, No. 1, pp. 2-15
(1972)

31. HUESMAN, L. R. and GOLDBERG, R. P.

Evaluating Computer Systems through Simulation

The Computer Journal, Vol. 10, No. 2, pp. 150-156
(1967)

32. HUTCHINSON, G. K.

A Computer Center Simulation Project

Communications of the ACM, Vol. 8, No. 9, pp. 559-568
(1965)

33. HUTCHINSON, G. K. and MAGUIRE, J. N.

Computer Systems Analysis and Design through Simulation

AFIPS Fall Joint Computer Conference, Vol. 27,
pp. 161-167
(1965)

34. JOSLIN, E. O.

Applications Benchmarks : the Key to Meaningful Computer Evaluations

Proc. ACM National Conference, Vol. 20, pp. 27-37
(1965)

35. KATZ, J. H.

Simulation of a Multiprocessor Computer System

AFIPS Sprint Joint Computer Conference, Vol. 28,
pp. 127-139
(1966)

36. KATZ, J. H.

An Experimental Model of System/360

Communications of the ACM, Vol. 10, No. 11, pp. 694-702
(1967)

37. KIVIAT, P. J.

GASP - A General Activity Simulation Program

Project No. 90.17.019(2), Applied Research Laboratories, U.S. Steel
(1963)

38. KOLENCE, K. W.

A Software View of Measurement Tools

Datamation, Vol. 17, No. 1, pp. 32-38
(1971)

39. LEHMAN, M. M., ESHED, R. and NELLER, Z.

The Checking of Computer Logic by Simulation on a Computer

The Computer Journal, Vol. 6, No. 2, pp. 154-162
(1963)

40. LEHMAN, M. M. and ROSENFELD, J. L.

Performance of a Simulated Multiprogramming System

AFIPS Fall Joint Computer Conference, Vol. 33, No. 2, pp. 1431-1442
(1968)

41. LUCAS, H. C.

Performance Evaluation and Monitoring

Computing Surveys, Vol. 3, No. 3, pp. 79-91
(1971)

42. MACDOUGALL, M. H.

Simulation of an ECS-Based Operating System

AFIPS Spring Joint Computer Conference, Vol. 30, pp. 735-741
(1967)

43. MACDOUGALL, M. H.

An Introduction to the Simulation of Computers

Computing Surveys, Vol. 2, No. 3, pp. 191-209
(1970)

44. McKINLEY, H.

Virtual Machine Modelling and Performance Estimation

Proc. Australian Computer Conference, Vol. 5, pp. 160-164
(1972)

45. MARKOWITZ, H. M., HAUSNER, B. and KARR, H. W.

Simscript - A Simulation Language

Prentice-Hall
(1963)

46. NAYLOR, T. H., WERTZ, K. and WONNACOTT, T. H.

Methods for Analysing Data from Computer Simulation Experiments

Communications of the ACM, Vol. 10, No. 11, pp. 703-710
(1967)

47. NAYLOR, T. H. and FINGER, J. M.

Verification of Computer Simulation Models

Management Science, Vol. 14, No. 2, pp. B92-B101
(1967)

48. NEMETH, A. G. and ROVNER, P. D.

User Program Measurement in a Time-Shared Environment

Communications of the ACM, Vol. 14, No. 10, pp. 661-666
(1971)

49. NIELSEN, N. R.

An Approach to the Simulation of a Time-Sharing System

AFIPS Fall Joint Computer Conference, Vol. 31, pp. 419-428 (1967)

50. NIELSEN, N. R.

Computer Simulation of Computer System Performance

Proc. ACM National Conference, Vol. 22, pp. 581-590 (1967)

51. NIELSEN, N. R.

The Simulation of Time-Sharing Systems

Communications of the ACM, Vol. 10, No. 7, pp. 397-412 (1967)

52. NIELSEN, N. R.

ECSS, An Extendable Computer System Simulator

Proc. 3rd ACM Conference on the Applications of Simulation, pp. 114-129 (1969)

53. NIELSEN, N. R.

An Analysis of some Time-Sharing Techniques

Communications of the ACM, Vol. 14, No. 2, pp. 79-90 (1971)

54. NIELSEN, N. R.

ECSS and the Problems of Constructing Computer System Simulations

Proc. Australian Computer Conference, Vol. 5, pp. 558-565 (1972)

55. NOE, J. D. and NUTT, G. J.

Validation of a Trace-Driven CDC 6400 Simulation

AFIPS Spring Joint Computer Conference, Vol. 40, pp. 749-757
(1972)

56. NOE, J. D. and NUTT, G. J.

Macro E-Nets for Representation of Parallel Systems

IEEE Transactions on Computers, Vol. 22, No. 8, pp. 718-727
(1973)

57. ROSIN, R. F.

Determining a Computing Centre Environment

Communications of the ACM, Vol. 8, No. 7, pp. 463-468
(1965)

58. SCHULMAN, F. D.

Hardware Measurement Device for IB System/360 Time-Sharing Evaluation

Proc. ACM National Conference, Vol. 22, pp. 103-109
(1967)

59. SEAMAN, P. H. and SOUCY, R. C.

Simulating Operating Systems

IBM Systems Journal, Vol. 8, No. 4, pp. 264-279
(1969)

60. SHEDLER, G. S.

A Queuing Model of a Multiprogramming Computer with a Two-Level Storage System

Communications of the ACM, Vol. 16, No. 1, pp. 3-10
(1973)

61. SMITH, J. C.

An Analysis of Time-Sharing Computer Systems Using Markov Models

AFIPS Spring Joint Computer Conference, Vol. 28, pp. 87-95 (1966)

62. SMITH, W. E.

A Digital System Simulator

IRE Western Joint Computer Conference, pp. 31-36 (1957)

63. STANG, H. and SOUTHGATE, P.

Performance Evaluation of Third Generation Computing Systems

Datamation, Vol. 15, No. 11, pp. 181-190 (1969)

64. STATLAND, N.

Methods for Evaluating Computer Systems Performance

Computers and Automation, Vol. 13, No. 2, pp. 18-23 (1964)

65. STONE, H. S. and FULLER, S. H.

On the Near-Optimality of the Shortest-Latency-Time-First Drum Scheduling Technique

Communications of the ACM, Vol. 16, No. 6, pp. 352-353 (1973)

66. TEORY, T. J. and PINKERTON, T. B.

A Comparative Analysis of Disc Scheduling Policies

Communications of the ACM, Vol. 15, No. 3, pp. 177-184 (1972)

67. THORNTON, J. E.

Design of a Computer : the Control Data 6600

Scott, Foresman
(1970)

68. WARNER, C. D.

Monitoring : a Key to Cost Efficiency

Datamation, Vol. 17, No. 1, pp. 40-49
(1971)

69. WIGAN, M. R.

The Fitting, Calibration and Validation of Simulation Models

Simulation, Vol. 18, No. 5, pp. 188-192
(1972)

70. YOUCHAH, M. I., RUDIE, D. D. and JOHNSON, E. J.

The Data Processing System Simulator

AFIPS Fall Joint Computer Conference, Vol. 26, No. 1, pp. 251-276
(1964)