STUDIES OF SOME FEEDBACK CONTROL MECHANISMS IN OPERATING SYSTEMS

A. ALDERSON

Ph.D. Thesis August 1974

University of Newcastle Upon Tyne

ERRATA

p 84 1 4 should read

... sets a level at which ...

p 97

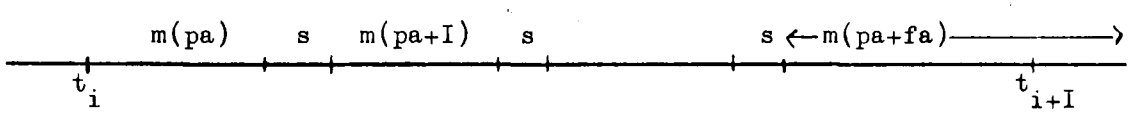
The reader's attention is drawn to the discontinuous behaviour of the term given for calculating the reference statistic when F(t,t+St) is zero. It is possible that this discontinuity could cause the behaviour shown in figure 4.6.

p 100 - p 101 should read

Then during the interval (t_i, t_{i+I}) process a will cause fa page faults, where

$$\sum_{i=pa}^{pa+fa+I} (m(i)+S) \leq St < \sum_{i=pa}^{pa+fa} (m(i)+S)$$

To see this see the diagram below



p 101 should read

At the end of the interval marked as m(pa)+S in the diagram ...

p 101 should read

Thus the number of page faults caused by process b is given by summation of the page faults caused in each of the intervals m(pa)+S, m(pa+I)+S,....., m(pa+fa-I)+S. The final interval must be treated separately being of length

$$St - \sum_{i=pa}^{pa+fa-I} (m(i)+S)$$

Abstract

The possibility of enhancing the effectiveness of an operating system by the introduction of appropriate feedback controls is explored by examining some resource allocation problems. The allocation of core, CPU and I/O processors in a multiprogramming demand paging environment is studied in terms of feedback control.

A major part of this study is devoted to the application of feedback control concepts to core allocation to prevent thrashing and develop algorithms of practical value. To aid this study a simulator is developed which uses probability distributions to represent program behaviour. Successful algorithms are developed employing a two stage page replacement function which selects a process from which a page is then chosen to be replaced. Improving the performance of these algorithms by using a 'drain process' to aid the dynamic determination of the current locality of a process is also discussed.

The complexity of the overall resource allocation problem is dealt with by employing a hierarchy of individual resource allocation policies. These control scheduling, core allocation and dispatching.

By considering the levels of the hierarchy as separate feedback control systems the restrictions which must be placed upon the individual levels are derived. The extension of these results to further levels is also discussed.

ACKNOWLEDGEMENTS

The work described in this thesis has benefited greatly from many discussions with my colleagues. In particular I would like to thank Prof. B. Randell for his constant help and guidance, and Prof. W.C. Lynch of Case Western Reserve University, Ohio, Prof. J.J. Horning of the University of Toronto, and Dr. H.C. Lauer who contributed a great number of helpful criticisms and suggestions.

I would also like to thank Mrs. B. Bennett for her accurate and speedy typing of this thesis, and my wife for her encouragement and her correction of the text.

This research was supported by the Science Research Council.

CONTENTS

1 INTRODUCTION

2 FEEDBACK CONTROL SYSTEMS AND THEIR APPLICATION TO OPERATING SYSTEMS.

3 PROCESS BEHAVIOUR, THRASHING AND A CORE CONTENTION SIMULATOR

4 PREVIOUS APPLICATIONS OF FEEDBACK TO CORE ALLOCATION

5 CORE ALLOCATION USING FEEDBACK CONTROL

6 SCHEDULING AND DISPATCHING

TABLE OF FIGURES

CHAPTER 1

Introduction

1.1 Feedback in Operating Systems

Feedback effects manifest themselves in various aspects of operating systems. Decisions at computer management level and the detailed strategies of the operating system may equally well cause or be affected by feedback effects. A number of examples have been described in the literature. For example, Lynch (1967) discussed the effects of introducing fast response output devices upon the volume of printed output, and the tendency of queue sizes to stabilise in certain situations. Coffman and Kleinrock (1968) and Lynch (1972) described the response of users to changes in resource allocation policies. Wilkes (1971) analysed an algorithm designed to control the number of users logged into a terminal system, and Bunt and Hume (1971) examined self-regulating operating systems.

The frequency with which such effects occur and the magnitude of the disturbances they can cause points to a need for a study of such phenomena in operating systems. Further, one is prompted to enquire into the possibility of enhancing the working of an operating system by controlling and exploiting the existing feedback effects and by introducing further appropriate feedback controls.

1.2 The Area of Study

In view of the wide range of aspects of operating systems in which feedback occurs or is applicable it has been necessary to restrict the scope of this investigation. The allocation of core, central processor time and input/output processors are problems crucial to the effectiveness of an operating system and are topics of general interest. Furthermore various attempts have been made to apply feedback control to this area. This has been done both explicitly, (Shils, 1968; Wulf, 1969), or implicitly, as a consequence of design (Denning, 1968b; Wharton, 1971). These attempts provide a basis for further study. We shall therefore confine our studies to these resource allocation problems. However, it is hoped that the insight gained by studies in this restricted field will be of value in other aspects of resource allocation.

The problems of resource allocation arise principally as a consequence of multiprogramming. This is because of the concurrent use of system resources by two or more processes which are in partial states of completion. The problem can arise in uniprogramming as in the case of a process being too large to reside entirely in core so requiring overlay techniques to be employed. However, the problems are more diverse, complex and of wider interest in the case of multiprogramming operating systems and solutions should be more widely applicable. It is just such systems with which we shall concern ourselves.

1.2.1 Core Allocation

The form of addressing scheme used will be of basic importance in the design of the memory allocation algorithm of an operating system. There are four major possibilities for the addressing scheme from the allocation point of view and these form two classes. In the first the whole program and data of a process must be moved in and out of core, in which case relocation may or may not be possible. In the second the program and data may be divided into many pieces which may be of equal size (paging) or of various sizes (segmentation). These pages or segments are used as the units of allocation of core.

Segments provide a logical division of the memory space based upon the form of the process using that space. The Burroughs B5000 and B6500 systems provide a direct implementation of a segmented memory. Paging provides an arbitrary division of the memory space with the aim of easing core allocation problems.

We will consider segmentation only where it is the basis of core allocation. When we subsequently refer to segments we mean those as in the B5000 and B6500 implementations and not those of the IBM 360/67 or the GE645 where the segments are paged.

Demand-type dynamic storage allocation, where pages or segments not currently in core are fetched in response to an attempt to access them, are particularly vulnerable to loss of effectiveness due to mismanagement of core. (Denning, 1968a; Brawn and Gustavson, 1968). One would hope that by suitable allocation schemes appreciable improvement would be obtained. Furthermore paged and segmented addressing organisations seem to offer opportunities for more varied feedback control schemes since allocation decisions can be made in terms of smaller units than is possible with other addressing organisations.

Paging is simpler to consider and we shall concentrate on demand paging. However it is our belief that much we have learned carries over more or less directly to segmentation. We will therefore talk in terms of demand paging but we also have in mind such systems as the B5000.

1.2.2 CPU and Input/Output Processor Allocation

The allocation schemes for CPU and I/O processors are not dependent upon the addressing hardware. Therefore the methods we develop should be generally applicable. However, the work of Denning (1968b) and Randell and Kuehner (1968) points out the necessity of developing an integrated strategy for resource allocation. Due to the intimate relationship between the progress of the computations a process performs and the pages of program and data which must be in core, memory management and the allocation of other resources should be closely related activities. They should not be treated independently, rather a decision to allocate some resource should be made with regard to allocation decisions made for other resources. Therefore, there should be an interdependence of the various allocation strategies.

1.3 Imposing a Structure

In order to deal with the complexity of the resource allocation problem it would be helpful if we could define some framework within which allocation of an individual resource can be made without explicit reference to the allocation of other resources. The necessary restrictions which should be placed upon such allocations due to the overall interdependencies must then occur as a consequence of the structuring.

A promising structure from the point of view of formulating and understanding such interrelations is a hierarchy. We have chosen to structure our decision process as a hierarchy of individual resource allocation policies. Furthermore the interest which has been generated in such structuring by the work on program correctness, particularly its use in the design of the THE operating system by Dijkstra (1968), enhances the potential value of a study involving a hierarchy.

Our hierarchy has three distinct levels with which we shall concern ourselves. We shall refer to these as scheduling, core allocation and dispatching. The function of the scheduler is to define a priority ordering upon the various processes which require system resources. The priority order may reflect management policies to favour certain kinds of processes

and to provide certain levels of turnaround to batch users and response to terminal users. It may also incorporate deadlock avoidance strategies. This is the most general level. We shall refer to it as the highest, all others being subordinate. It is not concerned with the allocation of resources but with specifying which processes should be allocated resources.

The second level is core allocation. The allocation is performed on the basis of the priority ordering placed upon the processes by the scheduler.

The lowest level of our hierarchy is dispatching. The dispatcher is responsible for the allocation of the CPU and I/O processors to those processes which have been allocated core by the core allocation strategy. The dispatcher will not necessarily use or be constrained by the priority order imposed by the scheduler. It may if desired apply its own priorities to the set of processes allocated core by the core allocation algorithm.

1.4 Aims of the Thesis

A hierarchy, and our hierarchy in particular, will not necessarily be a satisfactory structuring for resource allocation, and we will be concerned with examining its suitability. It is our intention to consider our hierarchy in terms of feedback concepts showing how feedback control might be applied to govern or improve existing feedback effects which occur in resource allocation. We shall discuss core allocation first since this is the algorithm of prime importance in a demand paging operating system.

A major part of the thesis will be devoted to the study of core allocation and we will try to develop algorithms of practical value. To this end we will develop a simulator whose parameters are drawn from the literature and which are representative of systems such as the Michigan Terminal System. However, our analyses of these algorithms will be concerned with their qualitative, rather than quantitative, performance. We shall be particularly concerned with their dynamic behaviour under overload conditions.

We shall then examine in terms of feedback concepts the relationships between the levels of our hierarchy. The interactions of the scheduler and core allocation, and core allocation and the dispatcher will be studied in an attempt to derive conditions which are necessary in such a structure if it is to be successful.

1.5 Structure of the Thesis

In Chapter 2 various well-established notions of feedback control systems are introduced and their importance explained by relating them to topics in operating systems. In particular, the elements of a feedback control system and the important concepts of stability and settling time are described. A classification of such systems is introduced. The analysis of feedback control and the difficulties of applying the analyses to operating systems is discussed. An example in which some measure of success has been achieved is reviewed.

In Chapter 3 we introduce the concepts we require to discuss core allocation. The phenomenon of thrashing is discussed and a structuring of core allocation algorithms to ease design problems is introduced. The simulation model which we use in our investigations is then described.

Chapter 4 reviews applications of feedback to core allocation preceding our main study. Four algorithms are discussed and analysed both in feedback terms and by use of the simulator.

Taking the lessons of Chapter 4 as a basis we develop a number of core allocation algorithms in Chapter 5. The

main themes of this chapter are the control of multiprogramming level and the estimation of the core required by each process. The concept of a 'drain' process is introduced as an aid to the estimation of the core required by a process.

In Chapter 6 we discuss the interactions between the levels of our hierarchy of allocation strategies. We analyse the interactions of the scheduling and dispatching strategies with the core allocation policy and derive the conditions necessary for effective operation to be maintained.

The application of these results to other areas of resource allocation in operating systems is discussed and concluding remarks are presented in Chapter 7.

CHAPTER 2

Feedback Control Systems and their Application to Operating Systems

2.1 Introduction

The purpose of this chapter is to relate the concepts of feedback control (Goode and Machol, 1957; Grabbe et al, 1958) to operating systems and to define the terms used in later chapters. Examples from operating systems will be used to illustrate the application of the concepts to this field. We will then discuss a classification of feedback control systems and examine the general properties of each class. Finally the analysis of feedback control systems in operating systems will be discussed.

2.2 Types of Control Systems

2.2.1 Open-Loop Control

The simplest type of control system is the open-loop control. Parameters of the control are set and each specific setting of the parameters determines a fixed level of performance for the controlled system.

A particular example is the original APL\360 system (Breed and Lathwell, 1968), the resource allocation of which is greatly simplified by a number of features. Each user has an input/output device of his own, a teletype or communications terminal, and is unable to command the use of other input or output devices. All users are constrained to work within a fixed amount of core, their workspace, which is the same size for all users. Simplification also results from all users programming in the same language, the interpreter of which is wholly resident in core. This enables very simple resource allocation strategies to be employed.

The number of workspaces which may be resident in core and the maximum CPU time any user may obtain before some other user receives service are given set values. These settings were experimented with until a satisfactory level of operating was observed and then fixed. The control is of the open-loop form with the level of

performance being determined by the setting of the parameters.

In order to achieve precise control with an open-loop system it is necessary to have accurate knowledge of the relationships between a number of variables. Furthermore, an open-loop system cannot deal with disturbing factors other than those specifically included in its design. Experimenting with the settings as in the APL system is one way of obtaining knowledge of the relationships. However, the precision of the knowledge will be determined by the number of settings it is viable to try. In fact the APL system experiences fluctuating performance due to disturbances caused by variation in the workload not catered for in the control design. The overall result however is satisfactory.

2.2.2 Closed - Loop Control

This type of control is also called feedback control.

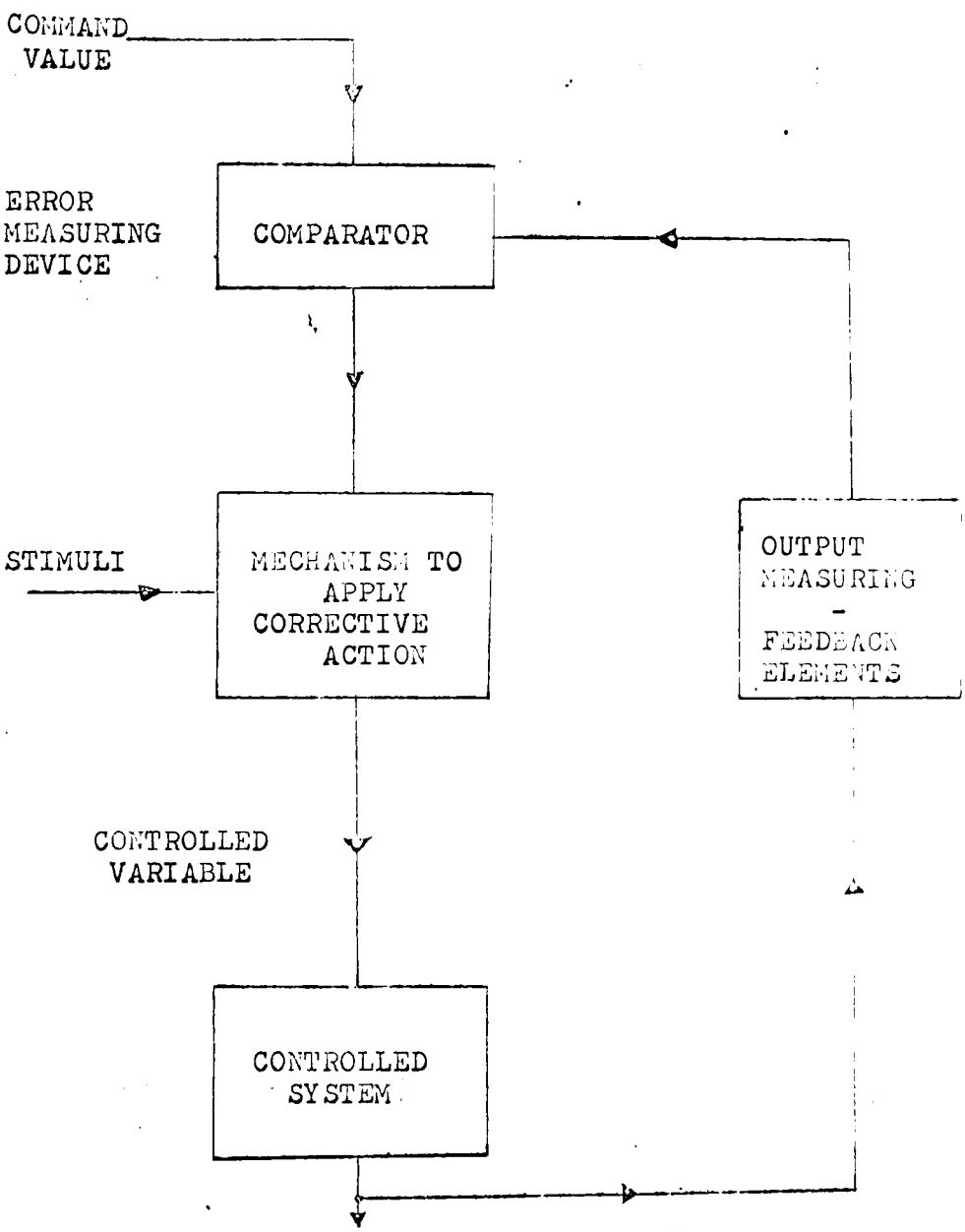
A closed-loop system can in general deal with all disturbing factors without an accurate knowledge of the relationships between the values of the various factors and the control action to be taken. The factor which is causing the disturbance need not be known. It is enough to monitor variations in the performance of the system and take action known to cause opposing variations.

In general purpose operating systems the factors which influence the usage of any particular resource are extremely complex and subject to many disturbances. In such operating systems the use of feedback control systems in resource allocation seems to be a natural choice. This is not to say that open-loop control is to be discarded. Such control has been successfully used in situations in which the relationships are simplified for some reason. However, feedback has a wider application in this field.

2.3 The Elements of a Feedback Control System

A feedback control system is a control system which tends to maintain a prescribed relationship of one system variable to another. This is achieved by comparing functions of the variables and using the result to activate a corrective mechanism as required. A feedback controller is supplied with an ideal value for the variable to be controlled. This ideal is called the command value. The actual speed of a car is an example of a controlled variable whose command value is the desired speed selected by the driver. The controlled variable and its command value, which may also vary, are the system variables whose relationship is of interest. As a result of a comparison of the command value and the measured value of the controlled variable the feedback controller manipulates the controlled system in order to maintain the required relationship. In the car the driver varies his pressure on the accelerator so as to match the observed speed to the desired speed, the prescribed relationship being equality of these values.

This kind of control is used in those situations in which the system experiences stimuli affecting the value of the controlled variable. Stimuli may take the form of a change of the command value or a disturbance of the controlled system due to some external agency. Both the



A typical feedback control system

Figure 2.1

actual and desired speeds of a car will vary according to road conditions which may be thought of as external stimuli.

A typical feedback control system, figure 2.1, includes an output measuring device feeding back to an error measuring device which activates a mechanism to apply corrective action.

2.4 Properties of a Feedback Control System

2.4.1 Stability

The stability of a feedback control system is a consideration of prime importance. There is in general some discrepancy between the value of the controlled variable and its command value. This is because of errors in measuring or because of delay between measuring and comparing values. The introduction of the feedback loop creates the possibility that stimuli may occur at times relative to the lagging feedback such that the two are out of phase. Measuring error may also cause the relationship to be incorrectly evaluated. The supposed corrective action may therefore augment the original error so that it becomes even larger. Thus the possibility of instability can be introduced.

A system is stable if the response to an input always reaches and maintains some useful value within a reasonable period of time. An unstable system will on the other hand produce persistent oscillations of the response and may even drive it to some excessive value.

Ideally a system would maintain zero error despite disturbances, respond instantly to any change of command

value and be entirely stable. In practice compromises are necessary. Increasing the accuracy of a system may be accomplished by making the controller more sensitive so that it provides the same increment of correction for smaller increments of error. However, stability will be adversely affected if the controller is made very sensitive. A corrective action may be initiated which is large enough to cause a response resulting in a greater error than that initiating the correction. This may continue until limited by the physical properties of the system. Thus a control system must be a compromise between stability and low error with fast response to stimuli.

The systems whose stability is of particular interest to us are those which allocate core in multiprogramming operating systems. Consider a demand paging system in which there is no attempt to limit the multiprogramming level. Typically there will be an instant at which all of the processes currently sharing core will be waiting for I/O, either for a missing page or a data transfer. The reaction of the dispatcher to the system becoming idle will be to introduce a further process to utilise the CPU. This however increases core contention so increasing the probability that at some time all processes will again be waiting for I/O. This leads to the introduction of further processes. We note that this

will be a persistent rather than a transitory effect. Thus when there is sufficient core to satisfy the requirements of all processes, the system is stable. However, when this condition is not met the system is unstable.

The phenomenon just described is that of 'thrashing'. This is a common cause of degraded performance in demand paging systems. We will discuss thrashing in detail later.

In a case in which instability can occur it may be simpler not to redesign the primary level of control but to add a further level which in effect reduces the physical limits of the system. This constrains the effects of the instability. These extra controls have been termed safety devices. A fixed limit upon the multiprogramming level of an operating system might be such a safety device. Safety devices are inefficient in that they permit a persistent deviation from the desired operating level. However, they may allow achievement of a level of operation which is on average greater than could have been obtained with the primary control alone.

2.4.2 Response and Settling Time

The speed of response of a control system to a stimulus is also of fundamental importance since the control system must complete its reaction to a stimulus within a reasonable time. If this settling time is greater than the interval between stimuli then the system may never catch up. The demands will not be met even if the required accuracy and stability are attainable.

2.4.3 The Operating Region

A control system may be stable under one set of operating conditions whilst it is unstable for others. The set of operating conditions for which the system is stable is called the operating region. Having designed a control system with as large an operating region as possible there are two ways of preventing degradation of the controlled system when the conditions prevailing are outside the operating region. The algorithms which represent the controls may be altered to ones more suited to the new conditions. Alternatively, the conditions themselves may be altered to maintain them within the operating region of the control system.

These techniques are discussed by Bunt and Hume (1971) and they describe a number of operating system strategies, such as those of Rolfson and Kleinrock, which employ the techniques. Rolfson (1968) proposed a strategy to regulate the load presented to a simple process-at-a-time environment employing roll-in/roll-out, in such a way as to give fast turnaround to those processes requiring the least execution time. This was achieved by varying a limit used to classify such short processes, the limit being decreased as system load increased.

Kleinrock (1970) proposed a parametric scheduling algorithm whose two parameters may be chosen in such a way as to bias the system against certain types of process. It is proposed that on entry to the system a process joins a waiting queue with zero priority. The priority is increased linearly with rate α until the process joins those being serviced when its priority is increased at rate β. By setting α and β appropriately the strategy may be made to discriminate against processes requiring extended execution. Bunt and Hume discuss the manner in which α and β may be varied dynamically so as to alter the operating region thereby adapting the system to its changing load.

A rather more ambitious scheme to regulate the load and maintain it within the operating region of a multi-programming system is described by Wulf (1969). This involves monitoring the resource usage characteristics of all processes and the utilisations of the resources. Individual resource utilisations in excess of preset values cause processes which use the resource heavily to be suspended. Specific processes may be reactivated to use underutilised resources. A decision tree is used to implement the control mechanism.

2.5 A Classification of Feedback Control Systems

Consideration of the ways in which feedback controls may be designed has led us to a classification which we have found useful, and which has also been put forward by Wilkes (1973). There are three basic design types. These may be described as inherent, superimposed and model driven feedback control systems.

2.5.1 Inherent Control

Inherent or implicit feedback is a property of the system which is being controlled. The system controls itself in that no explicit measurements are made of the controlled variables and the command values are set by implication and cannot be changed. An example of such a control system is seen in the tendancy of queue lengths to stabilise. (Section 2.7). Such behaviour is a property of the system itself, no explicit measurement or control being imposed. It is difficult to design stable controls of this form since the number of choices for control variables is often limited and so design possibilities may be constrained. Thus if such a control system should prove unstable the only choice open may be to design again completely.

2.5.2 Superimposed Control

Superimposed feedback control as the name suggests is separate from the system being controlled. The superimposed controller acts as a monitoring process which takes measurements of the variables to be controlled. On the basis of comparisons of these measured values with previously set command values, actions are taken which directly affect the controlled system. Subsequent measurements are taken which assess the success of the control actions and these generate further corrective actions.

It is a feature of such controllers that they are activated by actual errors, and they assess the success, or otherwise, of the control actions in terms of the errors those actions cause in the actual system. Obviously in control systems which rely upon errors to activate them, precise control at all times is not possible. A further property of superimposed controllers, which is important when considering stability, is that the information upon which the control is based introduces a time lag into the control. Essentially a control decision is taken and no further control is performed until the outcome of the control action produces an error.

A good example of such a system is the OS-3 operating system used at Oregon State University (Meeker et al, 1969). CPU time is allocated by providing service on a round-robin basis. Superimposed upon this simple base is a hierarchy of feedback control loops each designed to alleviate problems caused by its predecessor. At the first level is a control which cycles a 'high priority' pointer in a round-robin of the dispatcher queue. The core allocation algorithm is biased to give the 'high priority' process priviliged core usage. This pointer cycles at a rate proportional to real elapsed time and the high priority user's demand for core.

In an attempt to limit the number of active processes with high core demand, a further control is superimposed. This suspends processes whose core demands exceed a given value in the period taken by the high priority pointer to pass from one process to the next. Processes are suspended when page traffic is heavy as defined by a further monitoring loop.

Suspended processes may be reactivated provided page traffic is light, which is defined by yet another control loop. Each loop may also alter the parameters of each of the other control loops.

The OS-3 system provides an excellent example of superimposed feedback control. However, whether it is effective feedback control is difficult to say since analysis of a system of such complexity would be difficult indeed.

2.5.3 Model Driven Control

The third design type is the model driven control system. These control systems have found wide application in very complex situations such as the control of chemical manufacturing plants (Smith, 1970). As with the superimposed control systems the controller is separate from the system to be controlled and receives information concerning the controlled variables by monitoring the controlled system. The basis of the model driven systems is a model, either mathematical or simulation, of the system to be controlled. Proposed corrective actions are first applied to the model and the eventual control action taken is based upon the success predicted by the model. Contrast this with the superimposed systems in which control actions are in effect tested only upon the actual system.

The model driven feedback control system may be thought of as a synthesis of open-loop and closed-loop controls. The model forms the open-loop component, accepting parameters and by a deterministic process predicting the values of the control variables which will provide the required performance in the situation specified by the parameters. The inadequacies of the model are mitigated by the feedback of information provided by the closed-loop component.

These systems provide far greater scope for the elimination of instability due to the flexibility of the model which allows very detailed examination of a situation and the inclusion of special treatment of exceptional cases known to cause problems. However, the performance of the controlled system depends heavily upon the design of the model and accuracy of the information supplied to the model by the feedback component.

These controllers are a relatively recent development being associated with the introduction of computers into the control mechanism. The computer provides the speed and flexibility necessary to implement the control model.

Application of this technique in operating systems has so far been limited. Wulf (1969) utilised this form of control in a comprehensive resource allocation scheduling strategy in an operating system. It has also been used in those core allocation policies related to the Working Set Model of program behaviour. Denning (1968b) proposed a scheme using the Working Set Model and it has also been applied in the I.R.I.A. ESOPE system (Bétourné et al, 1970, 1971). Discussion of these schemes is deferred to Chapter 4.

2.6 Discrete Control

The use of computers to implement control of systems which are essentially continuous implies the need to incorporate analogue to digital conversion. This has led to the development of sampled control systems. Here the values of the controlled variables are sampled at equally spaced instants. The sampled data is then smoothed so as to simulate the continuous nature of the original variables. The smoothing operation implies the use of previous values and these provide the feedback. Techniques for handling sampled information are of course applicable to the problem of controlling a computer system. A notable example of such an application is the page replacement strategy employed in the MTS system on the IBM 360/67 at Newcastle-Upon-Tyne University until recently.

The strategy depends upon special hardware function which sets the 'reference bit' of a page whenever that page is accessed. The reference bits of all pages in core are examined at appropriate intervals and information concerning the usage of the pages is compiled in the form of reference statistics, one for each page. At these times the reference bits are reset. When a page replacement is required that page in core which has the lowest valued reference statistic is selected to be replaced.

A combination of discrete control and modelled control is well suited to a control system involving a computer. In particular such techniques can be employed in the design of an operating system for a computer. The work of Wulf (1969) is a striking example of their use. We will discuss the application of these and other techniques when we examine the resource allocation problems in detail.

2.7 Positive and Negative Feedback Effects

Two distinct naturally occurring forms of feedback effect may be distinguished. These are negative feedback, which tends to reduce the measured deviation of the control variable from the command value, and positive feedback which tends to amplify this deviation. We are interested in these naturally occurring effects since we may wish to induce them in the controlled system. Alternatively we may be able to use the feedback effects which already occur in the system to form part of the control itself.

An example of negative feedback occurring in an operating system is that concerned with the stabilising of queue sizes. Service rate may rise as the queue for service increases due to the economies of scale. This increase in queue size can be balanced by the reduction of the arrival rate due to suppression of the generating subsystem. This means that queue lengths are stabilised. (Lynch, 1967).

An example of positive feedback is the effect of improving turnaround time upon the output/compute ratio. As the turnaround time decreases the user tends to replace mass reporting techniques with sequential reporting, as has happened with the introduction of on-line terminals. A drop in the volume of output should reduce turnaround

further, and so we have a self-reinforcing effect. (Lynch, 1967). The magnitude of a positive feedback effect is eventually limited by physical properties of the system.

2.8 The Mathematical Analysis of Control Systems

Much effort has been made to produce mathematical analyses of control systems. Control systems belong to the domain of the engineer and for this reason the analyses are concerned with optimising the performance of the system under control. The closely related topic of maintaining stable operation has also received much attention. Of interest in this context is how the control system will react to various stimuli, studies of accuracy, stability and settling time being made when the command value is varied in some regular manner.

The problems to which the engineers have applied feedback control have been the control of systems which are essentially continuous in nature. Great success has been achieved in the control of motors and feedback is an essential element in a wide range of electronic equipment. This background has led mathematicians to search for general models of continuous feedback systems which are mathematically tractable.

A range of powerful techniques are now available to study such systems and many criteria exist for determining their stability. Typically such techniques require that the 'equations of motion' of the system be written down. An equation of motion represents the dynamic manner in

which a controlled variable is affected by the control actions. It is represented as a differential equation in the controlled variable. The command value is then represented by various functions, the two of most general interest being a step function, which is useful in determining settling time, and a sine function, which is used to study stability and accuracy.

The Laplace transform is an important tool in these analyses and by using its discrete analogue, the z-transform, much of the work can be carried over to systems in which the variables take on discrete values. These results can be applied to sampled systems.

Goode and Machol (1957) and Grabbe et al (1958) discuss the topics we have mentioned in some detail.

2.8.1 Difficulty of Application to Operating Systems

A common feature of the control systems analysed in the literature is that the equations of motion of the systems are expressable in a convenient mathematical form. The study of the natural phenomena typically controlled by such control systems is well founded and the basic relationships of the variables well established. Thus provided one is sometimes prepared to accept computational solutions rather than closed form solutions, the stability and settling time of the control can be examined. This, however, is seldom the case with operating systems.

Essentially, we must be able to express mathematically the way in which the error correcting mechanism of the control affects the value of the controlled variables. The most frequent control action used in operating systems involves altering a variable related to the controlled variable rather than the controlled variable itself. For example, contention for core is usually altered by varying the multiprogramming level of the operating system. Such a controlled variable as core contention here is said to be indirectly controlled.

Very little is known about the ways in which the various elements of an operating system interact and the relationships

which hold between the basic variables. We are certainly not able in general to write down a mathematically tractable function representing the changes caused in system variable by altering a related variable. Our general inability to produce the required equations of motion prevents us from taking advantage of the mathematical analyses available.

2.8.2 Analysis of an Automatic Load Adjustment Algorithm

One instance in which a feedback control applied to resource allocation was successfully analysed is described by Wilkes (1971). He deals with the adjustment of a number of console users on a time-sharing system. The control comprises two algorithms. The first makes a periodic prediction of the number of processes which could be allowed to enter the system without causing overloading. The second adjusts the number of users able to enter processes so that the average number of processes actually entered is as close as possible to the target number predicted by the first algorithm.

It is possible to write down a recurrence relation for the number of processes in the system at a given time in terms of the previous values of that quantity, the target number which is a weighted mean of previous values of the number of processes in the system, and the number of processes leaving the system. Initially Wilkes considered the case in which a decision to increase or decrease the number of users is put into effect without delay. It is possible to solve the resulting linear difference equation and study the behaviour of the roots of the equation. This allows one to examine the stability of the control.

Wilkes extends the analysis to cases in which the user is given a warning that he is to be logged out of the system and is given a period of grace to tidy up his work. The sampling time of the control was taken to be a multiple of the warning time. Again it is possible to derive the resulting linear difference equation and examine its roots. Wilkes was able to show that instability will be introduced whenever an attempt to estimate ahead over an interval greater than the interval used for calculating the running mean upon which the estimate of the number of processes entered into the system is based.

Wilkes was able to carry the analysis through because he was able to produce the equation of motion in each case. These equations could be derived because of the simple form of the control action which was to directly alter the controlled variable, this being the number of processes in the system. In such a situation one needs no knowledge of any special relationships between system variables for one to express the effects of the control action upon the controlled variable.

In general we will be unable to analyse the system we are controlling. Indeed it is in just such situations that closed-loop control is invaluable. It shields us from our ignorance of basic relationships, which if known could be used for open-loop control.

CHAPTER 3

Process Behaviour, Thrashing and a Core Contention Simulator

3.1 Relevant Topics Concerning Program Behaviour

Thrashing provides such an outstanding example of a feedback instability in operating systems that it is surprising it has not been studied from this point of view. In this chapter and the next two we will discuss the problem of thrashing and examine a number of core allocation algorithms which employ feedback control. As a basis for this discussion we will briefly review relevant topics concerning process behaviour.

A number of studies of process behaviour in a demand paging environment have been carried out. (E.g. Fine et al, 1966; Coffman and Varian, 1967; Brawn and Gustavson, 1968; Joseph, 1970). Each of these studies showed a distinct relationship between page fault rate and the amount of core storage that a process is constrained to use. The page fault rate remains at a reasonable level, increasing slowly as the number of pages of core available to the process is decreased, until a critical number of pages is reached. At this point the page fault rate rises very rapidly indeed. The critical number came to be known as the 'parachor' of the process (Randell and Kuehner, 1968).

This phenomenon can be explained in terms of 'locality of reference'. The process concentrates its references to its

address space in subsets of its total number of pages during intervals of time which may be significant fractions of its total running time. We refer to the subset of the address space which is being referenced during some interval as the 'locality of reference' of the process during that interval.

A process can run without any great hindrance from page faults in an amount of core smaller than the total address space, provided it has sufficient core for its current locality of reference. However, if the number of pages of core available to the process is less than the number of pages in a subset, some frequently referenced pages will need to be held on backing store and a higher page fault rate will be incurred.

We are assuming in this argument that there is an algorithm capable of maintaining in core the current locality of any individual process, when sufficient core has been allotted to that process. Such an algorithm does exist. It is the Least Recently Used (LRU) page deletion algorithm. (Belady, 1966).

The locality of reference may change gradually with the progress of the process. This may occur through the same section of the computation accessing different data pages or by the progression of the process through its various stages. Sudden changes of locality may occur due to

complete switches of action as may occur in the phase changes of a compiler.

Denning formalised the concept of locality of reference by his working set model of process behaviour. (Denning, 1968a). The working set of a process at time t is defined to be the set of pages referenced by the process during the interval (t-T,t), where T is a fixed period of time. The working set model is an attempt to produce a mathematical representation of the notion of locality and a number of interesting results have been derived from it. However, we have found that in examining core allocation algorithms we do not require to discuss locality of reference as formally as the working set model allows. We have found the concept of parachor easier to deal with and sufficient for our needs.

Somewhat confusingly parachor is but one of the several meanings which has been associated with the term working set. It is important to recognise that these meanings relate to distinct concepts. Denning's definition of working set implies observing the locality of reference during intervals of equal duration. The working sets are thus the observed localities during these intervals. The parachor is the number of pages of core required by the process in order for it to run without undue interuption by page faults. Thus it is a rather less precise quantity

than that of Denning. We observe that the parachor is in effect an average of the sizes of the localities of reference through which the process progresses, where each size is weighted by the number of page faults occurring while the program is in that locality of reference.

The term working set has also been associated with a style of organising core allocation. We shall discuss this in 4.3.

3.2 The Phenomenon of Thrashing

The high expectations of demand paging systems were rudely dashed by the appearance of the phenomenon of thrashing. Thrashing is recognised through its symptoms, namely poor CPU utilisation, high page traffic between core and backing store and degraded response. So deleterious is the effect of thrashing upon the efficiency of an operating system that the utility of paging systems has been seriously questioned. (E.g. Fine et al, 1966; Varian and Coffman, 1967).

Denning (1968a) traces the cause of thrashing to the relatively long time needed to transfer a required but absent page from backing store to core. Core may be required to fetch an absent page. It will be obtained by retiring a page which is resident in core onto backing store. However, the retired page may have been an active page of the process requesting the absent page or of some other process, so resulting in a possible chain reaction of page replacements. This chain reaction can occur more rapidly than a page can be transferred from backing store. Thus further active pages of a process can be removed from core before the absent page is available and so thrashing is observed.

We see then that the problem of allocating core so as to avoid thrashing is first one of ensuring that each process

which is allowed to obtain core has sufficient core to accommodate the pages in its current locality of reference. We must then ensure that these pages of core are occupied by the pages of the current localities of reference of the processes. The latter is not a difficult problem. In avoiding thrashing it is the manner in which the core is partitioned amongst the processes which is of importance.

Core may be shared amongst the processes competing for it in a number of ways. These may be characterised by what proportion of its current locality each process has resident in core. It has been demonstrated that a process requires a certain set of pages (which unfortunately is in general not predictable) to be in core in order that it may progress during some interval without undue interruption caused by the need to obtain further information from backing store.

The ideal partitioning of core is that in which all processes allowed to compete for core have all of their current localities in core. In this situation each process will progress at a favourable rate when it obtains the CPU. The CPU utilisation will be comprised of mainly user initiated activity with little overhead caused by page faults. In addition page traffic will be low. It should be noted that we wish to utilise core with the current

localities of as many processes as possible. The larger the number of processes available for dispatching the greater is the likelihood that there will be a process able to use the CPU when other processes are waiting for completion of I/O operations. CPU utilisation is thus improved.

The other general situation is that in which not all processes, and possibly none, have their current localities in core. Here if uncontrolled contention for system resources is allowed then thrashing is sure to occur. If processes compete on an equal basis then fair sharing will result in no process obtaining sufficient core.

It is possible to construct special cases of the over-partitioned state which are not unfavourable. Consider the case in which a priority ordering exists for usage of the CPU. Suppose the highest priority processes have their current localities in core and are sufficiently active to cause high utilisation of the CPU. The lower priority processes seldom obtain the CPU and so cause little paging. Thus although core may be said to be overpartitioned, thrashing might not be observed. However, if no further control is imposed this special situation is unstable because during their infrequent periods of CPU utilisation the lower priority processes will cause the higher priority processes to lose parts of their current localities from

core. This will in turn affect the CPU utilisation of the higher priority processes, allowing the lower ones more frequent periods of CPU usage. The core allocations of the higher priority processes will be eroded and thrashing will eventually occur.

Due to the unpredictability of current locality and our desire to optimise the underpartitioned state, our attempts at optimisation could cause overpartitioning to occur. In the overpartitioned state we require that some of the processes have the whole of their current localities in core and that the progress of these processes is not adversely affected by contention for core with the remaining processes. For this reason it is necessary that overpartioning is rigidly controlled. Thus we require to design control systems which will ensure that core is underpartitioned or is overpartitioned in a controlled manner.

3.3 Structuring the Core Allocation Algorithm

The existence of the kind of process behaviour we have described leads to the development of page replacement strategies which will 'learn' the locality of reference of a process and ensure that this set of pages remains core-resident. Belady (1966) concluded as a result of studies of single processes that the ideal algorithm should include some accumulation of data on the past references of the process. In other words that one should use some form of feedback in the core allocation policy.

The most basic attempts to apply feedback control to core allocation occur as applications of process behaviour studies which involved single processes. Belady analysed a number of strategies which use the setting of certain bits associated with each page of the process to provide feedback. The relative merits of the strategies were discussed and certain policies were shown to be very effective. However, the generalisation of these strategies applied globally over a set of processes have not had the hoped for success. Thrashing has been a common feature of operating systems employing such strategies.

The problem is that these globally applied algorithms take no account of the ownership of any page. Thus when

core is overpartitioned pages may be replaced indiscriminately from those processes which do and those which do not have their current localities in core. Essentially the amount of core which any process may obtain is dependent upon the memory demand characteristics of the other processes competing for core. Thus it is not possible to guarantee that any process will be able to retain its current locality in core. Such an algorithm must admit the possibility of thrashing which once it occurs is a persistent effect.

Allowing the amount of core storage which a process may obtain to be governed by the memory demands of other processes in an uncontrolled manner is therefore not a sound principle. The page fault rate of a process should be bounded by a function of its own demand for core and should not be unboundedly inflated by page replacements caused by other processes. Otherwise the possibility of thrashing occurs. This implies that the measure of core demand should depend only upon the properties of the process. Therefore there is a case for structuring the core allocation strategy so as to minimise the interaction between the core demands of individual processes.

In a demand paging operating system the basic allocation problem is the choice of which page to retire. If a page which does not belong to a current locality is removed

then a good choice has been made. The difficulty with page replacement is that the fact that a page is no longer needed is signalled by a non-event. After some point in time the page will cease to be referenced for a period which is long when compared to the times between references to pages being currently used by the process. In contrast a page fetch is triggered by a demand from the process.

The problem of deciding which pages of a process may be chosen for replacement has been widely studied. The work of Belady (1966) points to a number of suitable strategies. We could apply any of these strategies to a single process chosen by a control at a higher level of the core allocation algorithm. It is the problem of this level of the allocation algorithm to minimise the interaction of conflicting demands and ensure that core is partitioned in a controlled and effective manner.

The core allocation algorithm might therefore be structured as a two-level hierarchy. The principal algorithm selects on the basis of core partitioning a process from which a page is to be deleted. The subordinate algorithm selects a particular page for replacement from amongst the pages of the chosen process. Such a structuring allows the relatively well understood problem of page replacement to be disentangled from the problem of controlling core partitioning to prevent

thrashing. That is we separate the problems of allocating sufficient core for the current locality from those of ensuring that the pages of the current locality occupy the allocated pages.

In our studies we have tended to disregard the page replacement algorithm, assuming that a suitable candidate exists, and have concentrated upon developing the partitioning algorithm. We justify this by observing that Belady's studies show that the performance of good and bad replacement policies may differ by a factor of two. The performance of a system may well be degraded one thousand times by thrashing!

3.4 Difficulties of Designing Feedback Control Systems

The design of effective core allocation strategies which employ feedback control is not a simple matter. The synthesis of a feedback control system is an exercise for which there are few theoretical aids. Having completed a design adequate methods, both mathematical and simulation (if the problem proves mathematically intractable) are available to assess stability and effectiveness. In contrast, the synthesis is primarily a matter of engineering judgement. Although various criteria have been advanced (Grabbe et al, 1958; Goode and Machol, 1957), the complex and diverse factors which influence design are not amenable to simple mathematical representation.

Since judgement is a product of experience the course taken in this thesis was to design various core allocation algorithms, predict their performance and then use simulation to confirm or disprove our assertions. To this end a simulation model was constructed. This model is outlined sufficiently in the following sections to allow interpretation of the results produced.

3.5 Introduction to the System Simulator

3.5.1 Design Objectives of the Simulator

The primary requirement of the simulator was that it should be capable of displaying contention for core among a number of simulated processes under a variety of core allocation and scheduling policies. We further required that the level to which any element of the system was simulated could be varied. This was to allow detail to be added at any point at which it was found necessary or useful. This led to a highly modular design where the various modules representing the algorithms and hardware interfaced simply with the basic timing loop of the simulator.

The second major consideration was the need to minimise the execution time required since we wished to perform large numbers of simulations. The basic timing loop of the simulator is therefore event driven. It recognises two basic event types. These events are the 'internal interrupt', indicating that a process is requesting service (other than CPU), and the 'external interrupt' signalling completion of the requested service. End-of-time-slice is also implemented in the basic timing cycle.

To further minimise execution time it was decided to simulate the characteristics of the processes rather

than to employ the more realistic but more expensive traces of real process (e.g. Conti et al, 1968). The notion of generating process characteristics is extended to the simulation of paging behaviour. Here CPU time used between demands for pages is also modelled by the use of a probability distribution.

Since the simulator was required to allow comparison of behaviour of the system under different operating conditions and resource allocation algorithms, repeatability of the simulated processes was of importance. Much effort was put into ensuring that process generation was repeatable and that each process behaved repeatably in its demands for I/O and CPU service. This was achieved by supplying each process with its own seeds for the random number generators. Thus a process always generates the same sequence of service requests.

Experiments with the simulator were aimed at giving qualitative insight into the operation of the algorithms used. Therefore precision in the modelling of hardware was not of great concern. The I/O devices were simulated only to the level of a probability distribution of time taken to complete a requested operation. The distribution used was the negative exponential cut off at five standard deviations above the mean, (as in all cases in which we

employed exponential distributions), with a mean of 30 milliseconds. The individual pages of core were not modelled. However, the paging drum, which has a profound effect upon the working of a paging system, was carefully modelled. This model included sector queuing and carefully accounted for latency considerations.

3.5.2 The Simulated System

The system that has been simulated consists of a central processor, core storage and a paging drum. Each process is modelled as an alternating sequence of intervals of CPU usage punctuated by page faults and waiting for I/O. In our model we regard paging to be concerned with the drum and waiting to be concerned with disc I/O. We do not model contention for I/O devices nor do we require that any pages of a process be regarded as I/O buffers which are required to be in core during I/O activity.

The simulation allows various drum organisations to be modelled. The scheme employed in the simulations which we shall describe is a sector-queued organisation with priority ordering of the sector queues. We simulate the drum as being able to revolve every 17.5 milliseconds and capable of holding 4.5 pages per physical track. With each physical track divided into nine sectors it is possible to arrange that one page may be read in 2/9 of a 'physical' drum revolution (1/9 of a 'logical' drum revolution). This scheme is used in the Michigan Terminal System (MTS) at the University of Newcastle-upon-Tyne where an IBM 2301 drum is used.

A queue of requests is maintained for each sector. The maintenance of these queues is idealised in our model.

We assume the supervisor to be capable of maintaining the queues instantaneously. Thus reordering of a sector queue may take place up to the instant at which the drum read is to be performed for that sector. That is to say the time to set up channel programs for the drum is not modelled. We make the further assumption that the system is aware of the completion of the page transfer from the instant at which the transfer is completed physically. In a real system this 'posting' might not occur until the end of the logical drum revolution, or some other convenient time.

The sector queue which a particular page request will join is selected randomly with each sector queue having an equal probability of being chosen.

3.5.3 The Process Model

Each simulation experiment can involve one or more different classes of process. Each process is represented by a process profile indicating the size of the parachor and the amount of CPU and I/O time required. The parameters of individual processes are obtained by using the parameters of the profiles as the mean values of appropriate probability distributions. The size of the parachor, PCS, is sampled from the uniform distribution $U(X - \frac{1}{2}X, X + \frac{1}{2}X)$, where X is the profile's parachor size. The amounts of CPU time, CPUTIME, and I/O time required are selected in a similar manner.

Initially one process corresponding to each profile is activated. The processes of each profile have a regeneration period specified. A process corresponding to each profile will be entered into the mix at an interval after the previous process from the profile was generated. The interval is sampled from the uniform distribution $U(X - \frac{1}{2}X, X + \frac{1}{2}X)$, where X is the regeneration period for the profile. Regeneration continues until a preset number of processes have been generated.

The choice of the number of processes to be generated governs the simulation time, the simulation being completed

when the last process completes. We have set this parameter at 50 processes. This choice coupled with the process profiles described in 3.5.6 gives a simulated time of between 500 and 1200 seconds depending upon the resource allocation algorithms being simulated.

3.5.4 The Paging Model

Most paging simulators keep track of each individual page. They are either capable of actually executing programs or are driven by address traces previously gathered from the execution of programs. Such simulators are extremely laborious even for modelling the behaviour of a single process. Our model keeps track only of the number of pages which each process has core-resident and uses appropriate probability distributions to simulate the status of those pages.

The drawback of this method is that it is very difficult to produce an adequate model of an algorithm which uses the properties of individual pages. The extent of the difficulty may be judged from the simulation of the LRU derivative described in 4.4.

The amount of processing which a process will achieve before it suffers a page fault is calculated from a probability function which has as parameters the number of pages of the process currently in core and the parachor of that process.

The form of the probability distribution of the time to next page fault is based on published data, most notably that gathered on the M44/44X system, (Brawn and Gustavson,

1968). It attempts to model two distinct aspects of the behaviour of processes running in a paging environment. The first of these concerns the relationship that has been observed to hold between page fault rate and the amount of core that a process is allowed to use. The second aspect of process behaviour that is modelled is the gradual 'drift' of membership of the current locality.

The observed relationship between page fault and available core has been modelled in a perhaps over-simplified fashion by taking the probability that a given instruction causes a page fault to be

$$2^{-16\ \frac{RCP}{PCS}}$$

where RCP is the number of pages which the process has in core, and PCS is the parachor of the process. Even more arbitrarily we have assumed that the gradual drift of locality is steady and involves the process completely changing its current locality of reference three times during the course of its execution. The appropriate probability is

$$\frac{3*PCS}{CPUTIME*1000}$$

where CPUTIME is the total CPU time, in milliseconds, required by the process. The factor 1000 converts this to instructions.

From the combination of these two factors we obtain that the expected length of processing time that a process will achieve before page fault is given by

$$m = \frac{1-k}{k*1000} \quad \text{milliseconds}$$

where

$$k = \frac{2^{-16\,\frac{RCP}{PCS}} + \frac{3*PCS}{CPUTIME*1000}}{1 + \frac{3*PCS}{CPUTIME*1000}}$$

is the per instruction probability of a page fault being caused by a given instruction. We use m as the mean of a negative exponential distribution. This probability distribution was chosen for its 'lack of memory' property, which allows us to recompute the time to next page fault each pass through the basic timing loop. PCS and CPUTIME have the values described in 3.5.3.

It may be argued that the parameters of our paging function are difficult to justify. We accept this but we feel that there is no need to attempt to refine the function. We have simulated our algorithms using two further paging functions which are described in section 5.7. We see from comparisons of the simulations with these three functions, figure 5.18, that although the results differ quantitatively, the same qualitative behaviour is observed

for the algorithms studied. Since our interest is in the behaviour of the algorithms, this insensitivity to the exact form of the paging function diminishes the importance of providing a truely realistic paging function.

3.5.5 Simulator Output

Output from the simulator is in three forms. These are summaries of CPU and I/O usage and average queue lengths, the 'core map', and log information at the start and end of each process.

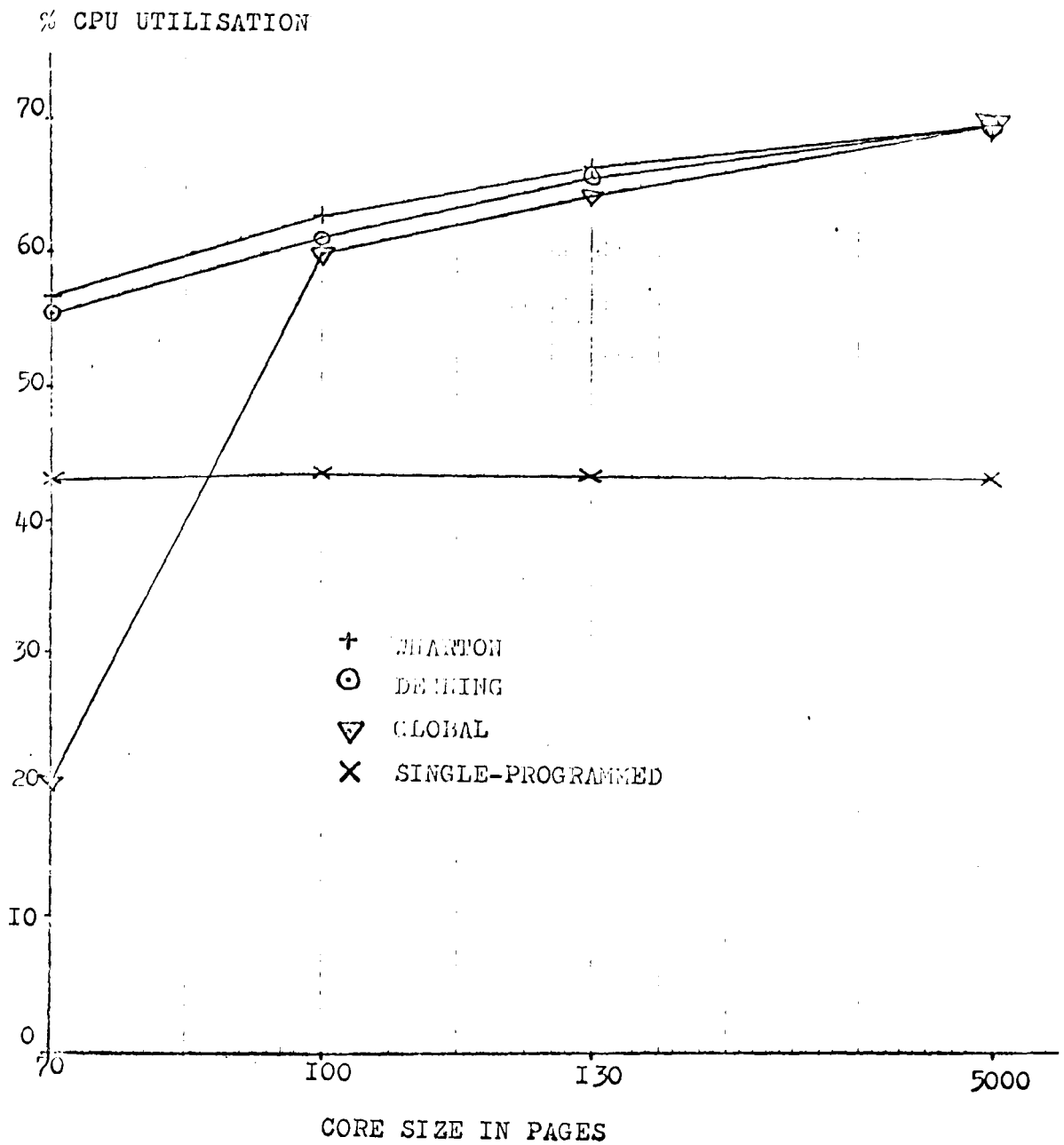
From the summaries over specified intervals and the whole simulation we can observe the gross effects of varying the parameters of the system. An example of a summary is shown in figure 3.1. Typically we plot this information giving graphs such as CPU utilisation against core size. An example is given in figure 3.2. These graphs have proved useful in indicating the susceptibility or otherwise, of an algorithm to thrashing.

The core map is an aid to insight into the detailed behaviour of the various algorithms simulated. An example is shown in figure 3.3. The core map gives a pictorial representation of the way in which the core is partitioned amonst the processes in the mix. A similar device was employed by Belady when working on the IBM M44/44X at the IBM T J Watson Research Centre, Yorktown Heights, New York. Each character of the core map represents a page of core, the code being the process number modulo ten. Thus 5 represents a page belonging to the 5th, 15th, 25th, etc., process introduced into the mix.

```
£££££££££££££££££££££££££££££££££££££

TIME TO BE SIMULATED IS 99999999

SYSTEM CONFIGURATION
NUMBER OF CPUS     1
CORE IS    70 PAGES
CORE THRESHOLD IS     0 PAGES
NUMBER OF DRUMS IS     1
NUMBER OF SLOTS IS     9
LOGICAL REVOLUTION IS          39
PAGE TRANSFER TIME IS           4
LIMIT OF JOBS IN SYSTEM IS    50
PRIORITY DRUM QUEUE
SECTOR QUEUEING
FIFO EXTERNAL SCHEDULER PRIORITY
TOP TO BOTTOM REQUEUE ON 0-0-T-S
RANDELLS RANDOM PAGE ALGORITHM
CORE MAP EVERY       1000 MSEC.
INTERVAL STATISTICS EVERY     100000 MSEC.

£££££££££££££££££££££££££££££££££££££

TIME IN WAIT STATE     290219 =40.4%
UTILISED CPU TIME    294661 =59.5%
SUPERVISOR TIME USED          0 = 0.0 %
HELLERMANS FIGURE OF MERIT = 0.0520
NUMBER OF JOBS DONE =  50
NUMBER OF PAGE FAULTS =   12941
NUMBER OF PAGE DROPS =   11364
TIME TO COMPLETE ALL JOBS    99.951
AVERAGE TIME IN PAGE WAIT    22.543
STANDARD DEVIATION OF TIME IN PAGE WAIT   35.602
TOTAL I/O TIME USED     312283 =62.10%
NUMBER OF I/O OPERATIONS PERFORMED   3046
HIGHEST NUMBER OF JOBS IN SCHEDULER QUEUE    21
AVERAGE NUMBER OF JOBS IN SCHEDULER QUEUE     9.410
NUMBER OF PREDICTIVE PAGE DELETIONS       0
NUMBER OF REFUSED PAGE REQUESTS    7904
MEAN VALUE OF LEVEL-TIME INTEGRAL     0.736
MEAN DRUM QUEUE LENGTHS
     0.088       0.085       0.095
     0.093       0.084       0.090
     0.082       0.086       0.101
MEAN NUMBER OF JOBS IN PAGE WAIT=    0.782
MEAN NUMBER OF JOBS IN I/O WAIT=    0.651

£££££££££££££££££££££££££££££££££££££
```

Example of a simulator summary

Figure 3.1



A graph of CPU utilisation against core size

Figure 3.2

An extract from a core map

Figure 3.3

```
PROTOTYPE          3
JOB    48 ON AT TIME     392021 REQUIRES        757 UNITS OF CPU AND     7 PAGES OF MEMORY
   REGENERATES AT TIME     401185
PROTOTYPE          2
JOB    49 ON AT TIME     393151 REQUIRES      12823 UNITS OF CPU AND    20 PAGES OF MEMORY
   REGENERATES AT TIME     433015
JOB    35 OFF AT TIME     395333
DRUM READS         340 I/O TIME       4097 I/O OPS         44 LEVEL-TIME INTEGRAL     2.000
ELAPSE TIME     125739 STRETCH FACTOR    23.116
JOB    36 OFF AT TIME     398382
DRUM READS         304 I/O TIME       2100 I/O OPS         18 LEVEL-TIME INTEGRAL     2.575
ELAPSE TIME     119303 STRETCH FACTOR    40.718
##########################################################
INTERVAL STATISTICS AT TIME     400000
CPU UTILISATION = 60.04%
NUMBER OF JOBS DONE =    35
NUMBER OF PAGE FAULTS =   1251
NUMBER OF PAGE SWOPS =   4793
NUMBER OF PREDICTIVE PAGE DELETES =      0
NUMBER OF REFUSED PAGE DELETES =   3411
AVERAGE TIME IN PAGE WAIT =    36.318
##########################################################
PROTOTYPE          0
JOB    50 ON AT TIME     401183 REQUIRES       1196 UNITS OF CPU AND     6 PAGES OF MEMORY
   REGENERATES AT TIME     408465
JOB    24 OFF AT TIME     429262
DRUM READS        1413 I/O TIME      19242 I/O OPS        190 LEVEL-TIME INTEGRAL     1.000
ELAPSE TIME     162209 STRETCH FACTOR     4.258
JOB    37 OFF AT TIME     430793
DRUM READS        1050 I/O TIME       3234 I/O OPS         37 LEVEL-TIME INTEGRAL     1.782
ELAPSE TIME     141405 STRETCH FACTOR    31.305
JOB    39 OFF AT TIME     434399
DRUM READS         150 I/O TIME       2142 I/O OPS         23 LEVEL-TIME INTEGRAL     2.107
ELAPSE TIME     121926 STRETCH FACTOR    41.519
```

Figure 3.4 An extract from a process log

Since codes are laid out according to the numerical order of the processes we can readily deduce from the other contextual information provided by the simulator the ownership of any page. Each page of core is represented by the code of the process occupying it, or by an '*' if it is unoccupied, or by an 'L' if it is a page into which a read-in from drum is taking place. (Such a page is not credited to a process until the read has been completed since the page must not be available for replacement until the read-in is complete. 'L' represents 'locked in core').

The contents of core may be displayed at any preset interval. A trace showing core contents every simulated second has been found adequate. Such core maps have been our major tool in confirming our predictions or understanding the behaviour of the core allocation algorithms. They have led on several occasions to the development of new algorithms.

The process log information provides a valuable complement to the core map. It aids the identification of the codes in the map. It also records the parameters of the processes. In addition it is useful when examining the service provided to the various classes of process. The ratio of the time a process spends in the system to its CPU usage is of importance here. This 'stretch factor' is a useful guide to the ability of the system to provide effective service. An extract from a process log is given in figure 3.4.

3.5.6 The Simulated Mix

The simulation model described above has been used to conduct a series of experiments determining the behaviour of various resource allocation strategies. The workload simulated in the majority of these experiments was an attempt to model a mix representative of that occuring on MTS at Newcastle. The model workload is composed of three components each with their own profile.

a) Small processes with a parachor of 5-15 pages and of the order of 1 second of CPU time and 3 seconds of I/O time. These were intended to represent interactive work such as editing. This type of work was estimated to demand 10% of the available CPU time.

b) Medium processes with a parachor of 15-45 pages and of the order of 20 seconds CPU time and 20 seconds I/O time. These were intended to represent the compilations and runs of simple programs which are the bulk of the work presented to MTS. These were estimated to demand 40% of the available CPU time.

c) Large processes requiring 25-75 pages and of the order of 100 seconds of CPU time and 33 seconds of I/O time. These were intended to represent the CPU bound component of the workload which accrues

from the research work of the university. These were estimated to demand 25% of available CPU time.

No attempt was made to model really large processes since these were unlikely to be run during a normal MTS session.

Overall the simulated mix demands 75% of CPU time. This was arranged by careful choice of the regeneration period for each profile. In our simulations using this 'standard mix' the maximum CPU utilisation obtained even when there is no contention for core has been 70%. This can be shown to be the maximum obtainable by considering the number of pages which each process will demand due to initially loading and because of change of locality which is modelled in the page fault probability function. This number of page faults is independent of the number of pages of core available. The idle time involved in accessing these pages depresses the maximum obtainable CPU utilisation from 75% to 70%.

We now have reason to believe that this 'standard workload' is more severe than the actual MTS mix. However this severity has been useful because the simulator has shown us how the various core allocation algorithms have reacted to the wide variety of circumstances which this load causes.

Since our interest has been in the qualitative analysis of resource allocation algorithms we have not attempted to improve the agreement of the actual and modelled workloads. In fact where we have wished to illustrate a particular feature of some algorithm we have employed very unrealistic workloads consisting entirely of identical processes.

3.5.7 Performance against Design Objectives

The modular structure of the simulator has proved very successful in allowing the modelling of various algorithms. The inclusion of new algorithms proved to be a simple matter. The structure was also useful in the initial implementation of the simulator. The first simple implementation provided results from which we could deduce the necessity for more precision in certain areas such as the drum model. This precision was easily added.

The event driven structure coupled with the specification of process characteristics and paging behaviour by probability distributions proved to be a most convenient level of simulation. It was possible to carry out many simulations inexpensively. Typically we were able to simulate 1000 seconds of operation in 200 seconds of CPU time on the IBM 360/67. In simulations involving traces of processes the CPU time used can greatly exceed the simulated CPU time. We also lost little scope in the type of algorithms that could be modelled and we were able to exhibit contention for core.

With regard to obtaining accuracy in the results, well tested random number generators were used. These were the IBM GPSS package random number generator, used to generate the process characteristics and seeds, and a multiplicative

generator (Marsaglia and Bray, 1968) used to generate the various times between events.

Taking all of those simulated CPU utilisations for Multiprogramming Limit equal to one and core size of 70 pages, where the core allocation algorithm would theoretically give the same CPU utilisation as uni-programming (twelve values), the Student t-distribution gives a deviation of approximately 0.2% at the 99.9% confidence level. Taking all of those simulated CPU utilisations for 'free' Multiprogramming Limit and 70 pages of core, where the core allocation algorithm would theoretically give the maximum obtainable CPU utilisation (eighteen values), the Student t-distribution gives a deviation of 0.1% at the 99.9% confidence level. Thus our comparisons of the core allocation algorithms using the simulator may reasonably be made.

CHAPTER 4

Previous Applications of Feedback to Core Allocation

4.1 Introduction

In this chapter we analyse four applications of feedback control to core allocation, using the system simulator described in 3.5, which preceded our own work. They are Wharton's algorithm (Wharton, 1971), Denning's algorithm (Denning, 1968b) a 'global' algorithm and the Load-leveller (Shils, 1968). They are representative of important classes of core allocation algorithms employing feedback control. Each of them provides insights into the problem of thrashing and ways of overcoming thrashing. These algorithms and their analyses provide the basis of our own work.

Wharton's algorithm employs inherent feedback control. The measurement of current localities of reference is implicit in the allocation strategy. As we shall see the control is stable even though the measurement of current locality is crude.

Denning's algorithm explicitly attempts to observe the current locality of each process and base allocation on these observations. The algorithm is interesting in that it uses model-driven feedback control.

Both Wharton's and Denning's algorithms have a hierarchy of control as we discussed in 3.3. The control is divided into a policy to select a process to lose a page and a subordinate policy to select a particular page from that process. The algorithms described in fact form the process selection part of the control. They may have any page replacement algorithm which is local to each process incorporated into them.

The Global algorithm is of interest because there is no separation of control as discussed above. Thrashing is frequently observed in systems employing these 'global' algorithms, so it is instructive to analyse such a strategy in order to understand the reasons for this.

The Load-Leveller provides an outer level of control in 'global' algorithms. This outer control is not involved with choosing pages to be replaced. Furthermore no modification of the existing core allocation policy is required to implement this extra level of control. This is an interesting approach which deserves consideration.

4.2 Wharton's Algorithm

This strategy was proposed by R M Wharton (1971) as an extension of the work of Belady and Kuehner (1969) on biased page replacement algorithms. The algorithm is as follows.

On occurence of a page fault any free page frame is allocated if there is such a page frame. When all free core has been allocated the lowest priority process which has pages in core and which is of priority less than or equal to the process causing the page fault is chosen to have a page replaced. The priority order is fixed externally to the core allocation algorithm. If no such process exists then the page request is denied and the requesting process cannot proceed until a higher priority process frees core.

This latter situation may arise if one assumes that all pages are loaded on demand. A process which has no pages in core may be dispatched but will page fault immediately. Such a process may then be unable to make a page replacement because of its priority and so be suspended.

The philosophy behind Wharton's algorithm is to give the top priority process the service it would obtain if it were running by itself in the system. Further processes are then dispatched as background work utilising any

core not required by the highest priority process. The scheme is a logical extrapolation of Belady and Kuehner biased replacement strategy where over some period one process is treated preferentially in the allocation of core, the others being treated equally. (Belady and Kuehner, 1969). Here that system of preferences is visualised as being extended to all processes with the bias referring to CPU as well as core allocation.

By this scheme the worst utilisation that will occur is the utilisation obtained by running the processes serially through the system. We acknowledge that if we were in fact producing a uni-programming system we would introduce optimisations and also that we are ignoring the interference caused by the supervisor dealing with the interrupts of lower priority processes. However, these should not cause large discrepancies and the observation is true to the precision of our simulation model.

The control upon the level of multiprogramming is obtained by the allocation of core. If a process has no core then it cannot affect the effective level of multiprogramming of the system. Since a process may only obtain more core by replacing pages of equal or lower priority the lowest priority processes may only obtain free pages of core. Also as the processes of higher priority acquire more pages the lower priority processes

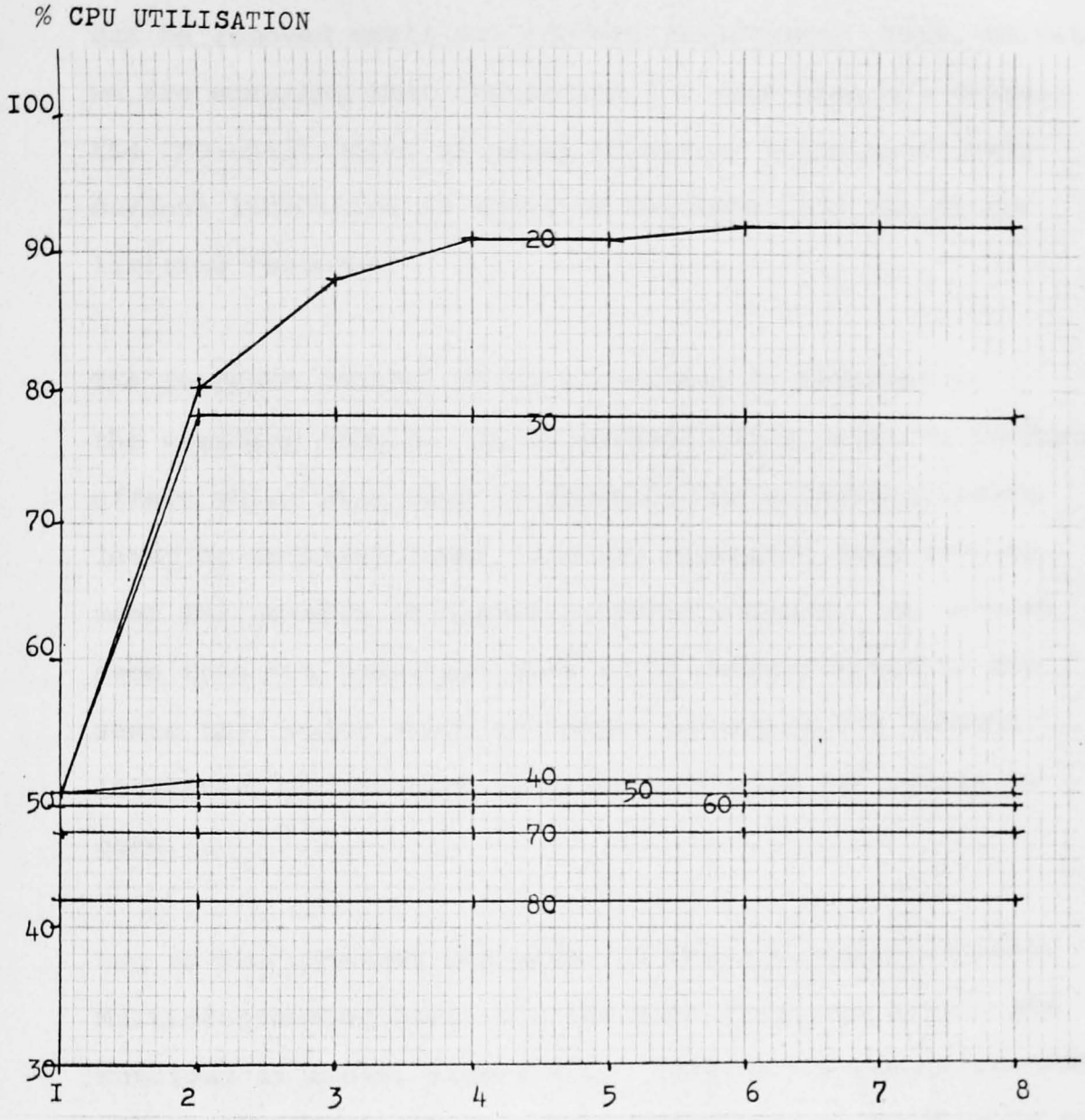
Wharton's algorithm - core map for identical processes

Figure 4.1

will be deleted from core. This is because their pages will be replaced by those of higher priority processes. They will be unable to obtain core and so will be suspended until a process of higher priority frees some core.

We can observe the manner in which this occurs by studying the core map in figure 4.1. The core map is that for a simulation of Wharton's algorithm when all processes in the mix are identical, each having a 20 page parachor. The priority order employed, as in all of our simulations, was first-come-first-served. A core size of 80 pages was used. We see that after time 2 seconds the core allocations of processes 1, 2 and 3 increase at the expense of process 4 until that process is deleted from core. Thus the effective multiprogramming level has been reduced. After this time we see that the core allocations of processes 1 and 2 increase at the expense of process 3.

It is interesting to examine the number of pages occupied by processes 1, 2 and 3 at the time that process 4 is deleted. They have 29, 26 and 25 pages respectively. Each has in excess of its parachor. We see that the highest priority processes accumulate pages in core which no longer belong to their current localities and unless a process becomes the lowest priority process



Identical parachors of 20 (10) 80 pages

Wharton's algorithm - CPU utilisation against Multiprogramming Limit for identical processes

Figure 4.2

which has core, there is no mechanism by which these pages can be removed until the process terminates. Thus, whilst we are ensuring that contention for core does not depress the CPU utilisation of those processes which have their current localities in core, we may make poor use of our limiting resource.

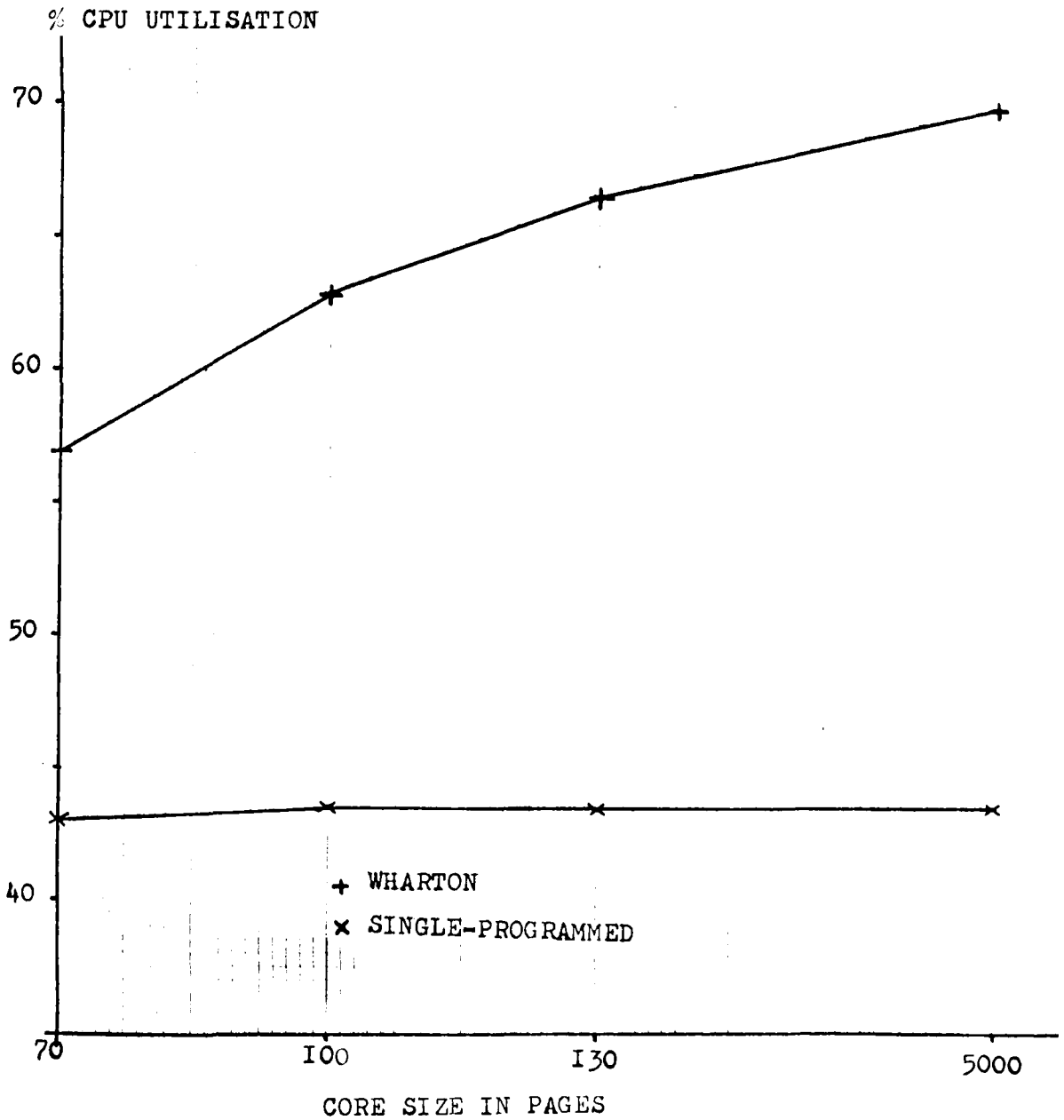
The feedback control in this strategy is inherent in the strategy itself. It is essentially a positive feedback effect which will tend to decrease the multiprogramming level by deleting lower priority processes from core to meet the demands of higher priority process. As we have seen this may cause the core to be underutilised in the sense that pages which no longer belong to the current localities of the high priority processes can remain in core.

Let us now consider the graph of CPU utilisation against Multiprogramming Limit for the same simulated system and workload as above, figure 4.2. (The Multiprogramming Limit is highest number of processes which will be allowed by the simulator to compete for system resources. It may be that a particular scheduler or core allocation algorithm will establish its own effective level of multiprogramming. However, the Multiprogramming Limit which is an initial parameter for each simulation will never be exceeded). We see that as Multiprogramming Limit increases from 1, CPU

utilisation increases to some maximum which it then maintains with only slight variations. The interpretation of this behaviour is that for a given workload Wharton's algorithm implicitly sets a level of which it will multi-program. When the preset limit, Multiprogramming Limit, is less than the level at which the algorithm is capable of working the CPU utilisation will be below the maximum attainable. As the preset limit increases the CPU utilisation improves to the maximum. Subsequent increases in Multiprogramming Limit have no effect since further processes will be prevented from obtaining core.

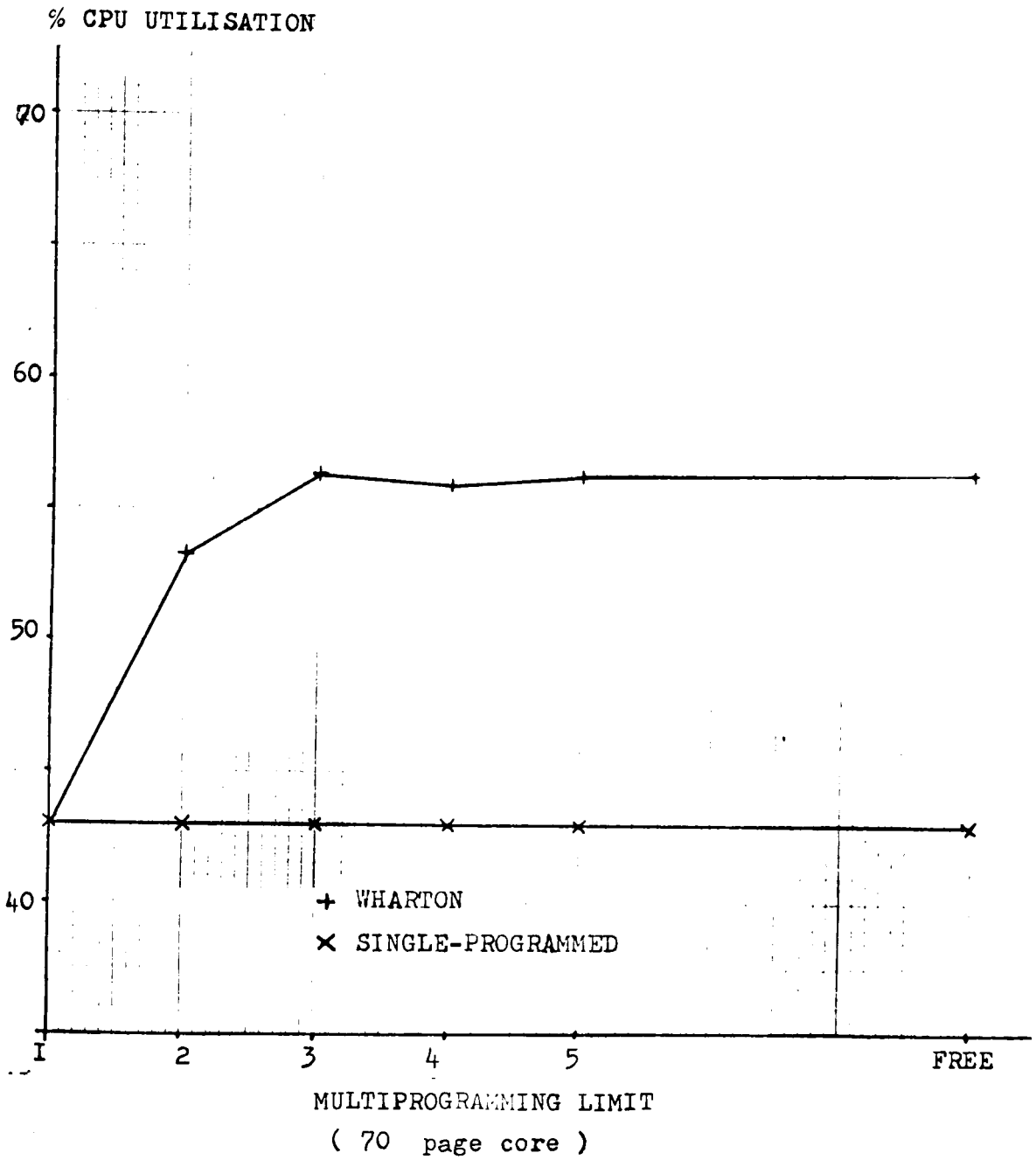
The Multiprogramming Limit marked as FREE in figure 4.2. represents a setting in excess of the number of processes to be simulated. Thus the effective level of multiprogramming observed will be that determined by the simulated system. The results labelled SINGLE-PROGRAMMED are the utilisations obtained by running the processes serially through the system and may be used as a basis for comparison of the various algorithms we shall simulate.

Simulation results for Wharton's algorithm where the standard workload was used are given in figure 4.3, CPU utilisation against core size, and 4.4, CPU utilisation against Multiprogramming Limit. In figure 4.3 the core size of 5000 pages may be thought of as an 'infinite' core, since with the standard workload no page replacements are required with this amount of core available. This core



Wharton's algorithm - CPU utilisation against core size for the standard mix

Figure 4.3



Wharton's algorithm - CPU utilisation against Multiprogramming Limit for the standard mix

Figure 4.4

size allows the maximum CPU utilisation obtainable for the load presented to be established since there is no delay caused by contention for core. These figures show the same behaviour as described above.

We see then that Wharton's algorithm guarantees that thrashing cannot occur and therefore gives a firm basis upon which to design further core allocation policies. However, Wharton's algorithm makes poor utilisation of core and we would hope to improve substantially upon its performance.

4.3 Denning's Algorithm

A basis for a promising group of strategies is the Working Set model of process behaviour (Denning, 1968b). The working set of a process at time t is the set of pages which the process accesses during the interval (t-T,t), where T is a fixed time interval. Denning has claimed the working set to be a good estimator of the set of pages which a process will access during the interval (t,t+T). Denning proposed the following strategy based upon the Working Set model.

The working set size of each process - the number of pages in the working set - is estimated in the manner described below. Processes are allowed to compete for core on the basis of the sum of their estimated working set sizes. A set of processes is chosen by some means such that the sum of their estimated working set sizes does not exceed the size of core.

When a page fault occurs core is allocated for the required page provided the estimated working set size of the requesting process exceeds the number of pages which the process has in core. When these two quantities are equal, core is allocated and the estimator increased by 1. (It is originally set to 1). However, this will only occur if the new sum of the estimators does not exceed core. If the latter does occur the process of lowest priority

is removed from core, its estimator being set equal to the number of pages it had in core at that time. Allocation continues on the basis of the reduced set of processes.

Further processes may be added to the set competing for core provided the sum of estimated working set sizes does not exceed core.

Denning's algorithm is a model driven feedback control system. A model - the Working Set model - is assumed for the behaviour of each process. The parameters of the model are the working set sizes. These are estimated from measurements of the paging activity. (We may think of observing the occurrence of page faults as measuring). The model of the controlled system is a 'core' into which 'processes' whose memory requirements are assumed equal to their estimated working set sizes, may be inserted. Decisions upon the controls to be applied to the actual controlled system are based upon whether the proposed action if applied to the model would cause the modelled core to be overpartitioned.

The controlled variable of this feedback control is the estimated sum of working set sizes. The command value of this variable is implicitly set to the size of available core. This denotes that ideally we would wish to utilise

the whole of core. The further requirement is made that the command value should never be exceeded. This expresses the wish that core should never be overpartitioned. However, it is difficult to ensure that the sum of the working set sizes is less than the core size. Therefore, the weaker condition involving the sum of the estimated working set sizes is used. If the estimates are good this will ensure that overpartitioning of core will be rare.

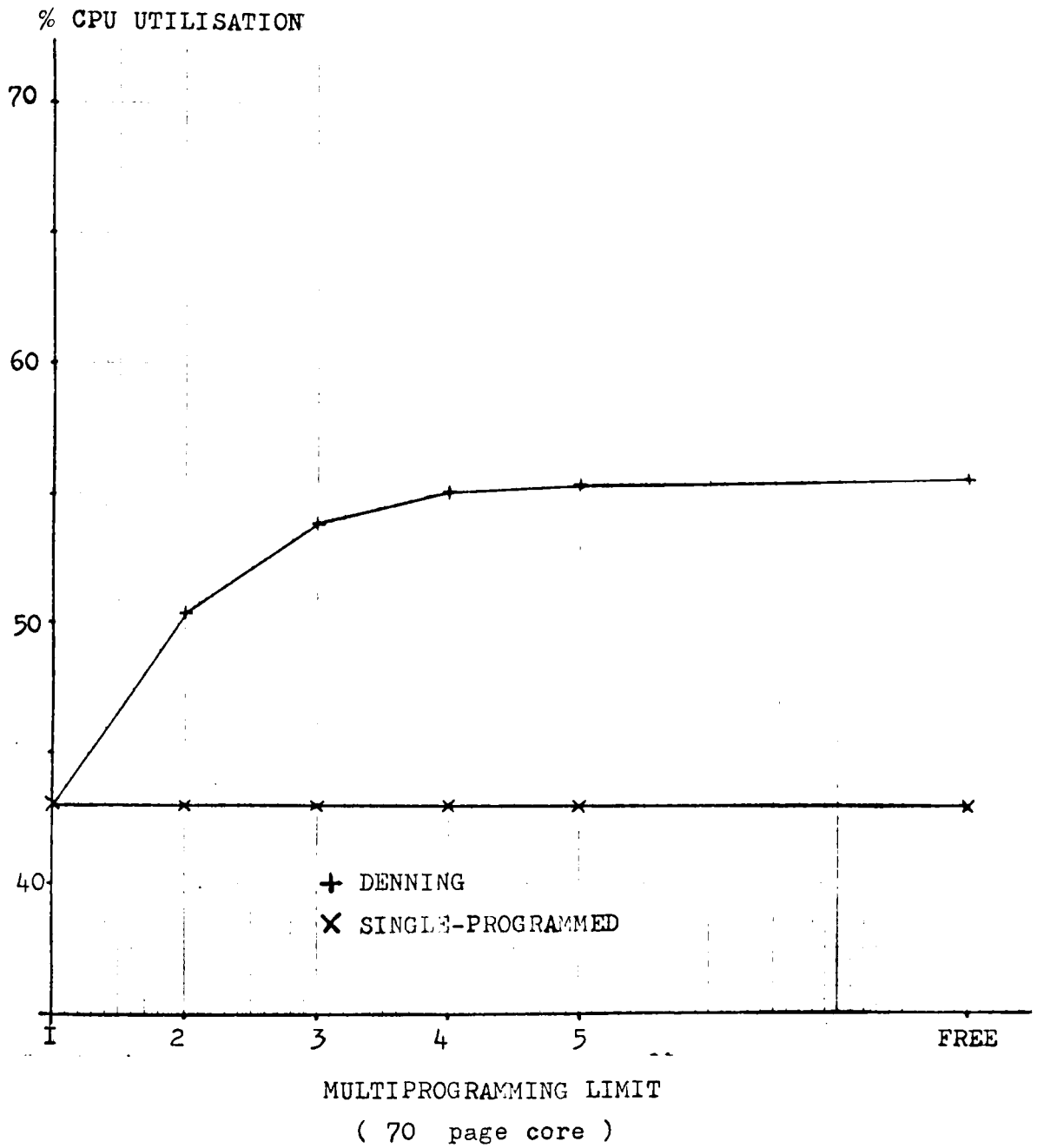
In order to define a steady state condition free core is allocated on a strict priority basis. Core is allocated, starting with the highest priority process, until the next process in priority order has an estimated working set size in excess of free core. Allocation is not made to this process and no attempt is made to allocate to lower priority processes. By choosing this first fit policy the possibly endless disturbances involved in a best fit policy, intended to utilise as much core as possible, are avoided. It also prevents processes with large working sets from being deferred indefinitely.

The success of a model driven feedback control system depends upon the accuracy of the model and its parameters. Unfortunately, the parameter estimation of Denning's algorithm is not very effective, although the basic strategy will most frequently err by overestimating working set size. However, it can underestimate by the

policy of setting the estimate equal to the number of pages which the process has in core at the time it is suspended by the system. Such a value is dependent upon the memory demands of other processes.

Let us now consider the simulation results for the Denning algorithm using the standard workload. In figure 4.5 we present a graph of CPU utilisation against Multiprogramming Limit for a fixed core size of 70 pages. We see from this graph that Denning's algorithm has the very desirable property that its performance improves monotonically with Multiprogramming Limit. This confirms that it is effective in limiting the effective multiprogramming level of the system so as to avoid the occurence of thrashing. As with Wharton's algorithm this strategy sets a level at which it will multiprogram. Increases of Multiprogramming Limit in excess of this have no effect upon CPU utilisation.

A strategy very similar to that of Denning has been implemented in the ESOPE operating system. (Bétourné et al, 1971). The algorithm is a generalisation of Denning's in that the estimated working set size is incremented by n pages rather than one. The estimate is formed in the same way as in Denning's strategy, differing only in that it resets the estimate to the number of pages which the process has in core at the end of each time slice and does not reset the estimate when a process is suspended. This avoids the



Denning's algorithm - CPU utilisation against Multiprogramming Limit for the standard mix

Figure 4.5

dependence of the estimate upon the memory demands of other processes which occurs in the Denning algorithm. Because of this the ESOPE algorithm probably produces a better estimate than the Denning algorithm. The ESOPE algorithm was simulated but due to the use of a different scheduler, used at the request of the ESOPE system designers, the comparison with the results for Denning's algorithm was inconclusive.

4.4 The Global Algorithm

4.4.1 Description of the Algorithm

In section 3.3 we discussed the global application of algorithms derived from the study of the paging behaviour of single processes. Such algorithms have achieved popularity, in particular the Least Recently Used (LRU) algorithm (Belady, 1966). However they are susceptible to thrashing. It is instructive to analyse such a strategy to understand the reasons for this.

Ideally we would study the LRU algorithm. However since our simulation model takes no account of individual pages this is not possible. We have therefore derived a global algorithm taking the LRU algorithm as our starting point so as to retain the structure though not necessarily the properties of that algorithm.

The LRU page replacement policy is stated simply as replacing, at the time a page replacement is required, that page from amongst those in core which was referenced least recently.

The popularity of this algorithm may be because intuitively one would expect that the probability of reference to the least recently used page in the near future is lower than that of pages more recently accessed. Consequently it is

more likely that this page does not belong to the current locality of any process. Therefore the page is a good candidate for replacement. In accordance with this the LRU policy is shown to be very effective in Belady's studies based upon the traces of single processes.

Due to uncertainty as to whether the least recently used page is the best candidate for replacement, there seems to be no great danger of loss of effectiveness in employing an algorithm which forms an approximation as to which page was least recently accessed. If we assume that as on the IBM 360/67 a hardware facility is available which sets the reference bit of a page whenever the page is accessed (2.6) then we can implement the following approximate LRU strategy.

The strategy is to examine the reference bits of all pages in core after some appropriate interval, a review interval, and to compile this information concerning the accessing of pages in the form of reference statistics, one for each page currently in core. The method of obtaining a reference statistic is to interrogate the value of the reference bit at time t+St, r(t+St), resetting it, and produce the reference statistic

$$R_{t+St} = (I-\alpha).r(t+St) + \alpha.R_t \quad , \; 0 \leq \alpha \leq 1$$

where St is the review interval. We now have an exponential

decay of the reference information with a half life of α where α may be thought of as a smoothing factor.

The reference statistic for each in-core page is obtained when the reference bits are sampled at the end of the review interval. When a page fault occurs that page in core which has the least value of the reference statistic is selected to be replaced.

We have in the LRU algorithm, the basic functions of producing a reference statistic for each page and then making a global choice based upon these reference statistics. The problem we have is to replace the value of the reference bits of individual pages, a property our simulator could not model, with a property it could model. We must use a property of each process since individual pages are not modelled. Thus we would produce a reference statistic for each process and choose globally amongst the processes.

Let us consider the algorithm using the following reference statistic based upon the rates at which the processes use the CPU and cause page faults. The number of page faults caused by a process during the interval $(t,t+St)$, $F(t,t+St)$, and the CPU time used, $C(t,t+St)$, are monitored. At the end of the interval, a usage statistic

$$R(t+St) = (I-\alpha).\frac{C(t,t+St)}{St} . F(t,t+St) + \alpha.R(t)$$

is defined for each process contending for core. During the interval (t,T+St) any page replacements required are made by replacing pages of the process with the least valued usage statistic.

If we denote the amount of CPU time which a process obtains by the term 'local time', then the local time of a process represents the passage of time from the point of view of the process. A high page fault rate in local time implies that a process does not have its current locality of reference entirely core-resident. Also we note that the rate at which a process references its pages is dependent upon the rate at which its local time progresses relative to the real time of the CPU. A process which obtains little central processor time is as likely to have pages unreferenced for long periods as a process which obtains a great deal of CPU and page faults rarely suggesting that it has at least its current locality in core. The term

$$\frac{C(t,t+St)}{St} \cdot F(t,t+St)$$

is an attempt to express these ideas mathematically.

This algorithm attempts to order the processes contending for core such that the more likely a process is to have pages which have been unreferenced for long periods, the more likely the pages of that process are to be replaced.

The algorithm described above we shall call the Global algorithm. As with the LRU algorithm it makes global decisions about which process shall have a page replaced. However, we do not claim that it has the properties of the LRU algorithm.

4.4.2 Mathematical Analysis

To model the Global algorithm mathematically let us suppose that there are M pages of core available and that we are multiprogramming two processes, a and b, with parachors da and db respectively. We will assume that a page fault is serviced in S milliseconds and that this service time is constant.

We further assume that the probability of a process causing a page fault at each instruction execution is p(r) = 2 ** (-16r/d) where d is the parachor and r is the number of pages of core which the process is occupying immediately prior to the instruction execution. This is similar to the probability function used in the simulator. Thus the expected CPU time used before the process causes a page fault is

$$m(r) = \frac{1 - p(r)}{p(r)} \times \frac{1}{1000} \quad \text{milliseconds}$$

where we assume a CPU capable of executing one million instructions per second. m(r) takes this form since if a page fault occurs at the k-th instruction execution only k-1 time units have elapsed since the last page fault.

The usage statistic for process x, $R_x(t_i)$, is updated at times t_i where $t_i = t_o + i.St$. The process with the lowest value of the usage statistic after time t_i is that

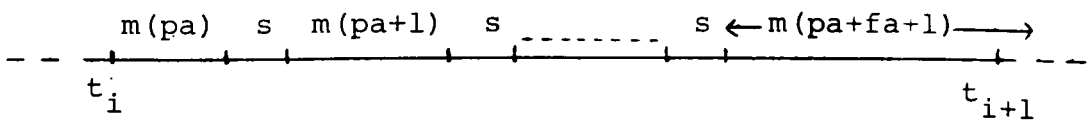
from which page replacements are to be made during the interval (t_i, t_{i+1}).

In order to find values for the number of page faults caused by each process during a period of length St, we must make two further simplifying assumptions. The first is that there is no statistical variation in the time between page faults. The second is that each process obtains the CPU whenever it requires. This can only be guaranteed by providing multiple CPU's and so we shall assume that each process has its own CPU. The latter assumption is not too unrealistic, for when core contention occurs CPU utilisation will be low and contention for CPU will seldom be experienced.

Suppose that process b has the lowest value of the reference statistic at time t_i, and that process a has pa pages at that time. Process b has pb = M - pa pages since we are only interested in the cases in which there is contention for memory. Then during the interval (t_i, t_{i+1}) process a will cause

$$fa = \left\lfloor \frac{St}{\sum_{i=pa}^{pa+fa} (m(i) + S)} \right\rfloor \quad \text{page faults}$$

where $\lfloor$ denotes 'the greatest integer less than'. To see this consider the diagram below.



If fa < pb then process b will lose fa pages during the interval, otherwise it will lose pb pages and be deleted from core. The number of page faults caused by process b is rather more complicated since the amount of core occupied by it is dependent upon the activity of process a.

At the end of the interval marked as m(pa) in the diagram process b will have lost a page and will have only pb-1 pages. Its page fault characteristics will change therefore. Thus the number of page faults caused by process b is given by repeating a calculation similar to that for program a with St replaced successively by the values m(pa) + S, ..., m(pa+fa) + S. The final interval must be treated separately with St being replaced by $St - \sum_{i=pa}^{pa+fa} (m(i) + S)$.

When process b has i pages the appropriate term is

$$\frac{m(M-i) + S}{m(i) + S}$$

These terms occur because process b replaces only its own pages. Thus,

$$pb^1 = \sum_{i=pb-fa}^{pb} \left\lfloor \frac{m(M-i) + S}{m(i) + S} + \left\lfloor \frac{(St - \sum_{i=pa}^{pa+fa} m(i) + S)}{m(pb-fa) + S} \right\rfloor \right\rfloor$$

Since we have assumed multiple processors we have that the CPU time used by process x is

$$St - fx.S \quad \text{milliseconds.}$$

Thus by a number of simplifying assumptions we are able to calculate the usage statistics.

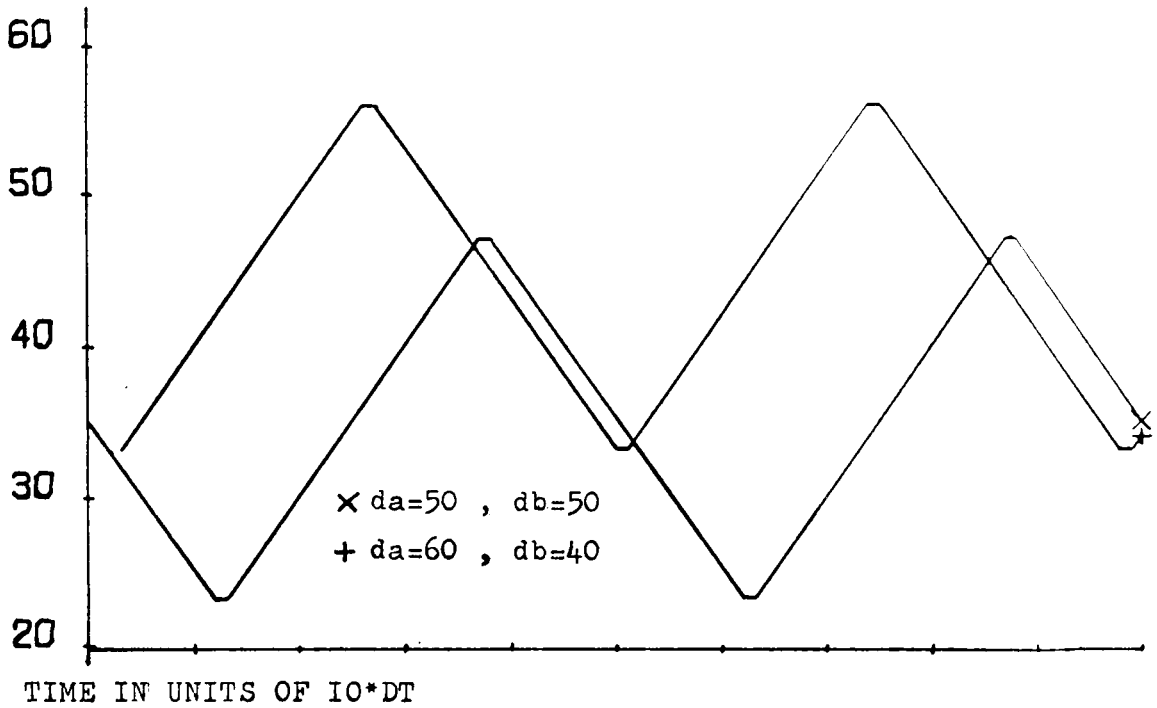
Although we have achieved a mathematical model of this algorithm it has required many unrealistic simplifying assumptions about the system. Many features which will be of importance in a practical situation have had to be excluded from the model in the interests of mathematical tractability. In order that we may study features which are not amenable to mathematical analysis we have also simulated the algorithm.

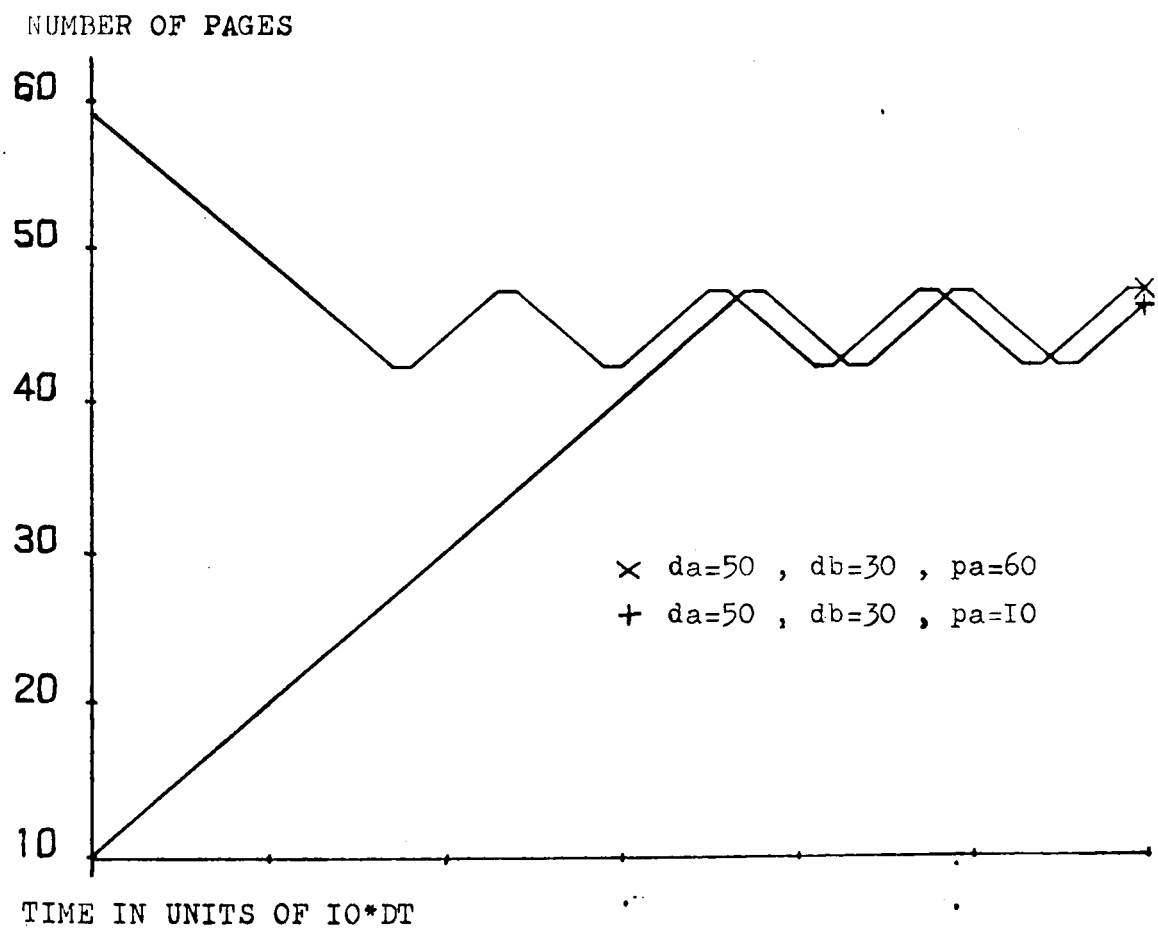
4.4.3 Discussions of Mathematical and Simulation Results

As we discussed in section 3.3 a global algorithm essentially allows all processes to compete for core on an equal basis. This strategy should lead to an equal sharing of core in some sense. Therefore, we would expect that thrashing would occur in a system using the algorithm.

The feedback component of the algorithm is in the form of a superimposed monitor which provides information on the past access characteristics of each in-core page. The access characteristics are quantified by a process of discretisation as described in section 2.6. Thus the algorithm does not conform to the structuring which we proposed for a core allocation policy in section 3.3.

Let us first consider the results of the mathematical model. Figure 4.6 shows the way in which core is divided between two processes a and b with parachors da and db. The number of pages of core occupied by process a is plotted against time. Process b occupies the remainder of core. The time scale is in units of St, with St chosen to be milliseconds. The core size was 70 pages, $\alpha = 0.1$ and the time to service a page fault was 17 milliseconds.





Global algorithm - modelled division of core with time

Figure 4.6

We can observe from these graphs that the algorithm tends to equalise the paging rates of the two processes. Using the page fault probability of the mathematical model, we require for equal paging rates that

$$2 ** (-16.pa/da) = 2 ** (-16.pb/db) .$$

That is $$\frac{pa}{da} = \frac{pb}{db} ,$$

but $$pb = M-pa ,$$

so that $$pa = M . \frac{da}{da+db} \quad . . . . \quad (1)$$

Given below are the values of da, db, the value of pa given by (1), and the value of pa about which oscillations occur in the graphs of figure 4.6. The initial value of pa in those graphs is also given.

Parachors		pa from (1)	Average pa	Initial
da	db	Equal Paging Rates	from figure 4	pa
50	50	35	35	35
60	40	42	44	35
50	30	44	44	60
50	30	44	44	10

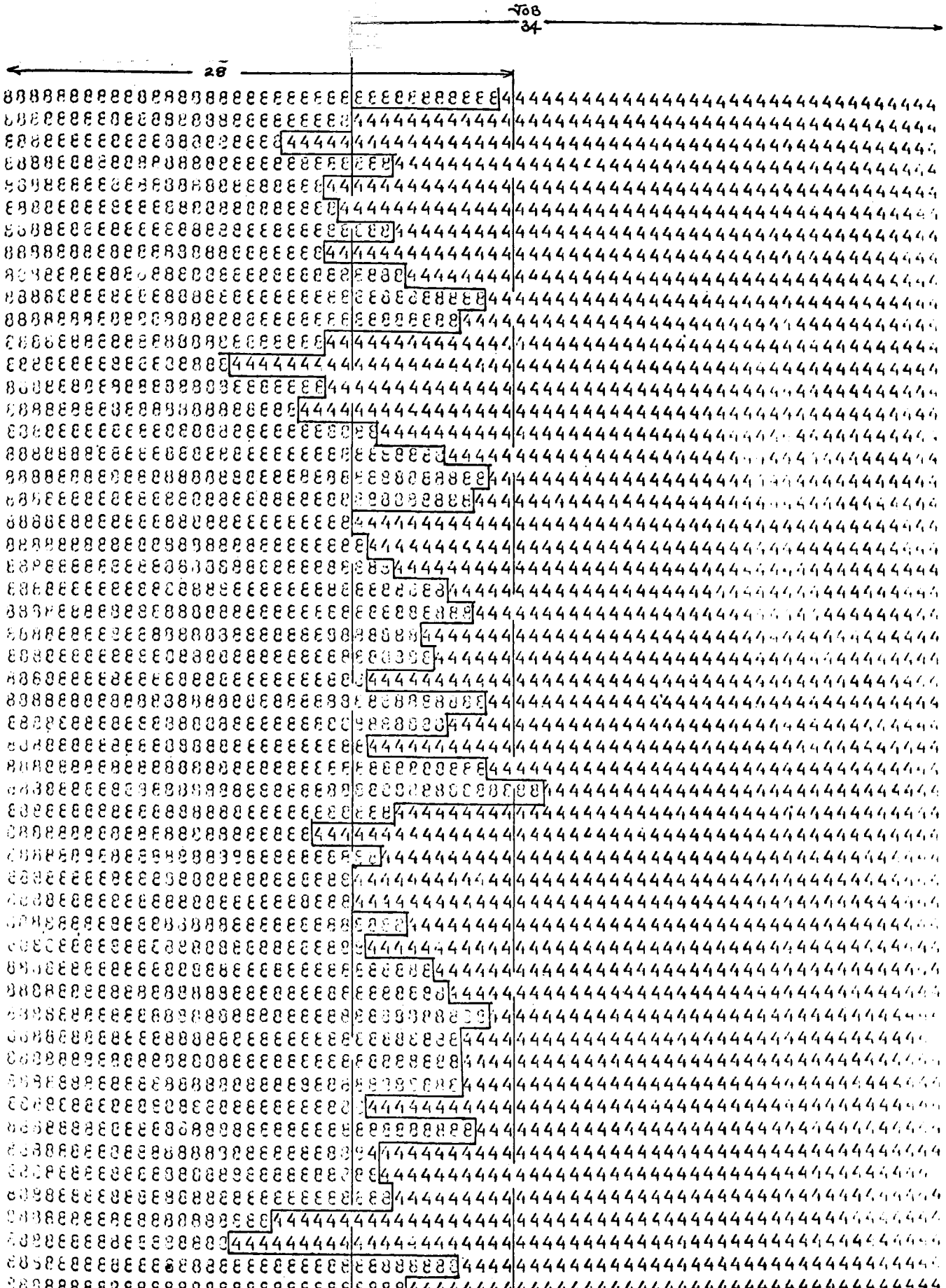
The value of pa about which the oscillations settle is not affected by the initial value although the settling time is. We see also that there is a good agreement between the values

of pa about which oscillation occurs and the values obtained by assuming that the paging rates of the processes are equalised. Therefore it seems likely that the ability of processes to compete on an equal basis will cause core to be partitioned in proportion to parachor. Such sharing of core will of course cause thrashing whenever core is overpartitioned.

We see from the mathematical model of the algorithm that there is a tendancy for the paging rates of each process competing for core to be equalised. In figure 4.7 we present a section of the core map of a simulation of the algorithm using the standard workload with a core size of 70 pages and Multiprogramming Limit of 2.

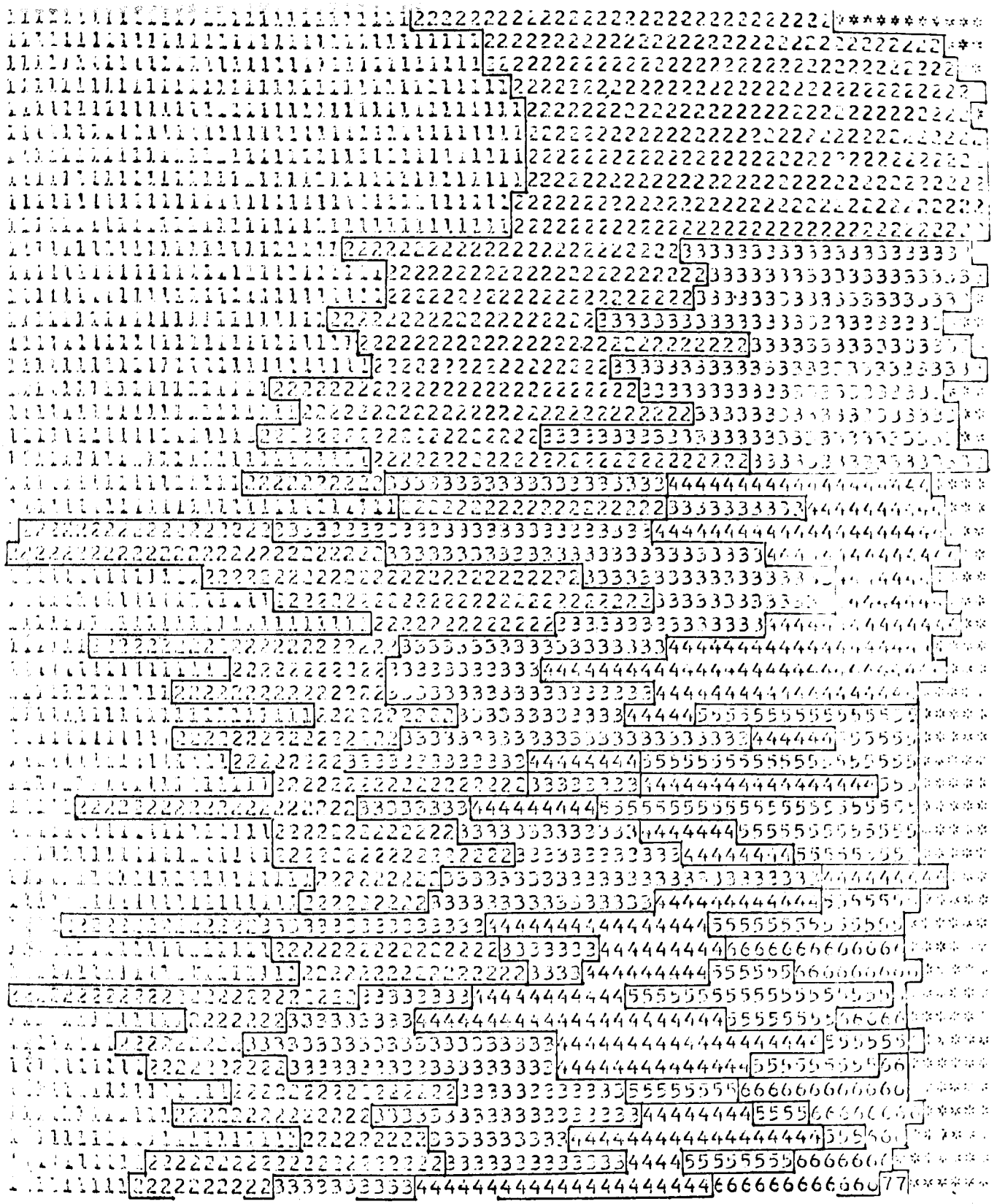
From time 359 to 412 seconds the core is occupied by process 28 with a parachor of 38 pages and process 34 with a parachor of 44 pages. The core is overpartitioned since the sum of the parachors is 82 pages. We observe that the division of core oscillates about a mean of 32 pages (from the left of the core map). This indicates that the core is being divided in proportion to parachor, that is that the paging rates of the processes are being equalised.

Let us now consider the core map of a simulation where processes were introduced into the mix in a controlled manner, figure 4.8. A core size of 70 pages initially occupied by 2 identical processes of parachor 30 pages,



Global algorithm - core map for the standard mix

Figure 4.7



Global algorithm - core map for identical processes

Figure 4.8

where each required 50% CPU time and 50% I/O time, was simulated. At 10 second intervals further identical processes were initiated.

We see from the core map that each process gains an amount of core as soon as it is initiated even though the core may be heavily overpartitioned. Since the scheduler has no load shedding or load limiting component incorporated into it, a new process will commence computation when the CPU is not fully utilised by the already initiated process. (Note that the Multiprogramming Limit was set to be in excess of the number of processes to be introduced into the mix in this simulation). Load shedding would occur due to this Global strategy only if the higher priority processes were fully utilising the CPU. (We used a first-come-first-served priority for CPU allocation). In such circumstances low priority processes would not reference their pages and so those pages would be removed from core under the algorithm.

However, if at any time the CPU became idle the scheduler would allow one of the lower priority processes to proceed and thus to compete for core. As competition for core increased the likelihood of the CPU becoming idle would increase due to replacement of pages belonging to current localities of reference. Further processes

would be allowed to compete. Thus a positive feedback effect occurs which leads to increasing overpartitioning of core and eventually to thrashing.

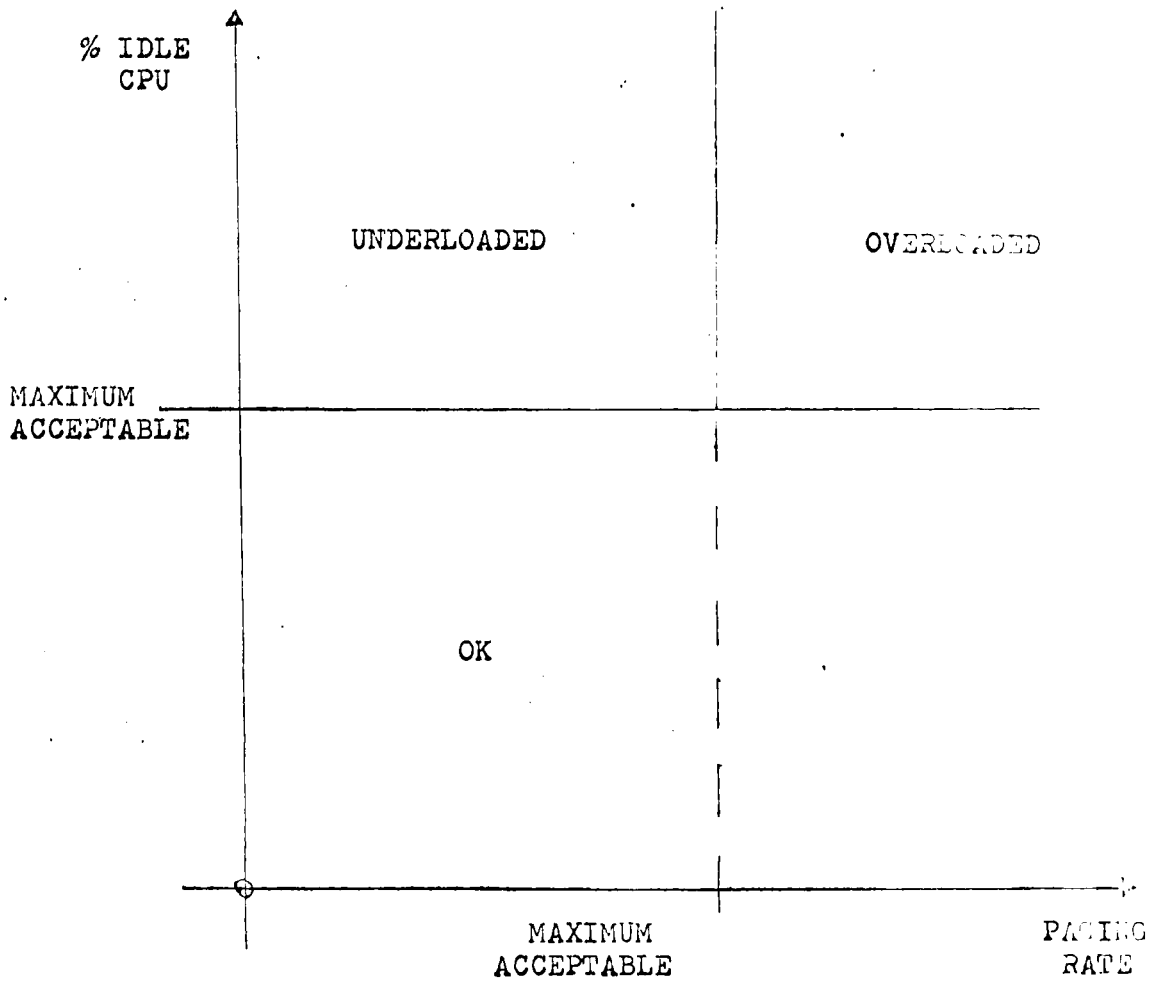
The study of this Global algorithm is instructive when considering the design of a core allocation or scheduling policy. It suggests that the raising of the multi-programming level to cover I/O processing is only effective whilst core contention is not being experienced. It further suggests the necessity of employing some form of load shedding dependent upon the level of contention for core, and that it is unrealistic to base load shedding or load increase upon CPU utilisation alone. It also raises doubts about the wisdom of allowing processes to compete for system resources on an equal basis. Equal sharing as we have seen may well reach to overload conditions. Such situations can be avoided by applying priorities which govern the availability of each system resource.

4.5 The Load-Leveller

The Load-Leveller (Shils, 1968) was implemented on the IBM M44/44X, an experimental machine built at the T J Watson Research Centre, Yorktown Heights to assess the feasibility of paging. The Load-Leveller was a special process which periodically assessed the degree of partitioning of core and dynamically adjusted the multiprogramming level. Decisions were taken depending upon the values of the percentage of idle CPU and the page fault rate during intervals of duration St. The state of the system was defined by comparing these values with maximum acceptable values. If idle CPU time was less than the maximum acceptable then the system was 'OK'. Otherwise if the page fault rate was too high the system was 'overloaded', if low it was 'underloaded', as shown in figure 4.9.

If the system was overloaded at the end of an interval then a process was suspended. That is the multiprogramming level was lowered. If the system was underloaded or OK then the multiprogramming level was increased. The Load-Leveller was thus a deliberate attempt to apply feedback control to the problem of thrashing by the dynamic adjustment of multiprogramming level.

The strategy comprises a feedback control of the superimposed type. Depending upon comparisons of the



The Load-Leveller - state diagram

Figure 4.9

measured values of the two controlled variables, the percentage idle CPU and the page fault rate, with their command values, the maximum acceptable idle time and page fault rate, the multiprogramming level was either increased or decreased. Such actions act as stimuli to the controlled system and subsequent measurements inform of the success or otherwise of the actions.

Essentially with this form of the control the error condition, unacceptable paging rate or idle time, must persist before the need for corrective action can be recognised. Thus the system performance may often be unacceptable. To reduce the time during which performance is poor it is necessary to make frequent measurements. However, this increases the sensitivity of the controller to random fluctuations in CPU usage and brief intervals of high paging activity.

There may well be a tendency for overcompensation to occur when the system is overloaded. In the period immediately following the deletion of some process there may still be high paging activity while the competing processes obtain the missing pages of their current localities which were removed as a result of overpartitioning. This may lead to a process being deleted unnecessarily. This tendency to overcompensate will increase as the sampling rate is increased. However

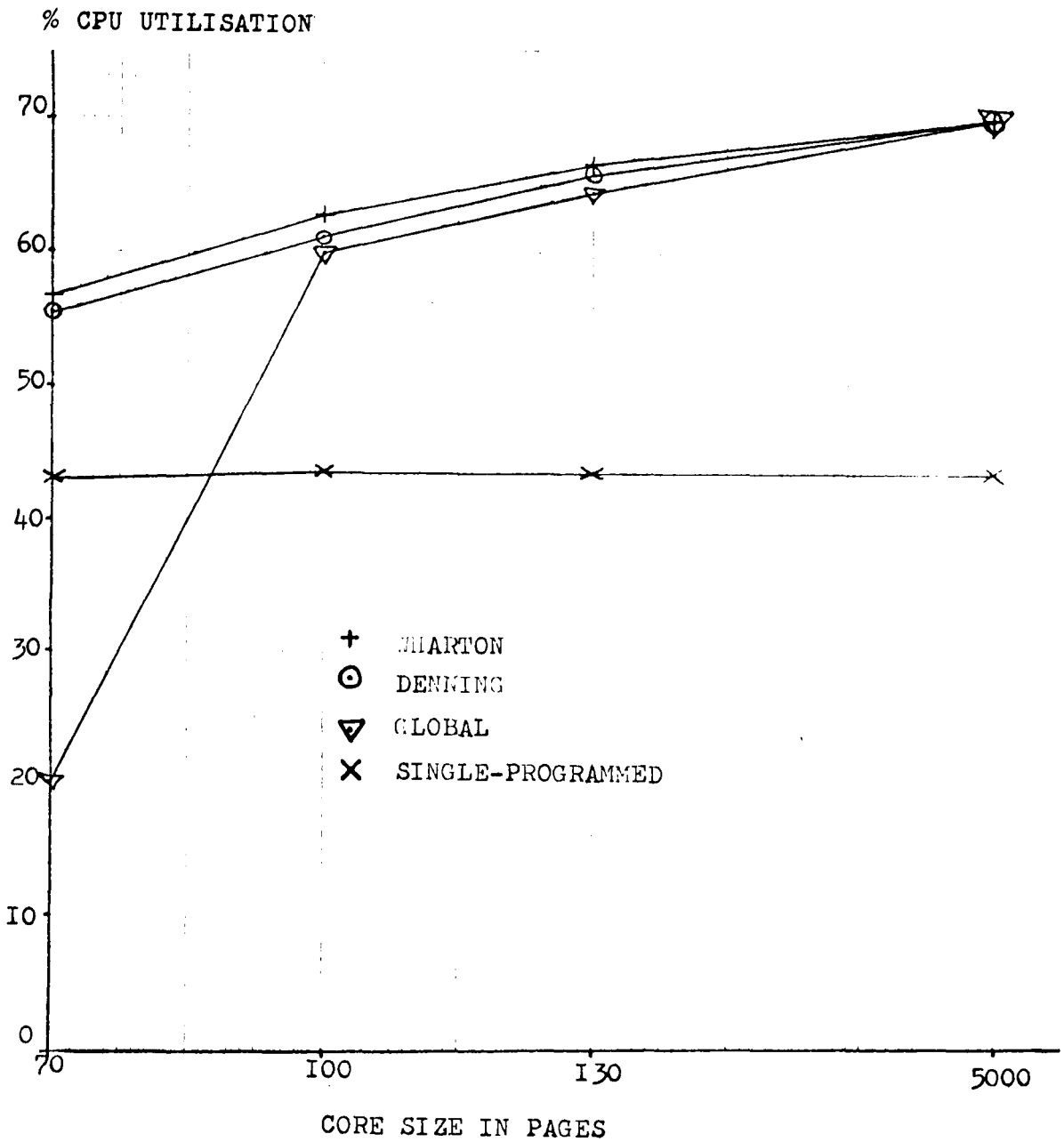
increasing the intervals between measurements leads to a greater proportion of unacceptable performance. Thus we have a classic example of the conflict between stability, speed of response and accuracy.

The Load-Leveller is designed to modify the load on the system when the situation has degenerated to an unacceptable degree. This mode of operation is unlikely to provide the best attainable performance.

4.6 Summary

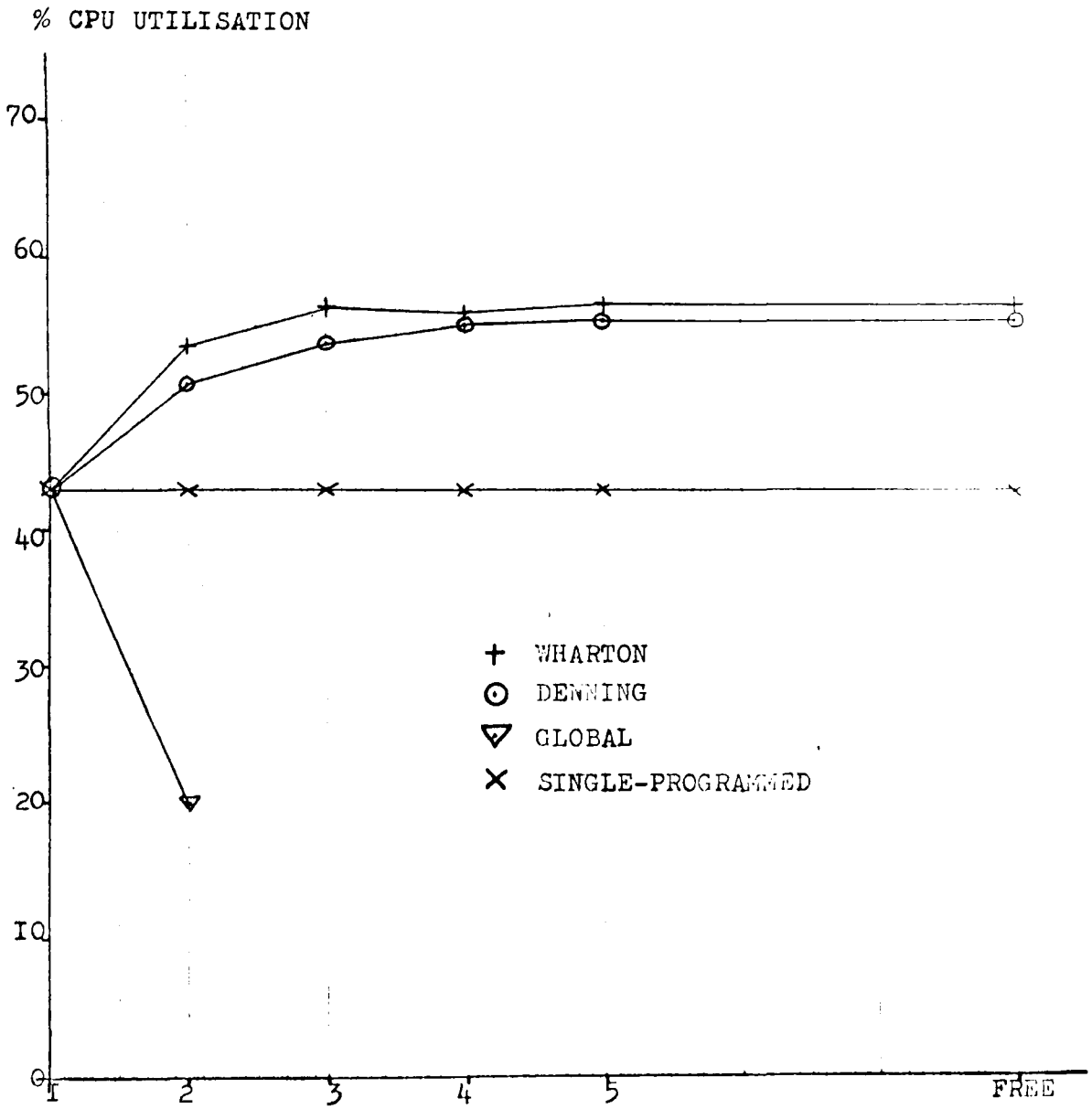
The core allocation algorithms we have discussed have the use of feedback control as a common feature. However, there is a great diversity in the success of its application as we see from figures 4.10 and 4.11 which are graphs of CPU against core size and CPU against Multiprogramming Limit respectively for three of the algorithms. (The Load-Leveller was not simulated because the work of Shils provided all the necessary information). The Global algorithm was susceptible to thrashing and the Load-Leveller could not prevent thrashing although it could limit the effect. From these two algorithms we can isolate features to avoid. Both the Denning and Wharton algorithms were effective in avoiding thrashing. However, both of these have shortcomings which we must overcome if we are to produce practicable algorithms.

The Global algorithm which we simulated was inferior to both the Denning and Wharton algorithms and was susceptible to thrashing. We would have expected this in view of the discussion of 3.3. The amount of core which any process may obtain is dependent upon the memory demands of the other processes competing for core. We have seen that the particular form of dependency in this algorithm leads to 'fair sharing' of core depending upon the demands of the processes. The slightest overloading of core will therefore



Summary of results - CPU utilisation against core size for the standard mix

Figure 4.10



Summary of results - CPU utilisation against Multiprogramming Limit for the standard mix

Figure 4.11

cause thrashing. Overloading is particularly likely in view of the load-shedding component. Load-shedding would only occur when CPU utilisation was high. Unfortunately, the probability of high CPU utilisation decreases as paging activity increases. Thus it is not realistic to base load manipulation on CPU utilisation.

We would expect to improve the performance of the Global algorithm if we could introduce a load-manipulating component based on a more suitable property of the processes. This is the purpose of the Load-Leveller. This process, which can be superimposed upon core allocation algorithms, uses CPU and paging disc utilisation to make load manipulation decisions. Shils showed it to be effective in limiting the incidence and duration of thrashing. However, he also found the CPU utilisation was frequently depressed below that of the uncontrolled algorithm. We discussed in 4.5 that this could be due to the insensitivity of the error measuring component of the control. We also noted the dangers of attempting to increase sensitivity. However, even with increased sensitivity the Load-Leveller could not overcome the dependency of memory allocations upon memory demands of other programs in the Global algorithm. This requires some priority mechanism and as we have seen from Wharton's algorithm rather more simple and effective load manipulation is then possible.

Wharton's algorithm successfully avoids thrashing by applying a strict priority ordering upon the processes competing for core. The priority order provides a very simple mechanism for load manipulation and also greatly diminishes the interdependencies of the allocations of core to the processes. Consequently, Wharton's algorithm shows better performance than the Global algorithm. Denning's algorithm which shows similar performance to Wharton's algorithm, is also successful in preventing thrashing. This is achieved by measurement of the core requirements of each active process and ensuring that the sum of those requirements never exceeds the available core. However both of these algorithms are inadequate in their utilisation of core. The Wharton algorithm allows high priority processes to accumulate pages in core which no longer belong to their current localities. Denning's algorithm is also prone to overestimate the size of the current locality, leading to poor utilisation of core.

From our analysis of these four algorithms we are able to draw certain conclusions about the manner in which core allocation may be successfully performed. Both the Denning and Wharton algorithms show that feedback of information concerning the memory demand of processes can be effectively employed to avoid thrashing and so is a technique worthy of consideration. The priority ordering of Wharton providing a lower bound upon the size of current locality

and the more direct measuring of Denning suggest two possible methods of applying this feedback of information. The Global algorithm demonstrates the dangers of uncontrolled competition for core and the need for effective control of the load presented to the system. It is encouraging to note that the successful algorithms employ a structuring which divides page replacement from the selection of a process from which to replace a page, whilst the Global algorithm does not. We suggested (3.3) that such a structuring would be a sound basis for a core allocation algorithm.

The success of the Denning and Wharton algorithms shows the utility of feedback control in core allocation. However, their short-comings in the utilisation of core suggest that they are capable of improvement, and we are therefore encouraged to seek ways in which the technique they use may be further developed.

CHAPTER 5

Core Allocation Using Feedback Control

5.1 Introduction

The analyses of the previous chapter highlight the importance of the control of multiprogramming level and the estimation of current locality in core allocation. Failure to do either of these can lead to thrashing. The extent to which these problems are successfully solved can considerably affect system performance. Thus the control of multiprogramming level and the estimation of current locality will be the main themes of this chapter. In particular we will introduce the concept of drain processes as a way of estimating current locality.

We found the Wharton and Denning algorithms were both effective in avoiding thrashing by the use of stable feedback control. However, both have shortcomings in core utilisation which need to be overcome. We will take these two algorithms as our starting point and consider ways of optimising their use of core.

First, we will take Wharton's algorithm as a basis and discuss our attempts to improve core utilisation by different priority schemes in the Horning and Randell algorithms. These modifications maintain the inherent nature of the

control. We will then introduce a form of drain process where the control is superimposed upon Wharton's algorithm. This is the Lynch algorithm. A further optimisation, the Lynch-Alderson algorithm, will then be introduced.

Next we will discuss the application of another form of drain process to the Denning algorithm. This will lead to a comparison of drain processes which we have discussed.

Finally, we will discuss an algorithm due to Hoare where settling time can be varied by the setting of a parameter. This algorithm will be of particular interest when we examine the importance of settling time in our control hierarchy.

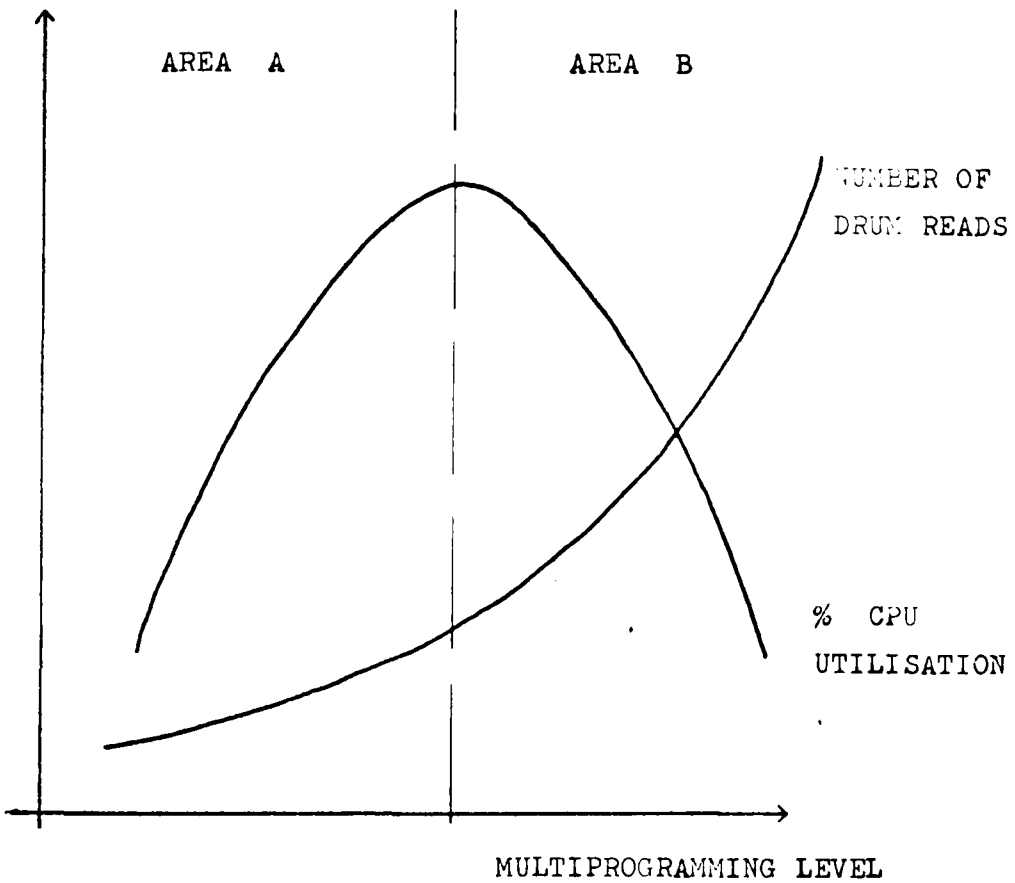
5.2 Horning's Algorithm

We have observed in the discussion of Wharton's algorithm that the higher priority processes retain pages which are superfluous to their progress. This algorithm, proposed by J J Horning, was derived from Wharton's algorithm and attempted to include a mechanism which would free this unproductive core.

As in Wharton's algorithm the allocation of system resources is biased by imposing a priority ordering upon the processes for access to both CPU and drum.

On occurrence of a page fault any free core is allocated if it is available. When all free core has been allocated, a process is chosen at random from amongst those which have pages in core. The probability of a process being chosen is proportional to the number of in-core pages which it has. A page is replaced from amongst the process's pages according to some appropriate scheme.

The reasoning that led to Horning's algorithm was as follows. If we superimpose graphs of paging drum reads versus multiprogramming level and CPU utilisation versus multiprogramming level we would expect to obtain a figure similar to figure 5.1. Obviously that level of multi-programming which gives the maximum CPU utilisation is



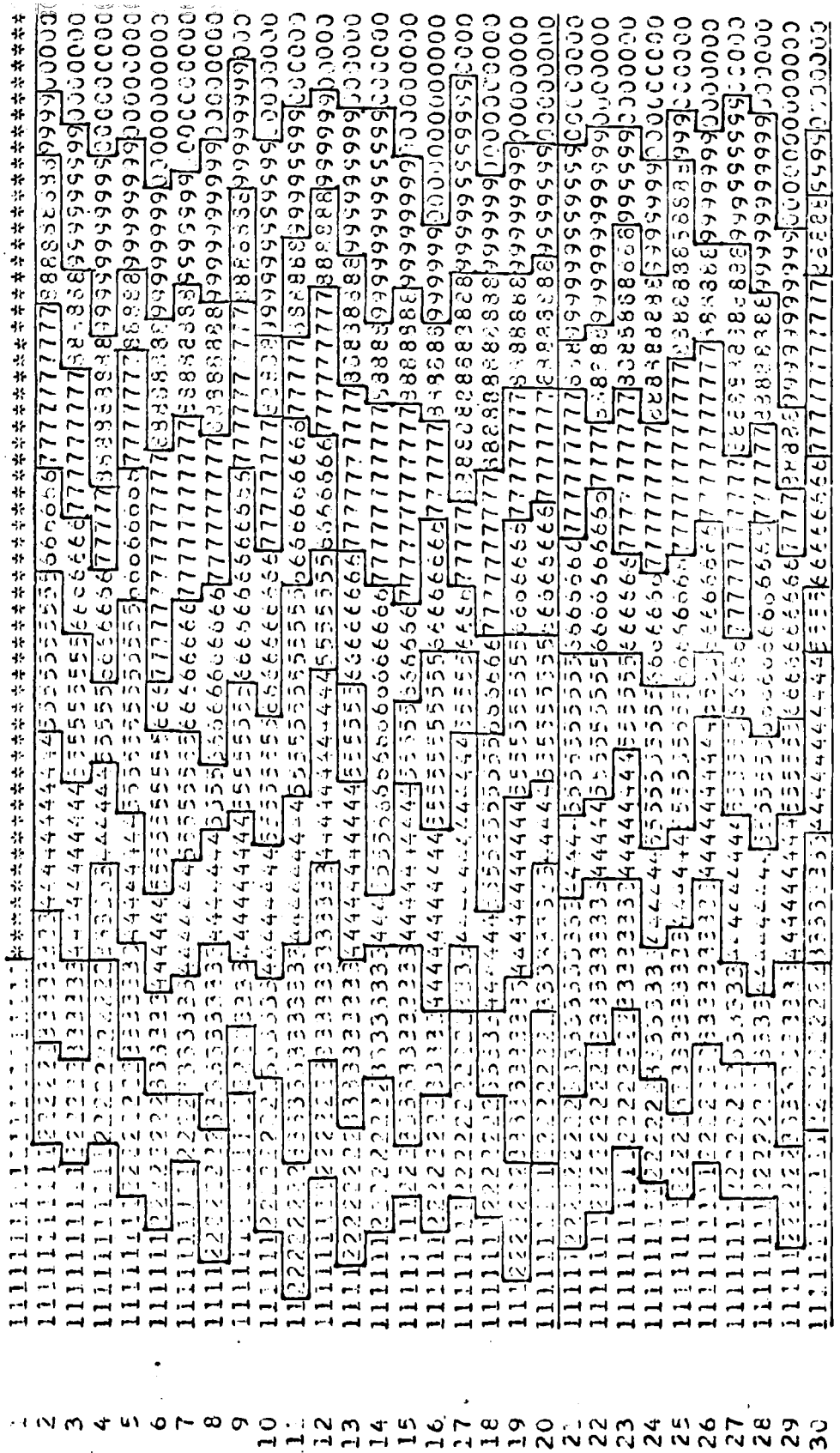
Superimposed Graphs of CPU Utilisation
Against Multiprogramming Level
and
Number of Drum Reads Against Multiprogramming Level

Horning's algorithm - state diagram

Figure 5.1

the level at which we would wish to run the system. However, this level is dependent upon the workload at any time. We are therefore interested in a mechanism which will vary the effective level of multiprogramming so that we may obtain this optimum. The form of control will be the introduction of processes into, or the removal of processes from core. Thus when the system is in area A of figure 5.1 we would like there to be a net drift of pages from high to low priority processes, increasing the multiprogramming level, and from low to high priority processes when in area B, decreasing the level.

Due to the random page replacement policy the strategy is biased towards 'stealing' pages from the processes with the most pages, which are probably the highest priority processes. When page demand is low, area A, the queues for drum service will be short and high and low priority processes will obtain essentially equal service. A net drift of pages from high to low priority processes will result. When page demand is high, area B, the lower priority processes will be blocked from obtaining drum service due to the rapid requeueing of the service requests of the higher priority processes. The priority ordering of the drum queues will therefore bias the gain of pages to high priority processes. Low priority processes will eventually be deleted from core. As paging activity subsides the service requests of the low priority processes



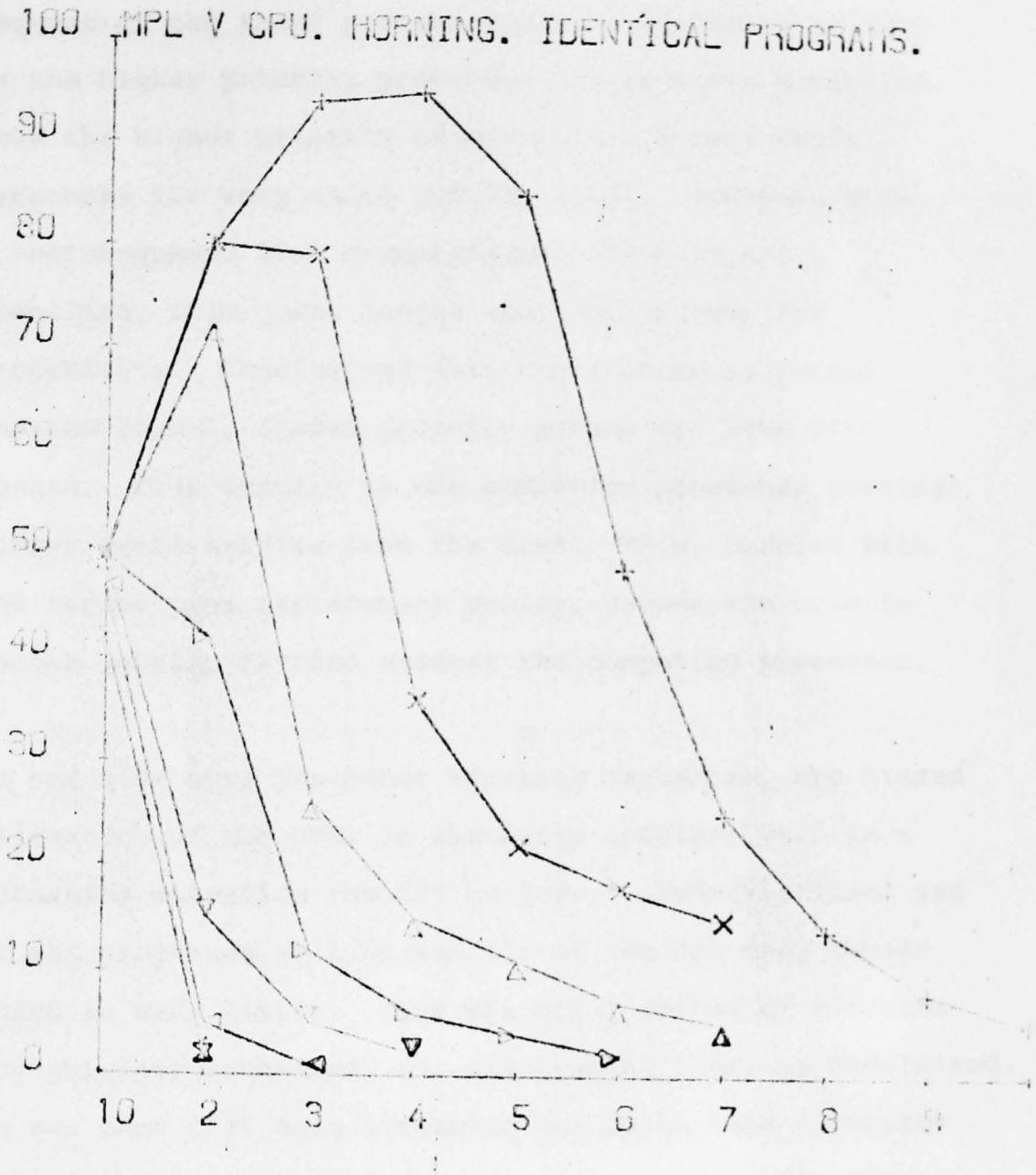
Horning's algorithm - core map for identical processes (70 page core)

Figure 5.2

will become unblocked and these processes will once more be able to obtain core.

Let us now consider the core map, figure 5.2, which was obtained for Horning's algorithm with the Multiprogramming Limit set at 10 and all processes identical, each having a 20 page parachor, as in the simulation of Wharton's algorithm. We observe that, in contradiction to our above reasoning, all of the processes quickly obtain pages in core and no process has its parachor. Also, the higher priority processes obtain no greater share of core than do those of lower priority. For instance at time 6 seconds, process 1 has 6 pages whilst process 10, the lowest priority process has 11 pages. We can explain the failure of this algorithm as follows.

The premise upon which the algorithm is based is that when drum queues are short the core is not overpartitioned. This implies that the effective level of multiprogramming may be increased. A simple example is sufficient to display that this need not be true. Suppose that the three highest priority processes have a combined parachor greater than the number of pages of core, so that thrashing would occur if avoiding action were not taken. The longest possible queue for drum service would contain just three elements which would be insufficient to block the service of the third process. (Even if this were sufficient the



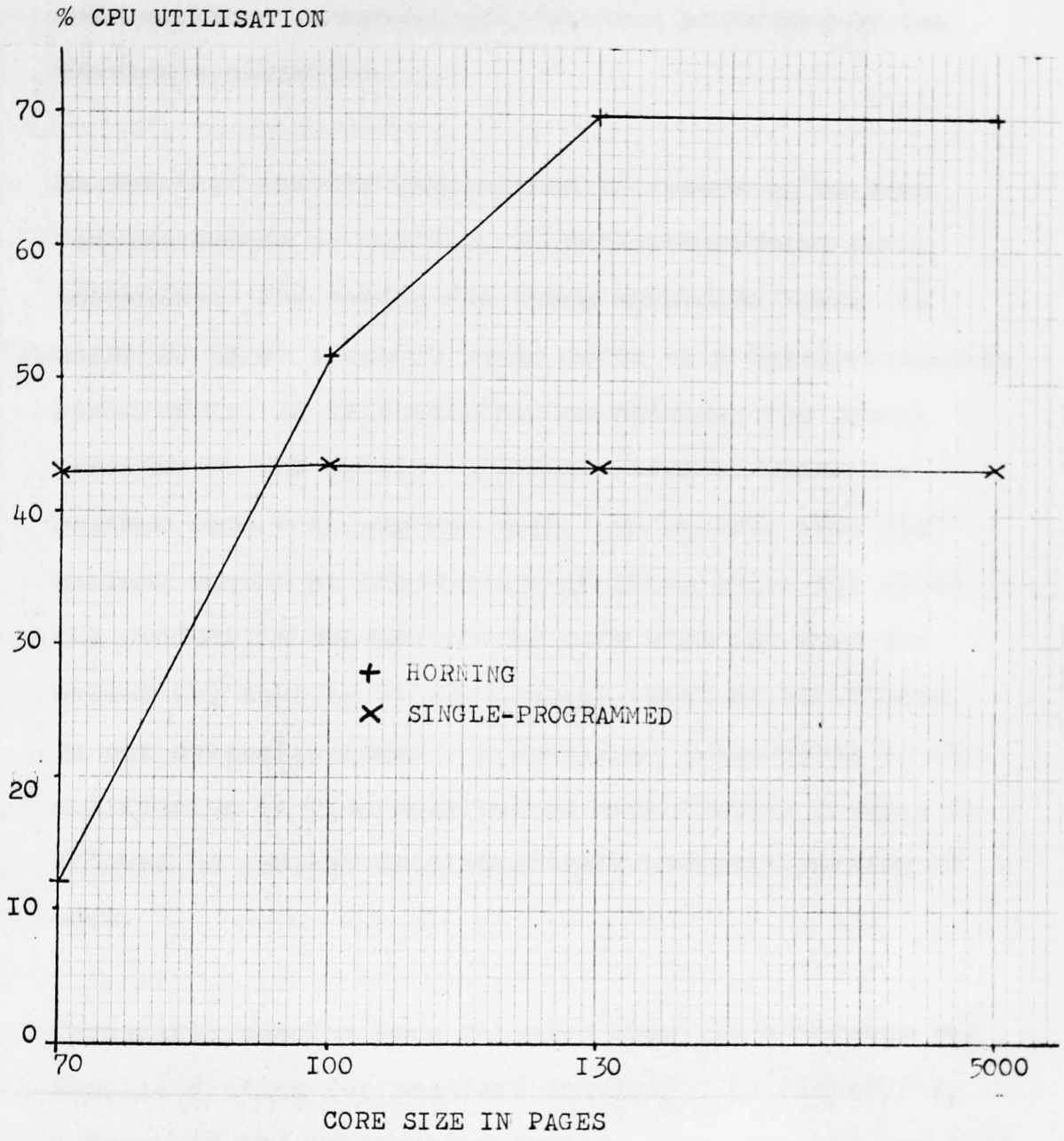
Horning's algorithm - CPU utilisation against Multiprogramming Limit for identical processes

Figure 5.3

request of the third process would be unblocked as soon as the higher priority processes obtain their parachors. Thus the higher priority processes would have their parachors for very short periods only). However, with a sector-queued drum organisation, which we are modelling, this queue length would be of very low probability. Thus we see that the problem is caused because lightly loaded priority queues act like FIFO queues. This results in the competing processes getting almost equal service from the drum. This, coupled with the random page replacement policy, causes the core to become equally divided amongst the competing processes.

We see also that the other blocking mechanism, the biased allocation of the CPU, is similarly ineffective. In a thrashing situation the CPU is grossly under-utilised and so all processes will obtain all of the CPU they demand which is very little. Thus the other method of blocking low priority processes, not dispatching them, is undermined. We see then that once thrashing has begun this algorithm will cause further degradation. Similarly we see that if the CPU is not fully utilised, the common situation in all operating systems, the reaction of this algorithm is to introduce further processes to utilise it. Such a policy must eventually lead to thrashing.

Let us now consider the graph of CPU utilisation against Multiprogramming Limit, figure 5.3, obtained using the



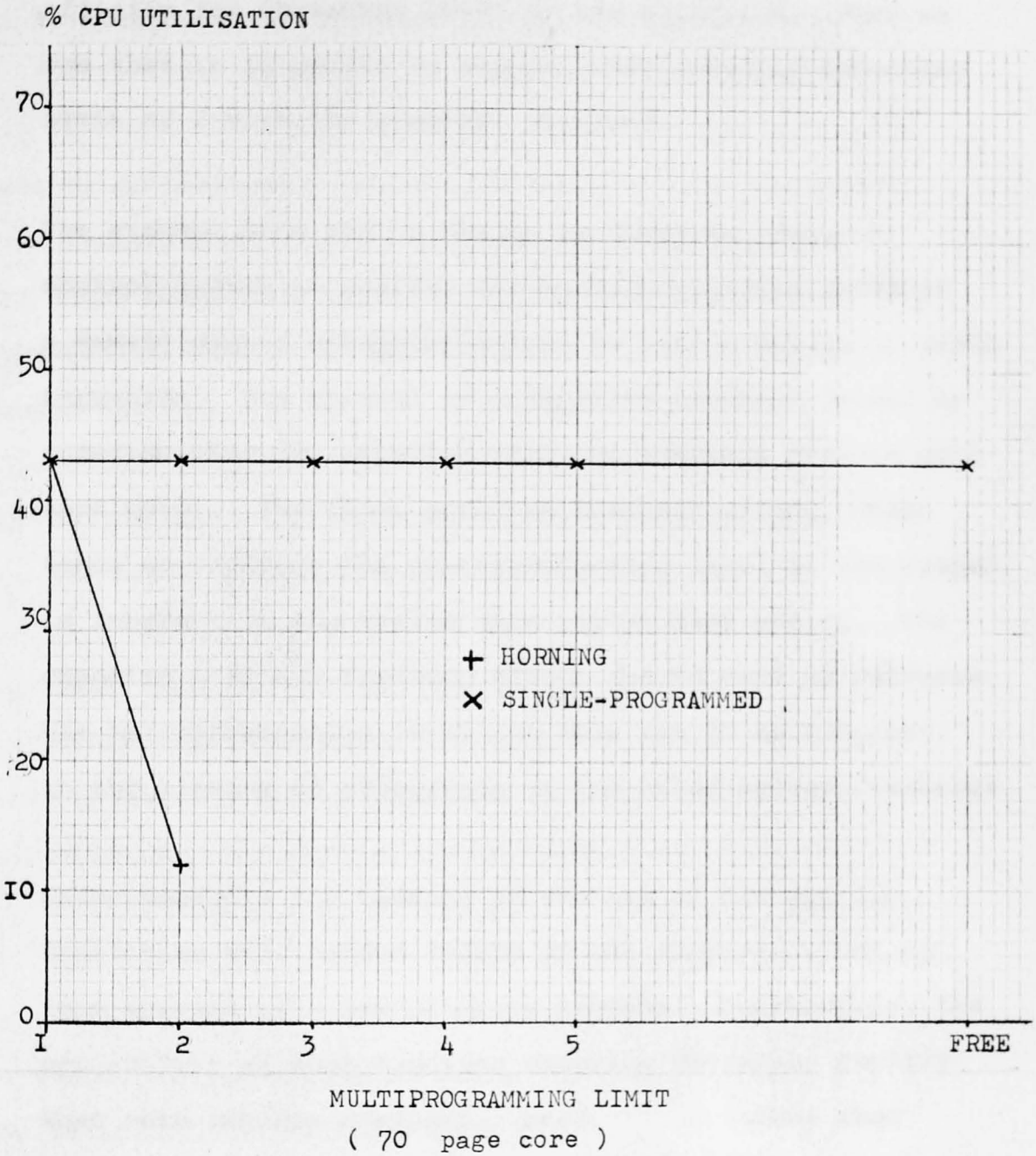
Horning's algorithm - CPU utilisation against core size for the standard mix

Figure 5.4

same workload composed of identical processes as for Wharton's algorithm.

We see that the CPU utilisation increases up to some maximum before decreasing, as Multiprogramming Limit increases. The maxima for those workloads where the parachor is at least 50 pages occur at a Multiprogramming Limit of 1. It is sufficient to multiply the common parachor by the Multiprogramming level at which the maximum occurs to explain this. We observe that the maximum occurs at the Multiprogramming Limit for which the product is closest to the core size but does not exceed it, that is at the highest level at which core is not overpartitioned. A continual degradation of CPU utilisation is then observed as each further process is allowed to contend causing greater overpartitioning of core.

Comparable results were obtained when the algorithm was simulated using our standard workload. In figure 5.4, a graph of CPU utilisation against core size, we see that thrashing has occured when only a limited core was available. These results for Horning's algorithm also demonstrate that given enough core even the most ill-conceived algorithms can be made to perform satisfactorily! The graph of CPU utilisation against Multiprogramming Limit from Horning's algorithm, figure 5.5, again displays the



Horning's algorithm - CPU utilisation against Multiprogramming Limit for the standard mix

Figure 5.5

affinity for thrashing shown by the algorithm. Here we see that it is unable to support even a Multiprogramming Limit of 2 with the standard workload.

The attempt here was to design an inherent feedback control system to control the multiprogramming level of a demand paging operating system in such a way as to avoid thrashing. The control is a negative feedback formed by superimposing two opposing positive feedback effects upon each other. The first positive feedback effect, which tends to increase the multiprogramming level of the system, is inherent in the random page replacement policy. The opposing positive feedback effect should tend to decrease the multiprogramming level and this was to be inherent in the biasing of allocation of the other system resources.

Unfortunately, the biasing of the use of CPU and I/O facilities will seldom result in the complete blocking from service of a low priority process. Furthermore, the probability of such blocking occuring decreases rapidly when core becomes overpartitioned. Thus rather than this second positive feedback becoming more assertive as the system deviates from the desired operation it becomes weaker. The tendancy to increase multiprogramming level therefore dominates and so the system will quickly deteriorate into a thrashing situation whenever core becomes overpartitioned.

Horning's algorithm was also simulated with the sector-queued drum organisation replaced by a single queue for drum service. This should increase the probability of low priority processes being blocked since the average drum queue length will be greater than with sector-queuing. The results obtained however showed precisely the same defects in Horning's algorithm as when sector-queuing was used.

An important by-product of the work on this algorithm was to increase at the outset our already keen awareness of the ease with which one can fall into the trap of developing an algorithm whose behaviour cannot in practice be successfully predicted. The reasoning used to justify the Horning algorithm found ready acceptance amongst a number of interested parties until experiment showed how inadequate this reasoning was. Yet the typical operating system contains many algorithms whose behaviour is far more impenetrable than this comparatively simple core allocation algorithm!

5.3 Randell's Algorithm

We have seen in Wharton's algorithm that the biasing of allocation of system resources is a powerful tool in avoiding the problems of excessive demand for a resource. This is also shown by the following algorithm proposed by B Randell. Here, by addition of a simple priority rule to Horning's algorithm a synthesis of Wharton's and Horning's algorithms is created which achieves an improvement upon Wharton's algorithm where Horning's algorithm failed.

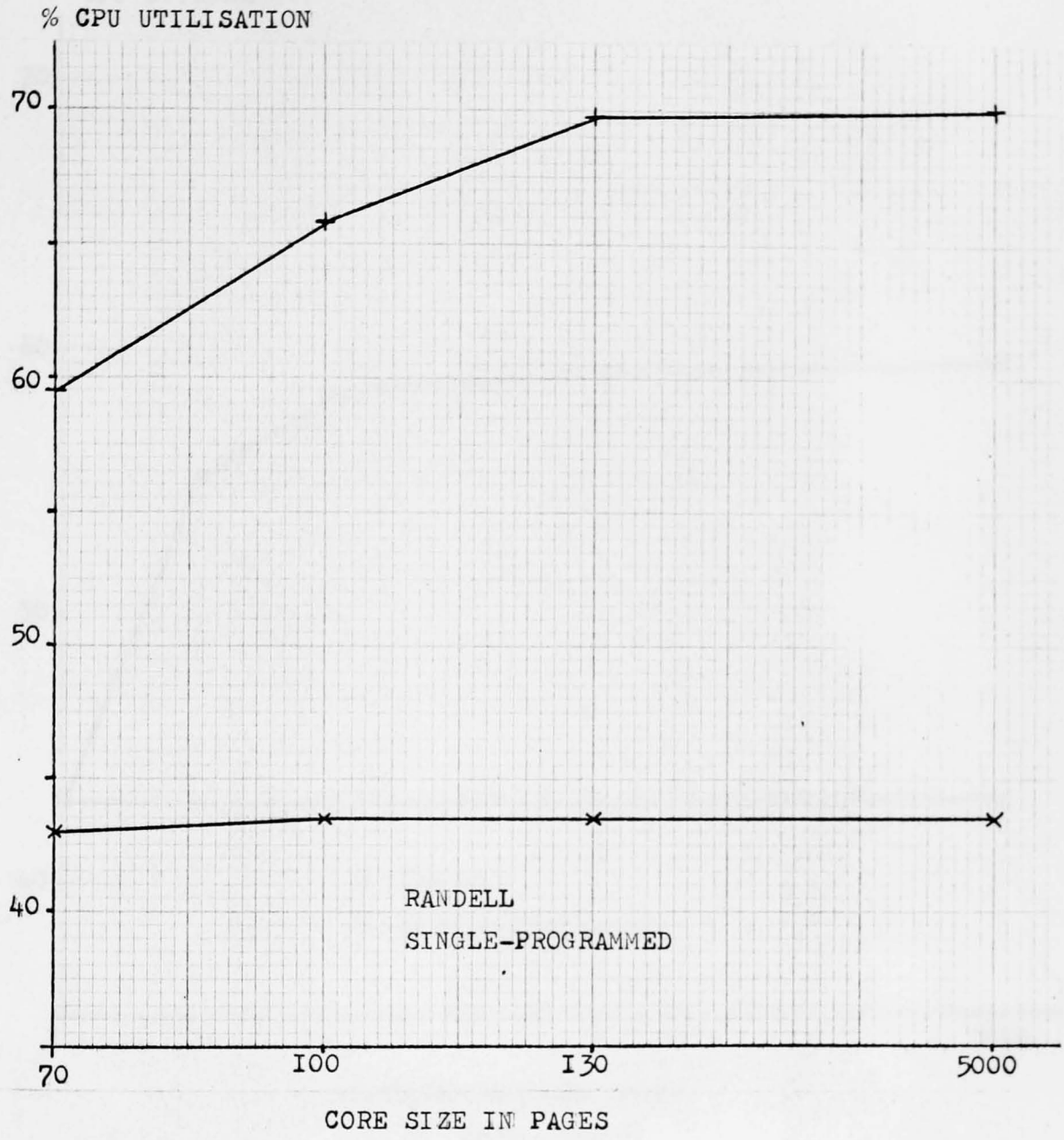
The algorithm is as stated for the Horning algorithm with the following modification to the page replacement strategy. When a page replacement is required a page is chosen from the process requiring a free page of core or from a process of lower priority . The process is chosen at random from amongst this set, each process having a probability of being chosen proportional to the number of pages of core it occupies.

Two points may be noted about this algorithm which ensure its stability. First, the top priority process never decreases the amount of core it occupies, that number either staying the same or increasing when that process requests a page replacement. Secondly, the number of pages of core occupied by the n highest priority

processes does not decrease unless one of these processes terminates execution. Thus high memory demands by low priority processes cannot affect the progress of higher priority processes. This means that high memory demands by high priority processes will eventually be met at the expense of lower priority processes but not as rapidly as by Wharton's algorithm.

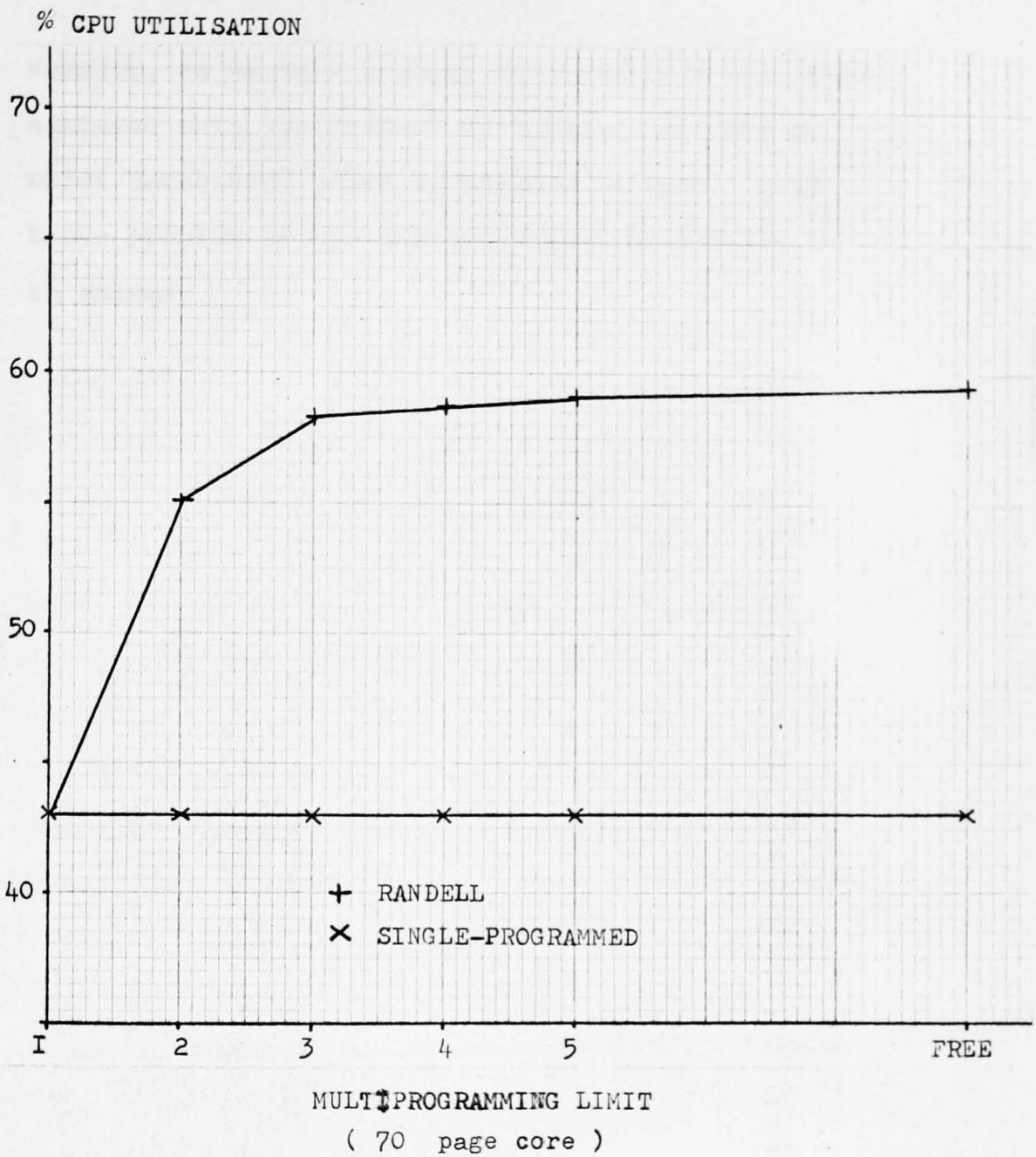
The priority constraints placed upon the random page replacement policy slows the rate at which high priority processes increase their memory allocations. In Wharton's algorithm the highest priority process has probability 1 of increasing its core allocation each time it causes a page demand. In Randell's algorithm if that process occupies x pages of core then the probability that it will increase its core allocation is (C-x)/C, where C is the core size.

We would expect then that Randell's algorithm would give improved performance over Wharton's algorithm since the rate at which pages no longer belonging to processes' current localities will accumulate in core is reduced. This expectation is confirmed by simulations of the algorithms. Figures 5.6 and 5.7, which are graphs of CPU utilisation against core size and Multiprogramming Limit obtained using the standard mix, both show the expected improvement.



Randell's algorithm - CPU utilisation against core size for the standard mix

Figure 5.6



Randell's algorithm - CPU utilisation against Multiprogramming Limit for the standard mix

Figure 5.7

However, as we have pointed out Randell's algorithm achieves this improvement by slowing the rate at which unrequired pages accumulate in core. This basic problem of the Wharton algorithm remains to be solved.

5.4 Lynch's Algorithm

The algorithm of Wharton (section 4.2) may be said to avoid thrashing by erring grossly on the side of safety when estimating a process's locality of reference. It does this by assuming that every in-core page belonging to that process belongs to the current locality of reference. This is wasteful of core which we are assuming to be a scarce resource.

A simple way to improve Wharton's algorithm would be to periodically rotate the priorities of the process, for as we have noted the only situation in which a process can have pages replaced is if it becomes the lowest priority process. Such a solution is not totally acceptable for the mechanism by which core is wasted remains, only the scale of the wastage has been reduced as it was by Randell's algorithm. This solution also raises problems of stability of the feedback control of the operating system. However, this topic is more properly discussed in the context of scheduling and will be reviewed in Chapter 6.

What is required is a modification of Wharton's algorithm which is able to estimate whether or not a process has more than its current locality of reference in core. Wharton's algorithm employs memory demand as its control

variable when estimating whether a process has sufficient core for its current locality or not. It would be natural therefore to attempt to employ the same control variable when deciding if a process occupied a greater amount of core than required by its parachor. A successful modification of this form was proposed by W C Lynch.

The proposal is to couple a 'drain' with the Wharton algorithm. The 'drain' is an autonomous process activated periodically. This process marks as available for replacement one page of core, chosen by some suitable strategy, from the process which was occupying the CPU at the time the drain was activated (if such a process exists).

The drain has the useful practical advantage of providing an inventory of free pages. This reduces the number of times the page replacement strategy (Wharton's part of the algorithm) must be invoked to force a page out of core. This means that the writing of copies of the 'drained' pages to the drum is not urgent and can be scheduled when convenient.

The reasoning behind Lynch's modification is that if a process has core in excess of its parachor then the drain will gain pages from the process. If it were to drain a page which belonged to the current locality of the process

which had been utilising the CPU, that process would upon resumption demand the page. In particular if the process was of the highest priority then we would expect that the page would be re-acquired in half of a logical drum revolution on average. Thus if we were to consistently drain from the current locality of the highest priority process we would impair that process's performance by a maximum of P%, where P = ½ x period of a logical drum revolution x 100/drain period. We would hope to recoup this loss by improved performance of lower priority processes. We would also expect that the processes most likely to be utilising the CPU would be those which had at least their parachor. Thus there should be a high probability that pages are being drained from the correct set of processes.

Lynch's algorithm has essentially the same feedback controller structure as Horning's algorithm, in that it is composed of two opposing positive feedback effects. One, that attributable to the Wharton algorithm, tends to decrease multiprogramming level. The other, caused by the Lynch drain, tends to increase the multiprogramming level by removing from core pages not in the current locality of any process, thus making core available to further processes. These two opposing effects set up a dynamic equilibrium with each other. The position of balance depends upon the rate at which the draining process acts and also upon the memory demands of the

processes contending for core.

This dynamic equilibrium is reinforced further since the Lynch drain tends to increase memory demand by allowing further processes to contend and by restricting the core which each process can obtain. On the other hand the Wharton component tends to decrease memory demand by assigning more core to each process in response to its demands.

We see then that each component of this algorithm endeavours to create a situation which weakens its own effect whilst intensifying the effect of the opposing component. This leads to a very strong negative feedback control upon multiprogramming level and this is in the form of an inherent feedback control.

The property which we require most of this control is that it should be stable. That is under conditions of heavy load, thrashing should not be allowed to occur. To observe that Lynch's algorithm has this property we need only observe that under conditions of high memory demand Wharton's component dominates. We have already established that Wharton'a algorithm avoids thrashing and so the Lynch algorithm must be stable.

Having established that we have an anti-thrashing control we are interested in how quickly it will settle to its

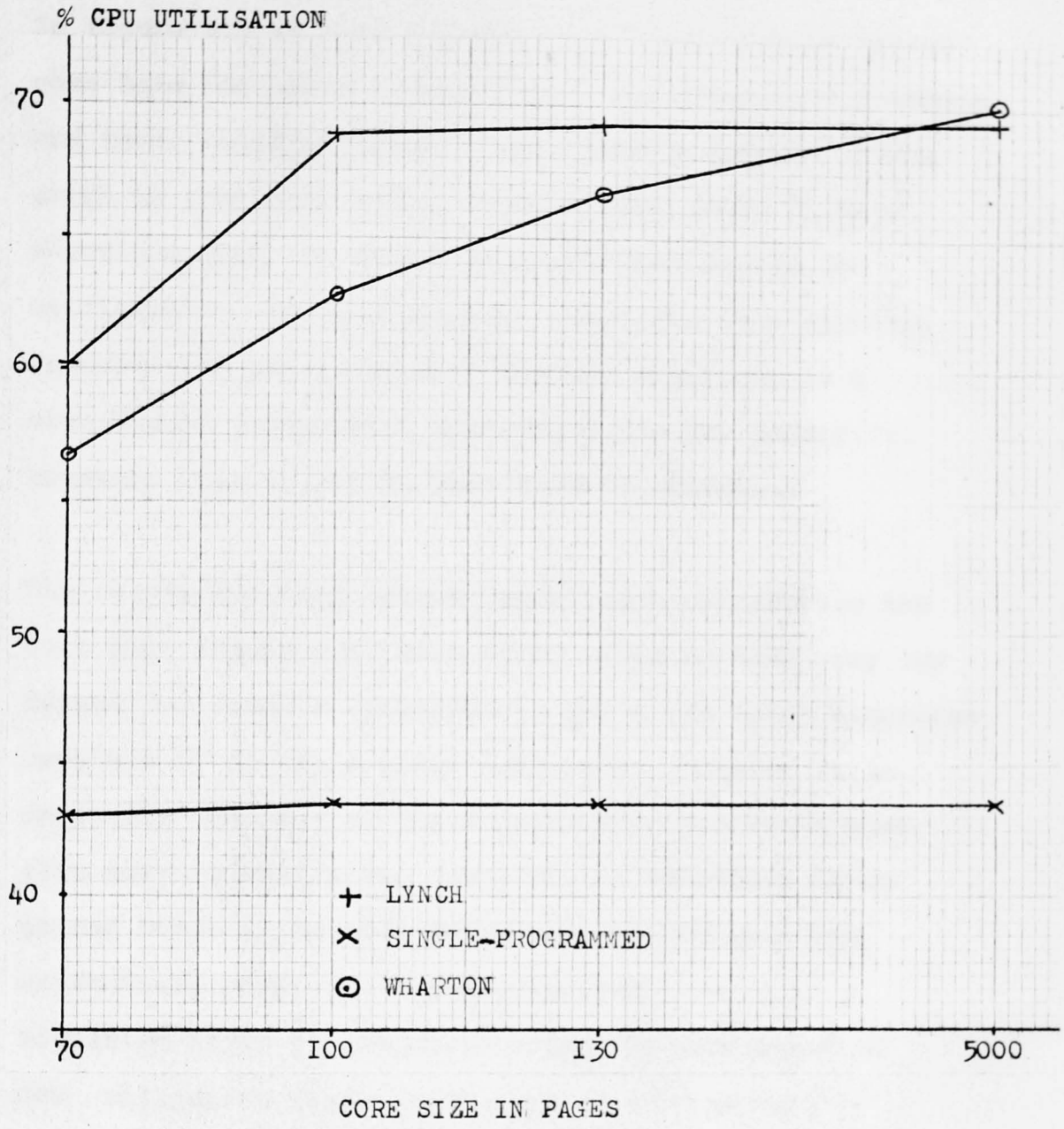
position of equilibrium. The settling time of the algorithm is to some extent dependent upon the rate at which pages can be loaded, by demand paging, from the drum. Consider the core initially empty and a number of processes loading into it by demand paging. The allocation will stabilise when as many of the high priority processes as possible have loaded their current localities of reference. These are essentially loaded in parallel and so settling time would be the time to load the largest of these.

The settling time in response to a disturbance such as a sudden change of locality by a process will be equal to the time taken by the drain to remove the old locality. One would expect settling to occur rather more slowly in this case since it is a feature of the strategy that pages may be added on demand more quickly than they may be removed by the drain.

Let us now consider part of a core map from a simulation of Lynch's algorithm, figure 5.8. The standard workload was used and the drain process was activated every 70 milliseconds which is equal to two logical drum revolutions in the simulation model. The two processes represented in the core map are process 42, code 2, with a parachor of 15, and process 49, code 9, with a parachor of 20. We see that Lynch's algorithm limits both processes to enough core to contain the current localities.

Lynch's algorithm - core map for the standard mix

Figure 5.8



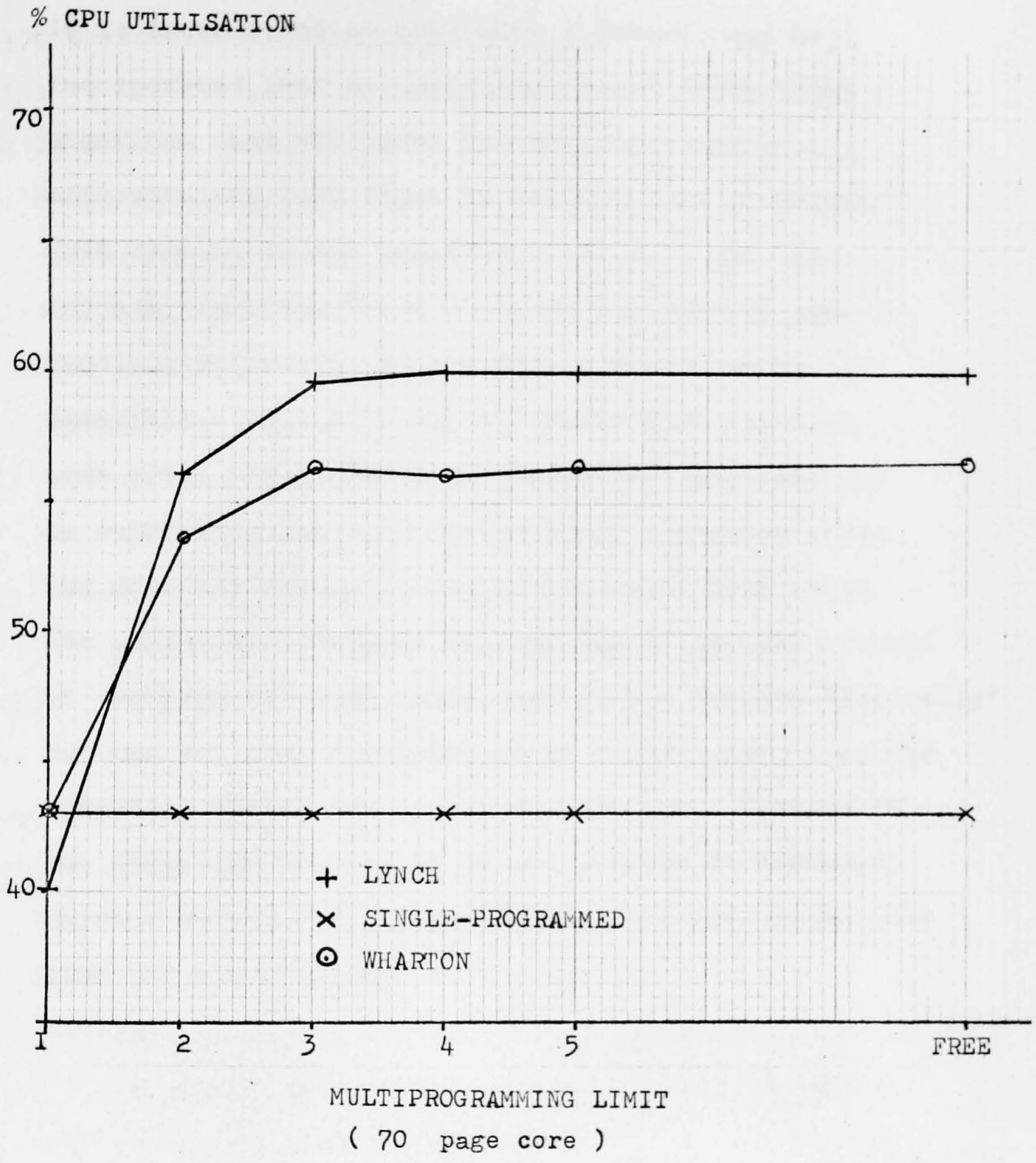
Lynch's algorithm - CPU utilisation against core size for the standard mix

Figure 5.9

In figure 5.9 we have a graph of CPU utilisation versus core size for Lynch's algorithm. The results of interest are those labelled LYNCH. The striking feature of the graph is that even with a core of 5000 pages Lynch's algorithm does not attain the maximum possible CPU utilisation. As we predicted, when core is a limiting resource the performance of Lynch's algorithm is a significant improvement upon Wharton's and Randell's. However, this is not so when core is abundant.

The reason for this becomes plain upon considering the core map, figure 5.8, once more. Here we have very low demand for core, a situation to which the Lynch algorithm responds by draining pages from core. However it is obviously wasteful to drain pages from processes when free core is available. There is no advantage to be gained since no further processes require core and unnecessary page faults are generated. It is this situation which is responsible for the depression of CPU utilisation with a core size of 5000 pages.

The graph of CPU utilisation against Multiprogramming Limit, figure 5.10, shows as we would expect that the Lynch algorithm has the desirable property that CPU utilisation increases monotonically with Multiprogramming Limit. Again we see that the Lynch modification gives a substantial performance improvement over the basic Wharton algorithm.



Lynch's algorithm - CPU utilisation against Multiprogramming Limit for the standard mix

Figure 5.10

It is interesting to note that a feature can be incorporated into an operating system using Lynch's algorithm that will give further increases in effectiveness than shown in the simulations (where this feature is not modelled). This is the 'page reclaim' function which is often included in page handling software. This function makes use of the fact that a page will not be overwritten for some time after the decision to remove it. The time may be especially extended in the Lynch algorithm since the page may remain in the inventory of free pages for some time before it is required to be overwritten. If the page has been taken away from a process 'logically' but has not been scheduled to be overwritten, then the operating system can 'reclaim' the page. That is it can logically restore it to the process if demanded, thereby saving a page transfer and the associated idle time for the process.

5.5 The Lynch-Alderson Algorithm

The comments concerning the unnecessary draining of pages in low memory demand situations suggest a further refinement of Lynch's algorithm. As we have seen the Lynch drain provides an inventory of pages which are available for replacement. Free pages of course are included in this inventory. The next step is to place some limit upon the size of this inventory.

This may be achieved by setting a 'threshold' size for the inventory. The draining process takes action only if the current inventory size is less than the threshold value. This addition to the Lynch algorithm we have called the Lynch-Alderson algorithm.

The setting of a threshold value causing the drain to be switched on and off allows the draining rate to alter with memory demand. When memory demand is low the drain switches off since there is no need to utilise memory efficiently in such a situation. When memory demand increases the drain switches on removing pages not in the current localities of contending processes in an attempt to satisfy the increased demand. (Again however, there is a maximum extent to which any processes may be retarded by the drain). If the increased memory availability provided by the drain proves insufficient the Wharton component will ensure memory demand is reduced by removing

low priority processes from core.

Again we can show the stability of the algorithm even though both memory demand and draining rate may vary. This is possible because stability is only a problem when overload conditions exist. Where such high memory demand occurs the algorithm essentially reverts to the Lynch algorithm the stability of which we have already discussed.

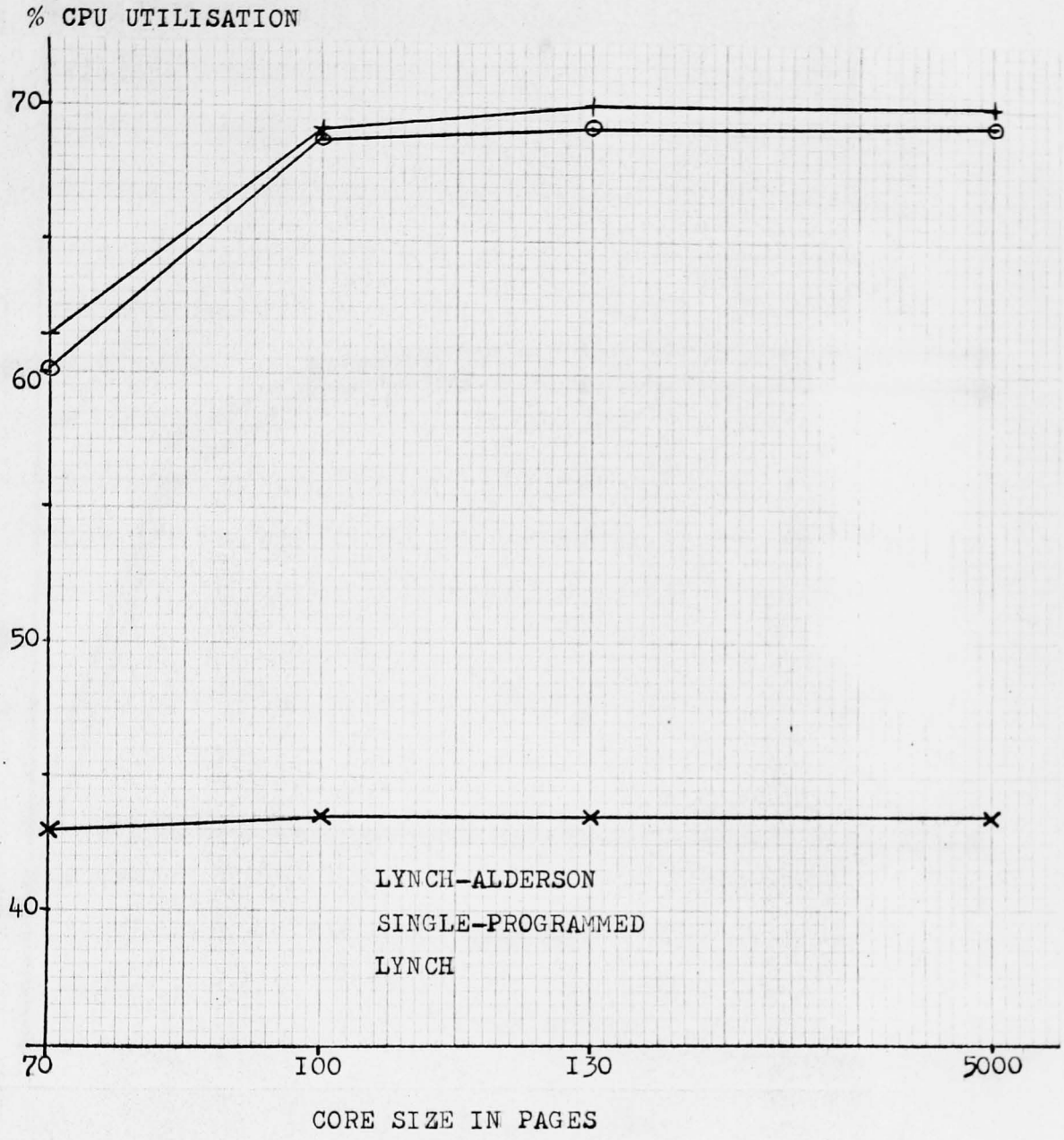
The settling time of the Lynch-Alderson algorithm in response to a stimulus, such as a sudden surge in memory demand caused by the introduction of a further process into the mix, is similar to that of Lynch's algorithm since under such conditions the Lynch-Alderson reverts to the Lynch algorithm. It is possible that the settling time could be slightly greater than that for Lynch's algorithm. This is because there may have been an accumulation of pages not belonging to any current localities of reference which may need to be deleted by the drain. However, one would expect a substantially higher drain rate to be tolerable (and, perhaps, even appropriate) in Lynch-Alderson than in Lynch, since the inventory threshold tends to limit overdraining. The higher rate would reduce settling time considerably.

Lynch-Alderson algorithm - core map for the standard mix

Figure 5.11

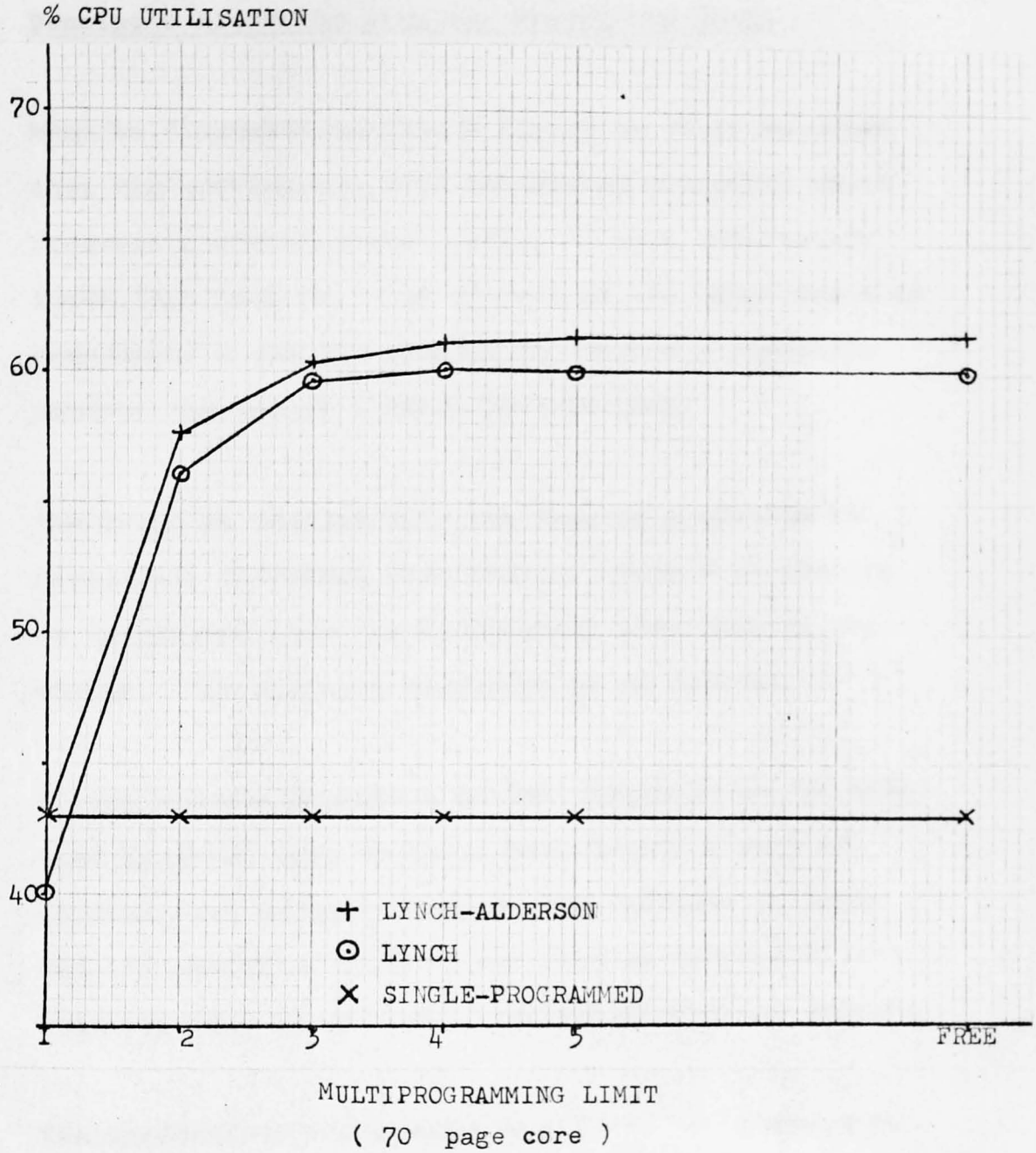
Let us now consider, figure 5.11, which is a core map from a simulation of the Lynch-Alderson algorithm using the standard workload and the threshold set at 1. From time 451 seconds to time 474 seconds we again have processes 42 and 49 in contention for core, as in figure 5.8, the core map for Lynch's algorithm. We see that the two processes are no longer restricted to their parachors as they were by Lynch's algorithm. We also note that the algorithm is maintaining a single page inventory as indicated by the '*' which frequently appears in the final column.

The graphs of CPU utilisation against core size and Multiprogramming Limit, figures 5.12 and 5.13 respectively, show as we would expect that the Lynch-Alderson algorithm gives an improvement upon Lynch's algorithm. In particular we note that with the 5000 page core the Lynch-Alderson algorithm obtains the maximum attainable CPU utilisation.



Lynch-Alderson algorithm - CPU utilisation against core size for the standard mix

Figure 5.12



Lynch-Alderson algorithm - CPU utilisation against Multiprogramming Limit for the standard mix

Figure 5.13

5.6 Denning's Algorithm with the Predictive Drain

When we discussed Denning's algorithm (4.3) we noted that the working set size estimation procedure could frequently overestimate leading to core utilisation lower than need be. The success of the Lynch drain in overcoming a similar problem in Wharton's algorithm prompts the use of a drain process here.

The drain we coupled with the Denning algorithm is distinctly different from that of Lynch's in that it is integrated into the dispatching algorithm of the system. The draining algorithm is as follows.

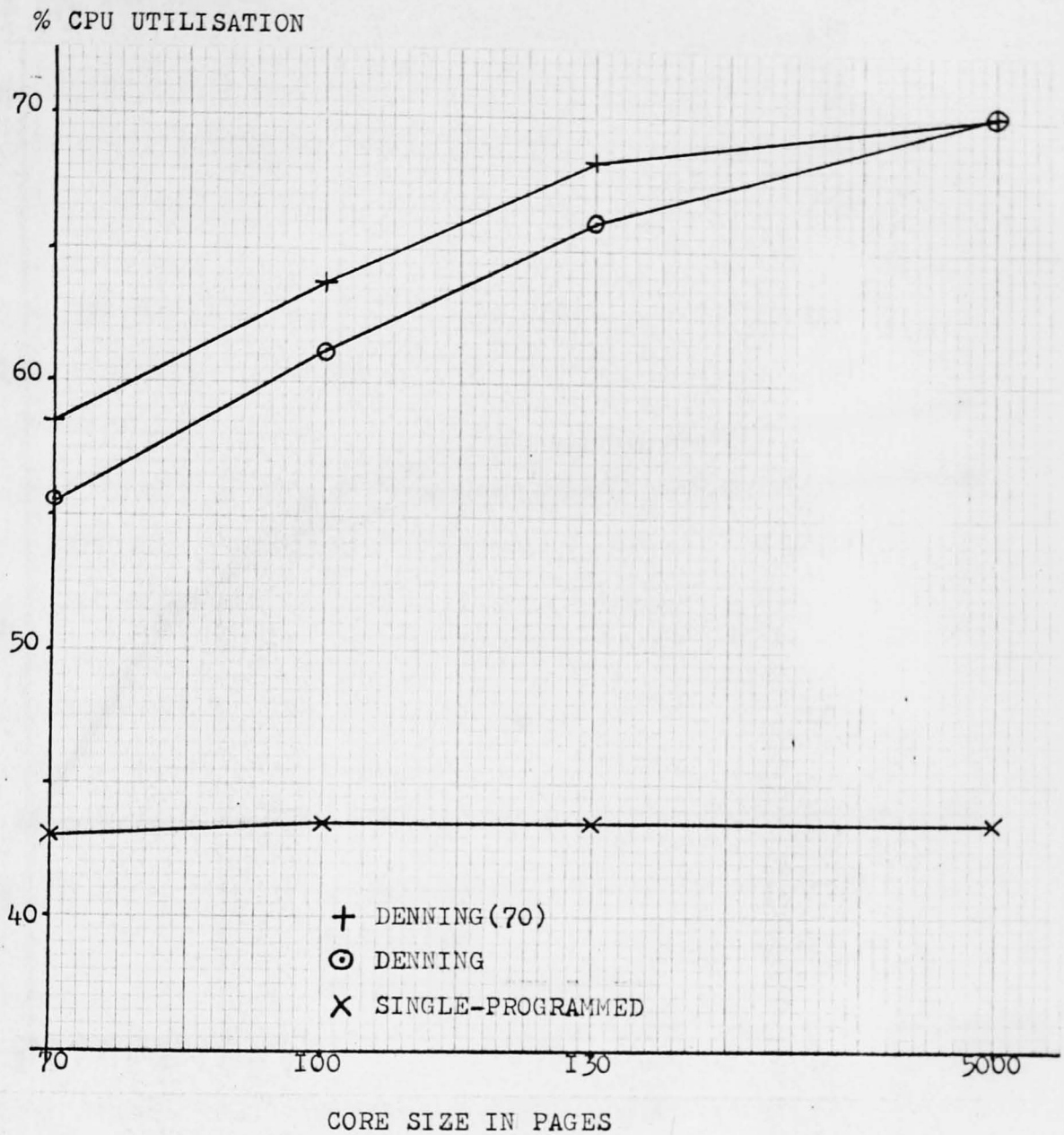
If any process exceeds a preset threshold of CPU time used since it last caused a page fault, a page is deleted from amongst those it has resident in core and its estimated working set size is reduced by one. This strategy we call the predictive drain or delete.

The predictive delete approximates to the removal of a page from the working set if it has not been accessed in the previous interval. As with the Lynch drain this strategy is a superimposed feedback control. This control has the property that it does not affect the stability of the Denning algorithm. If any process has insufficient core then the periods between page faults for that process will reduce below the threshold and

the drain will cease to have effect for that process. Thus if core contention is high the drain ceases to be effective and the algorithm reverts to the basic Denning algorithm.

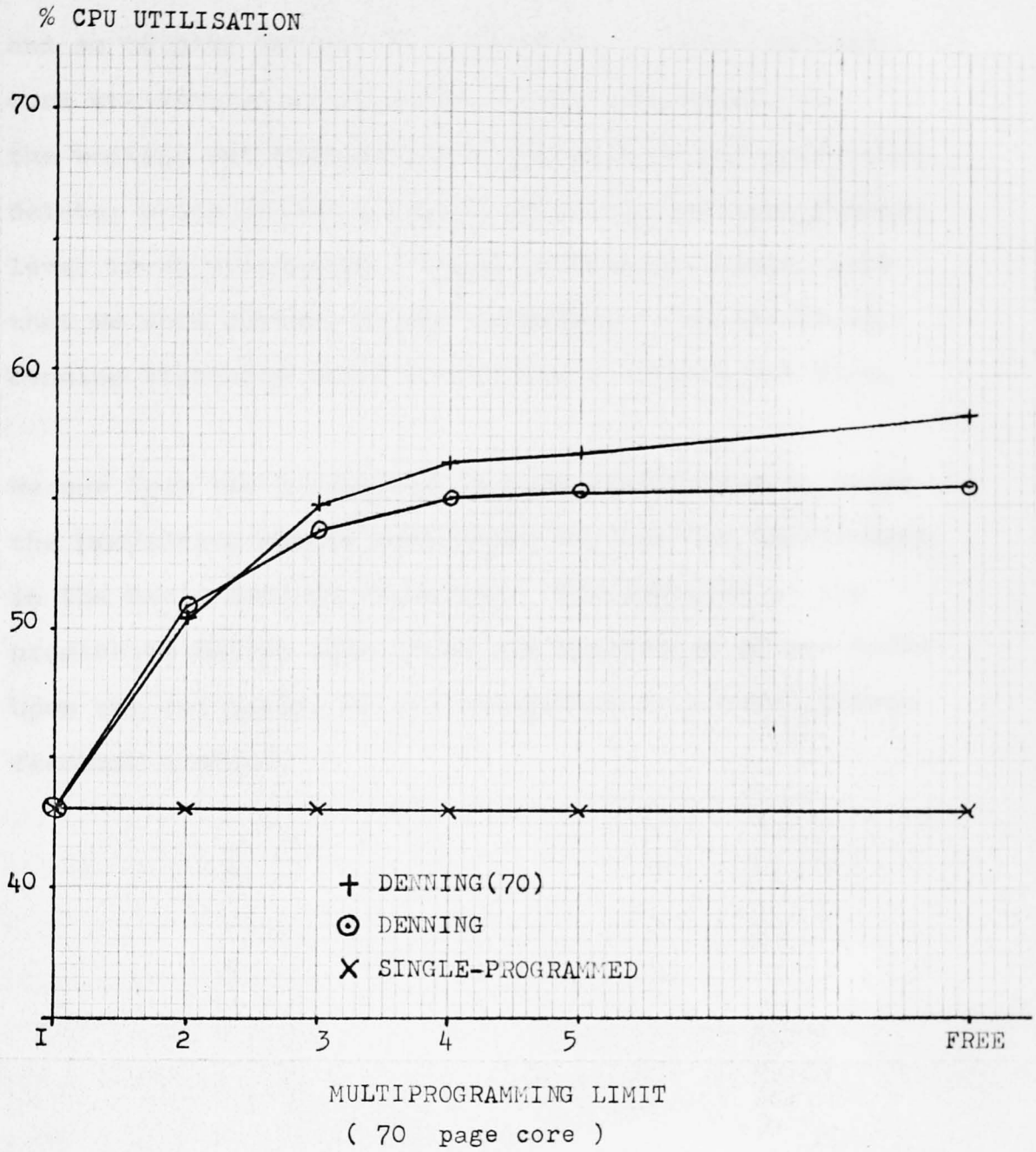
Let us now consider the simulation results for Denning's algorithm with and without the predictive delete. Figure 5.14 shows the variation of CPU utilisation with core size for the standard mix. The results are marked DENNING and DENNING(70). These represent Denning's algorithm with no predictive drain and Denning's algorithm with the predictive delete threshold set at 70 milliseconds respectively. Comparison of these shows that the Denning(70) algorithm gives a consistent improvement over the Denning algorithm. We note also that the improvement increases as the amount of core available increases (until core availability ceases to be a constraint on the system). This occurs because as the system becomes less core constricted more processes run for CPU intervals in excess of the threshold without a page fault occurring and so the predictive delete strategy becomes increasingly effective.

In figure 5.15 we present a graph of CPU utilisation against Multiprogramming Limit for a core size of 70 pages. Again we see that employing the predictive delete allows the Denning(70) algorithm to utilise core more effectively



Denning's algorithm and predictive delete - CPU utilisation against core size for the standard mix

Figure 5.14



Denning's algorithm and predictive delete - CPU utilisation against Multiprogramming Limit for the standard mix

Figure 5.15

and so to give better CPU utilisation. This improved core utilisation is a result of the improvement in the working set size estimate provided by the predictive delete, which allows a higher effective multiprogramming level to be maintained. These simulation results show that we were correct in our assessment that the basic Denning algorithm would overestimate working set size.

We see from the difference in results obtained by using the predictive delete that there is room for improvement in the basic Denning algorithm. The addition of the predictive delete also shows the dependence of the model upon the estimation of its parameters in a model driven feedback control.

5.7 Drain Processes

As we have seen from the Lynch and Lynch-Alderson algorithms, the concept of a drain process appears to be successful in promoting the effective use of core allocation. The Denning algorithm also benefits from a drain process which appeared in the form of the predictive delete strategy. In the light of this they seem deserving of further consideration. In particular we shall compare the Lynch drain and the predictive delete which provide an interesting contrast.

To recapitulate, the Lynch drain involves periodically marking as available for replacement a page of the process currently allocated the CPU. The predictive delete involves marking as available for replacement a page from each process which uses in excess of a preset limit of CPU time without generating a page demand. The basic difference between these schemes is that the predictive drain is applied separately to each process whereas Lynch's is applied to the set of processes contending for the CPU. From this view point it would appear that the predictive drain has a number of advantages.

First, the predictive delete conforms with our desire that the core allocated to any process should have a

lower bound determined only by the properties of that process. With the Lynch strategy the core allocation of a process can depend upon the extent to which other processes utilise the CPU and this may depend upon their memory demand. The reason that this does not have a disastrous effect is that there is a built in limit to the extent to which Lynch's drain can affect the progress of a process as we discussed previously.

The second advantage of the predictive drain is that since it applies in parallel to all processes we would expect it to be more effective than the Lynch drain in restricting all processes to their parachors. However, with both schemes a process must obtain CPU time before it can lose pages due to the drain. Where we employ a priority scheme, as we have done consistently, the priority bias will decrease the parallelism of the predictive drain. Therefore it is unlikely that the parallel capability of the predictive strategy bestows as great an advantage over the Lynch drain as would at first appear. However, the predictive delete does suffer a very real disadvantage.

The predictive delete requires that a limit be set upon the CPU time which a process may obtain without generating a page demand before the drain is invoked. As with the Lynch strategy it is wise to set the limit such that if

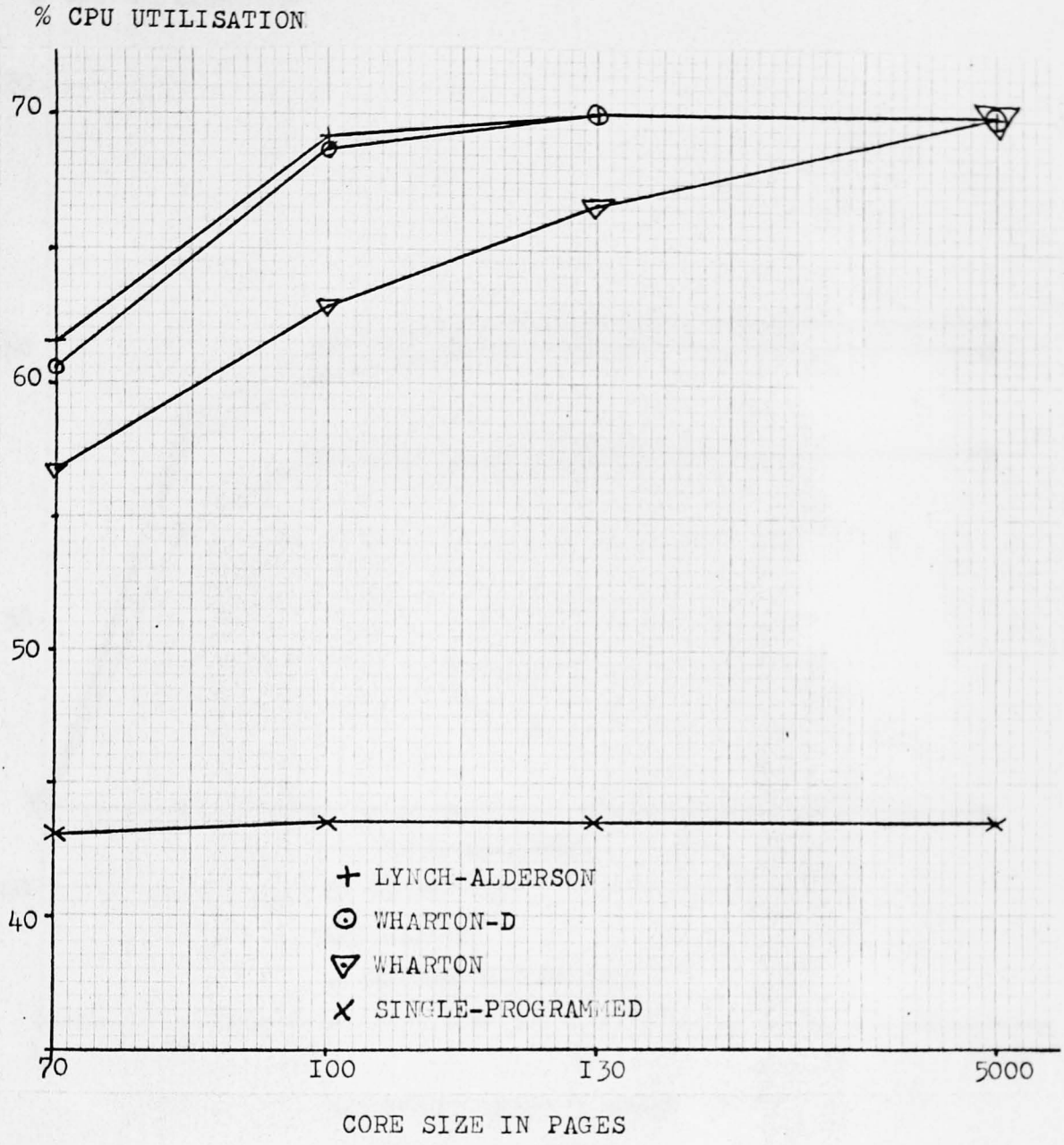
pages belonging to the current locality of a process are consistently deleted the effect upon the progress of the process is not unacceptable. However, any process which changes members of its current locality at intervals which are on average shorter than this limit will seldom be affected by the predictive drain even when occupying core in excess of its parachor. Thus the predictive drain may not have the desired effect upon all processes. In particular, we would expect that the settling time of the predictive drain in response to closely spaced multiple stimuli, such as sudden changes of locality by a number of processes would be inferior to that of the Lynch drain. This is because a sudden switch of locality is signalled by a flurry of page faults and so would not be detected by the predictive drain.

In order to obtain an indication of the importance of these factors a series of simulation experiments was undertaken to compare the two drain processes. The Wharton algorithm provides an ideal test bed for these two drain processes due to its simplicity and its antithrashing properties. We have previously discussed the combination of Wharton's algorithm with the Lynch drain which we called Lynch's algorithm. The combination of Wharton's algorithm with the predictive delete we will call the Wharton-D algorithm.

Figures 5.16 and 5.17 show graphs of CPU utilisation against core size and CPU utilisation against Multi-programming Limit respectively for the Lynch-Alderson and Wharton-D algorithms in which the inventory threshold was set at two pages, the drain processes parameter was set at two logical revolutions of the drum, and the standard workload was used. The relevant results are those marked LYNCH-ALDERSON and WHARTON-D.

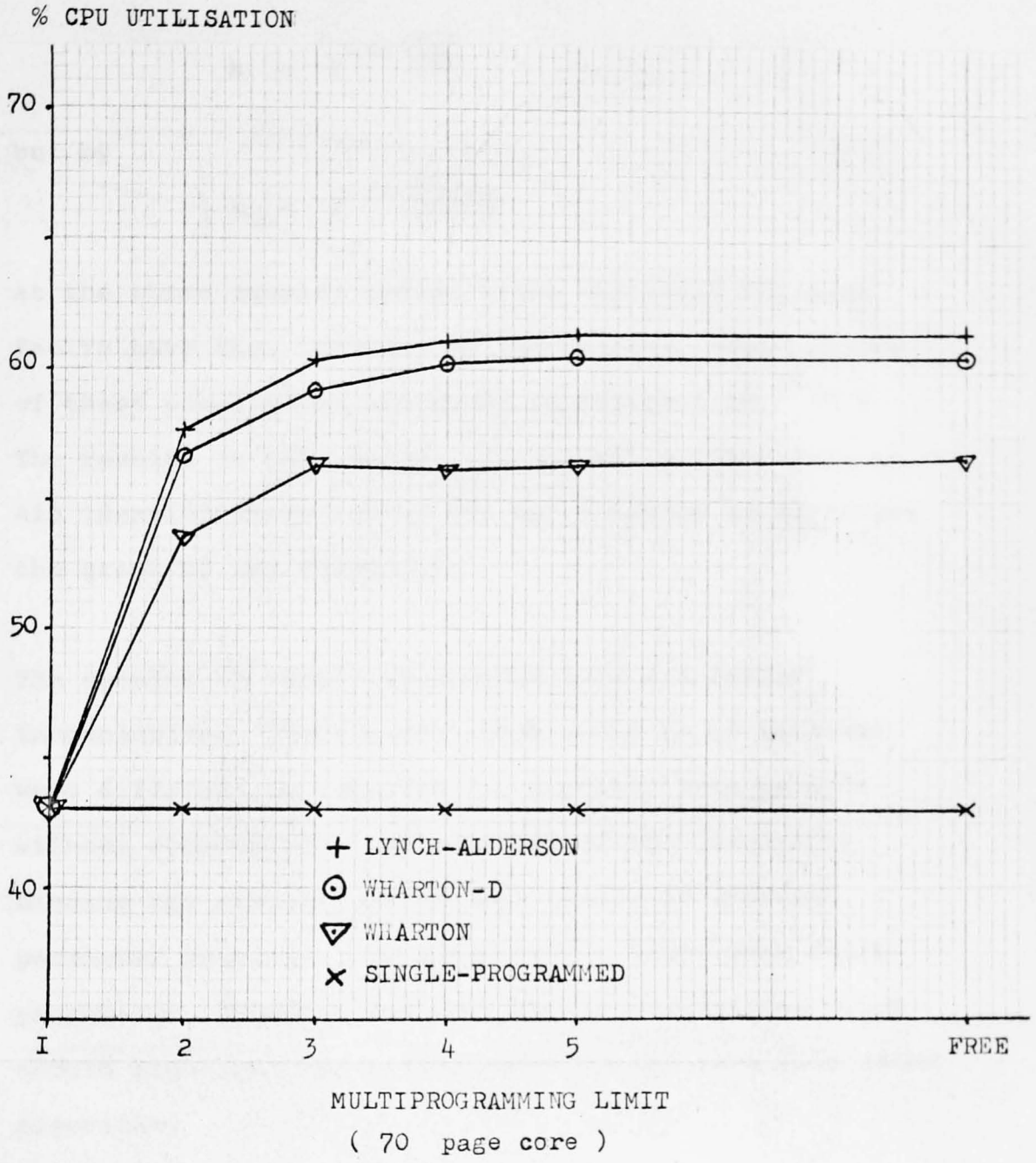
Results for the Wharton-D algorithm without the inventory threshold (not shown), are very similar to those for Lynch's algorithm.

To observe the way in which the two drain processes deal with sudden change of locality of reference the normal page fault probability function, which we term the DRIFTING function, was replaced by a 'PEAKING' function. The PEAKING function causes the locality of reference of a process to suddenly change completely three times. These sudden changes occur at times spaced equally throughout the execution of the process. Each change is modelled by assuming that during the change both the old and new localities are required, that is the size of the current locality doubles for this period. The change of locality is deemed to be complete when a number of page faults equal to the size of the locality have been incurred by the process. Thus to obtain the PEAKING function we replace k of 3.5.4 by



Drain algorithms - CPU utilisation against core size for the standard mix

Figure 5.16



Drain algorithms - CPU utilisation against Multiprogramming Limit for the standard mix

Figure 5.17

$$k = 2^{-16\frac{RCP}{PCS}} \qquad \text{usually,}$$

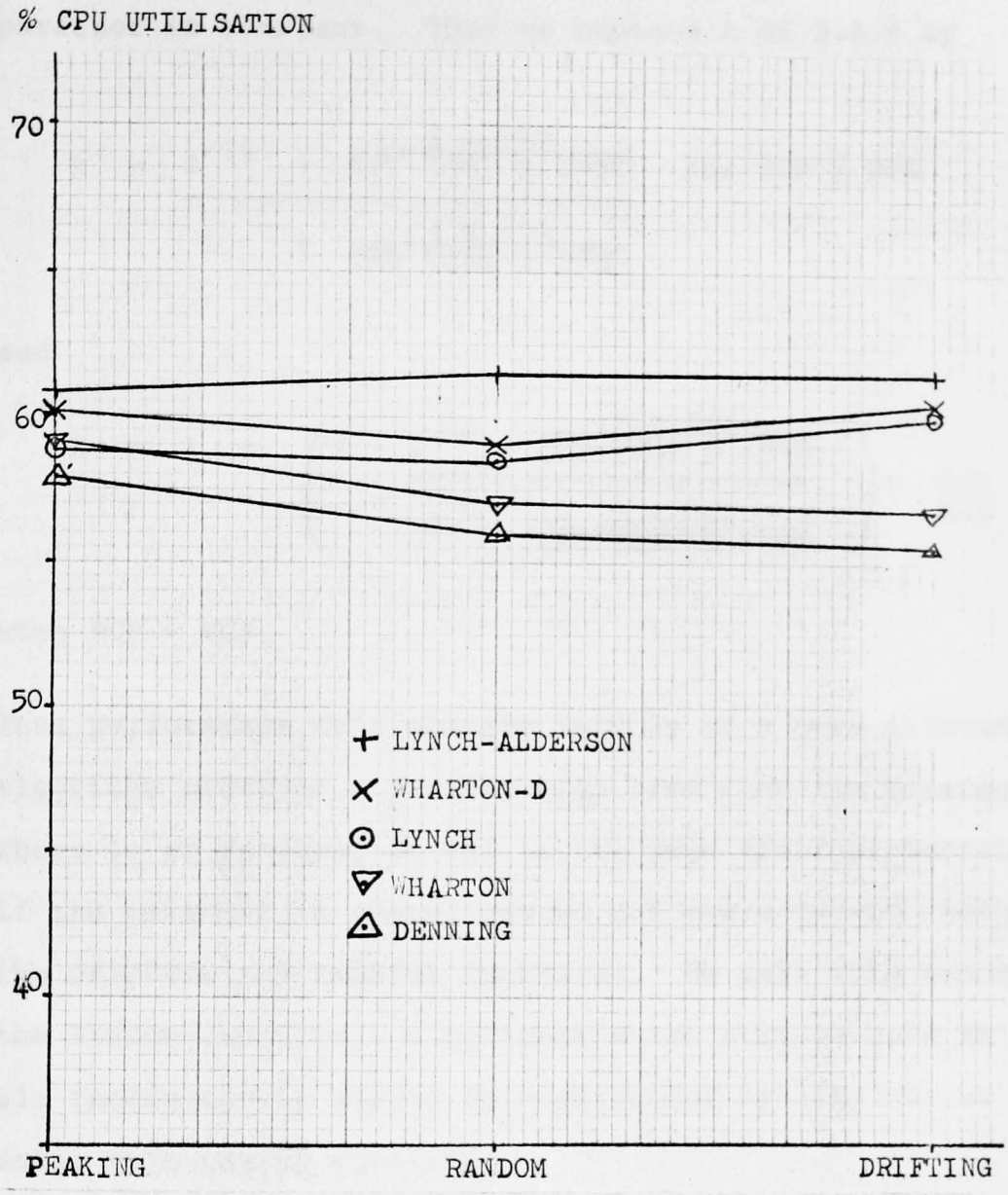
but by

$$k = 2^{-16\frac{RCP}{2xPCS}}$$

at the three equally spaced times and until PCS page faults have been incurred by the process. The results of these simulations are shown in figure 5.18. (N.B. The results in this figure are joined by lines only to aid identification - they are not intended to represent the graph of any function).

The results of these simulations were all rather inconclusive. Those cases which would be of interest were difficult to simulate for suitably long periods without compromising the validity of the results by biasing the simulations through choice of unusual parameter settings. To avoid this a third page fault probability function was used in the simulations which should highlight any inadequacies in the core allocation algorithm.

The function is such that the probability of page fault of a process which has less than its parachor is inversely proportional to the amount of core it occupies. This is equivalent to the process accessing the pages of its current locality at random. The probability of a page fault occuring when the process has at least its



(70 page core)

Various paging functions - CPU utilisations of various algorithms for the standard mix using a 70 page core

Figure 5.18

parachor is constant. Thus we replace k of 3.5.4 by

$$k = \frac{2^{-16} + \frac{3 \times PCS}{CPU\ TIME \times 1000}}{1 + \frac{3 \times PCS}{CPU\ TIME \times 1000}} \quad \text{for } RCP \geq PCS$$

and

$$k = 1 - \left(1 - \frac{2^{-16} + \frac{3 \times PCS}{CPU\ TIME \times 1000}}{1 + \frac{3 \times PCS}{CPU\ TIME \times 1000}} \right) \times \frac{RCP}{PCS}$$

when RCP < PCS.

Thus performance will degrade swiftly if a core allocation algorithm provides a process with less than its parachor. There is no decrease at all in the page fault probability if the parachor is overestimated but there is with both the DRIFTING and PEAKING functions. We call this function the RANDOM function. Algorithms which utilise core at all ineffectively should be highlighted by simulations using this paging function.

The results obtained for a number of algorithms including the Lynch-Alderson and Wharton-D algorithms are shown in figure 5.18. We see that for the PEAKING and DRIFTING functions there is little difference in the results. However, there is a difference when the RANDOM function is employed. This suggests that avoidance of the predictive drain by certain processes does occur to

some extent but that in our simulations it is not an important factor. It is only noticable when the effect is amplified by the RANDOM function. (N.B. It is not valid to compare the results of any algorithm for the three paging functions. Under the same conditions similar memory demand behaviour will not occur with the various paging functions).

To sum up we have seen from our studies that drain processes are a very useful tool in promoting the effectiveness of a core allocation algorithm which avoids thrashing. Care must be taken that the draining process can have only a limited effect so that it does not precipitate thrashing. However, we have shown that this need not be difficult to arrange and that a very simple approach can be very successful.

The two drain processes which we have examined here both have disadvantages and it may well be that further study will provide rewarding development. However, the important point displayed by these strategies is that it is possible to design an initial core allocation algorithm in which the accent is solely upon avoiding thrashing and still to leave a degree of freedom, in the form of a drain process, which can be utilised to improve performance without affecting stability.

5.8 Hoare's Algorithm

The core allocation algorithm described here was evolved over a period of time by C A R Hoare. The original proposal has been modified by Hoare as a consequence of intuitive arguments by himself. These were endorsed or prompted by simulations using the system simulator. The algorithm is of interest in that it shows a further successful way in which a drain process may be implemented. It is also of interest in that its settling time is well defined and can be altered by a parameter change.

Every logical drum revolution a number of pages of core are scanned by a pointer which moves cyclically round the core. Any that have remained unused since they were last scanned are 'victims'. If a victim has a valid backing store copy then it is immediately added to the free list, assuming it is not already on the list, otherwise it is placed on the drum queue and is freed after the transfer takes place. No page replacements can be forced by a page demand and so processes may only obtain more core by obtaining pages from the free list. Hoare calls this drain process 'second-chance' page replacement (Hoare and McKeag, 1972).

Thus if core demand is light the free chain builds up until it is large enough to justify the loading of a

further process. If demand is heavy no victims will be found and the free list will be diminished. When the free list diminishes to zero all processes could be halted due to core contention, a situation equivalent to thrashing. The proposed load shedding component, which is necessary to deal with such cases, is that when a number of processes are waiting for a free page, the highest priority process always gets preference.

However, without further refinement this allocation strategy will be unsatisfactory. While the free list is non-empty the mechanism intended to block low priority processes from obtaining core in cases of high memory demand relies upon the occurrence of events, page demands, which are stochastic in nature. As we have seen with Horning's algorithm such blocking mechanisms are not sufficiently effective. Furthermore, when the free list is empty processes may be halted and unable to access their in-core pages and so blocks of storage may be released by the scanning mechanism. This will completely destroy any effectiveness which the blocking mechanism may have had.

To overcome this a process is not allowed to acquire a page from the free list unless the number of free pages exceeds twice, its own priority number. Thus each process leaves a float of two pages for the benefit

of each higher priority process. The number of free pages thus places a limit upon the number of processes allowed to compete for core. A process halted in this way may only continue if the free list size exceeds three times the process' priority plus six. Thus the effective multiprogramming level is controlled by the size of the free list.

Suppose a group of processes increase their memory demand suddenly. Eventually the lowest priority process will be halted from obtaining core. This should both immediately reduce demand and eventually increase supply since that process's pages will become victims. However, if this is insufficient further processes will be suspended until supply and demand are equal. If supply increases then a process will be reactivated. This increases demand and reduces supply so that the increase of the free list is retarded. However, if this is insufficient further processes will be reactivated until a balance is reached.

Further thought led to the suggestion that it may be advisable to impose a delay upon the rate at which processes are suspended or reactivated. For example, when a process has been suspended the drain should remove all of that process's pages (one cycle of the scanning pointer) before suspending any further process. Again

when a process is activated or reactivated it should be allowed to obtain its parachor (if possible) before any further process is allowed to do so.

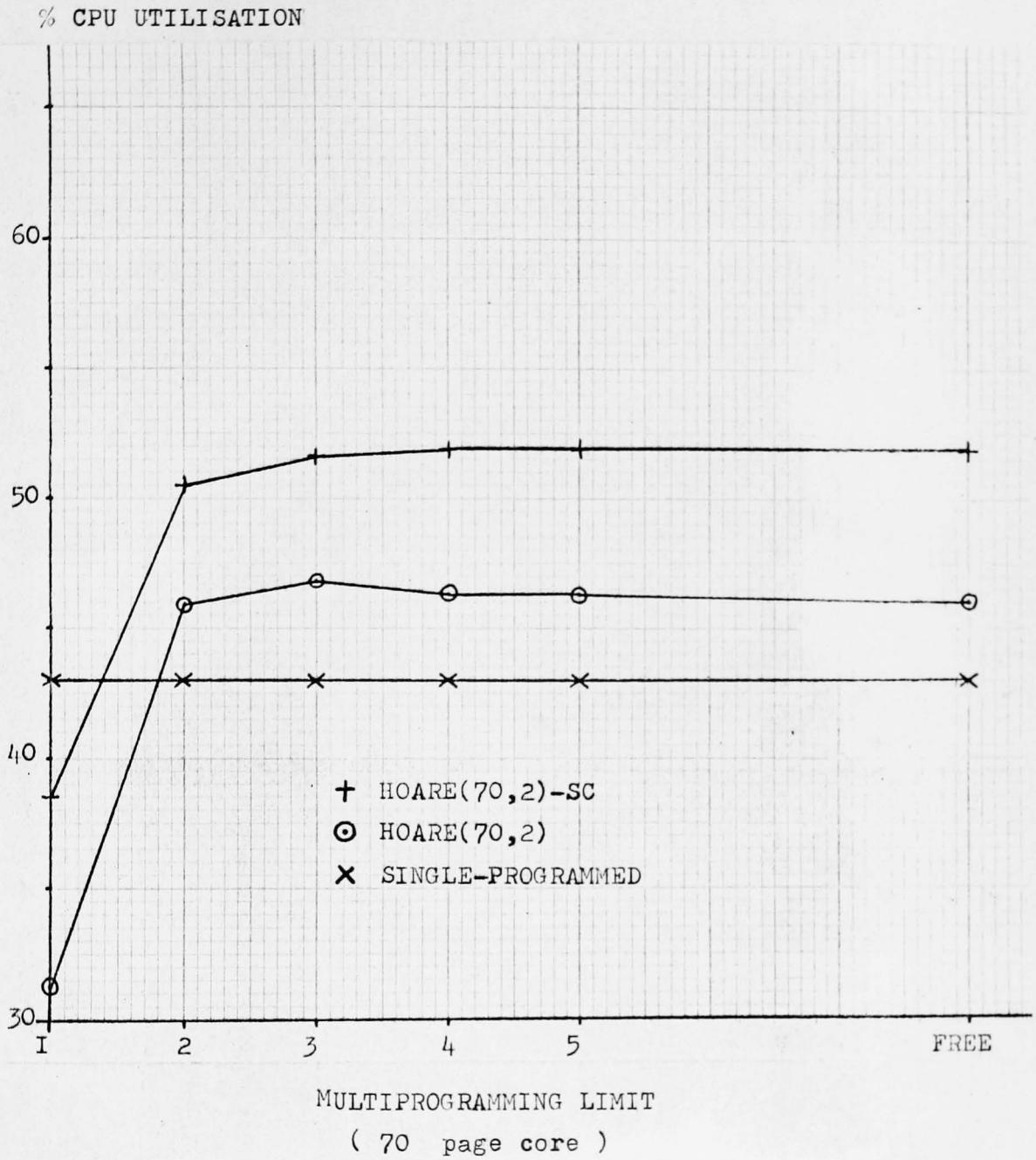
This form of suspension/activation hysteresis by timing can be synchronised to the scanning pointer of the drain process and is very easy to implement. It is better than using the size of the free list alone for control purposes since the free list fluctuates too randomly to be a good parameter for controlling hysteresis. In particular the free list will suddenly increase in size when a process terminates execution. Without the timing hysteresis a number of processes would be allowed to compete for core without any regard for the total memory demand. This will lead to rapid depletion of the free list which may well cause processes to wait for the drain to free pages. This is a situation to be avoided.

Unfortunately, we are unable to simulate algorithms which take account of the individual pages of a process. The account of the attempted simulation of an LRU derivative (section 4.4) shows the extent of our problem. However it is possible to simulate FIFO page replacement, and since the second-chance algorithm approximates to FIFO under overload conditions, it was thought that it would be a fair test to use it in the simulations. The algorithm simulated was Hoare's algorithm with the scanning pointer

moving two pages every second logical revolution of the paging drum, a simulated time of 70 milliseconds. The standard workload was used.

Let us consider the graph of CPU utilisation against Multiprogramming Limit in figure 5.19. We see that the Hoare algorithm avoids thrashing, the graph showing the characteristic non-decreasing function of Multiprogramming Limit. Of special interest is the result for Multiprogramming Limit of 1 in which CPU utilisation is depressed below that obtainable by simple single-programming. This may be explained by the action of the drain process in this algorithm which, as with Lynch's algorithm, continues to drain pages when there is no core contention. This leads to unnecessary page faults which depress CPU utilisation.

Let us now consider the core maps for Hoare's algorithm with 70 pages of core, using the standard workload. An example of the loadshedding mechanism in operation is shown by lines 566-574 of figure 5.20. At line 566 sufficient free core is available to allow the introduction of a new process. Process '5' is admitted but quickly uses up the core available to it and is suspended. Thus in lines 568-570 we see that process's pages are being drained from core. Process '5' remains suspended until the higher priority process '4' terminates execution enabling process '5' to be reactivated.



Hoare's algorithm - CPU utilisation against Multiprogramming Limit for the standard mix

Figure 5.19

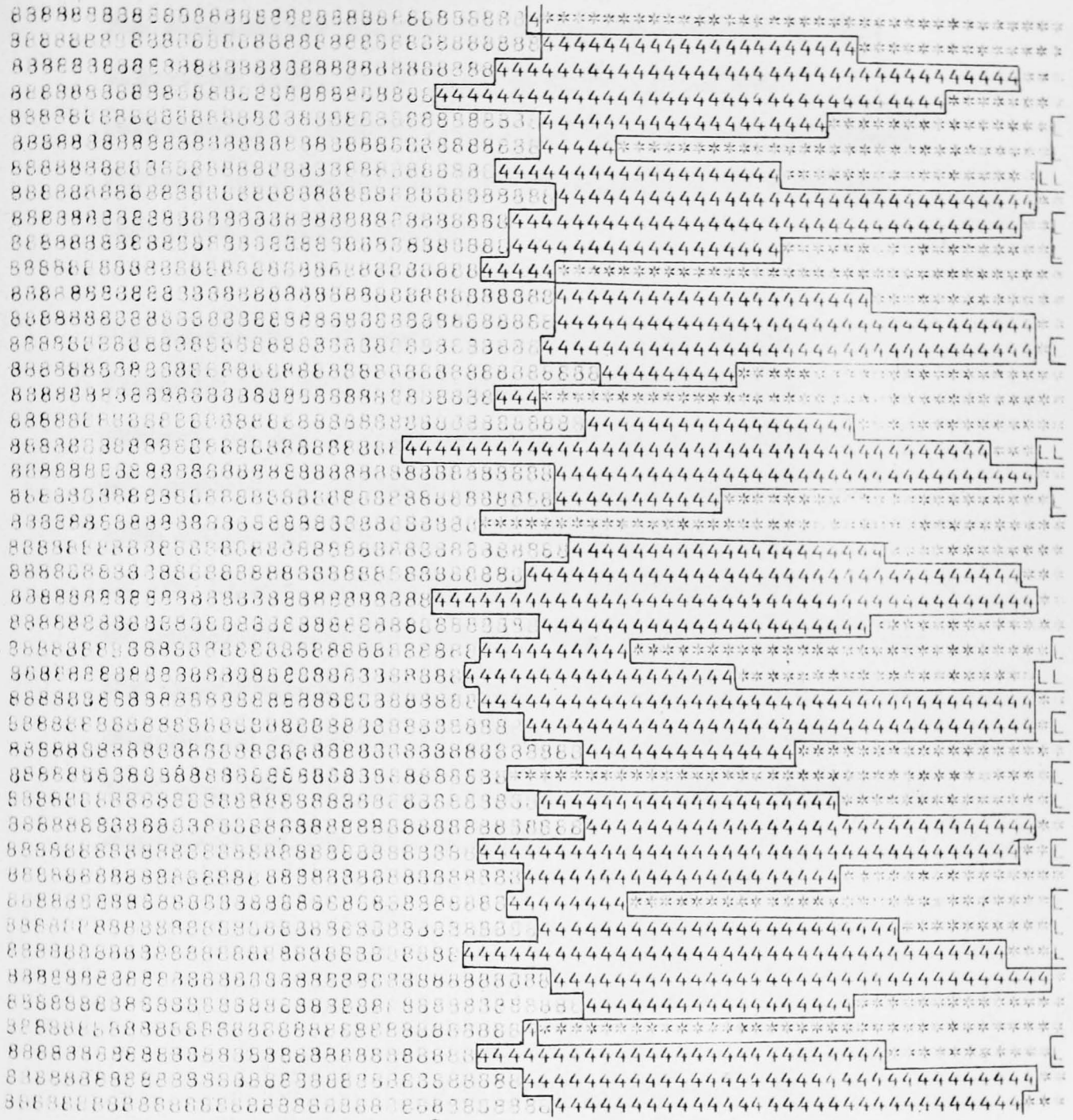
Hoare's algorithm - core map for the standard mix

Figure 5.20

Lines 581-590 display the manner in which the time hysteresis prevents a sudden influx of processes into core when process '4' terminates execution. We can plainly see processes being introduced at the rate of about one each 2.5 seconds, the time taken by the scanning pointer to complete a cycle of core.

In figure 5.21 lines 480-522 show a situation in which processes '8' and '4' have a combined parachor in excess of core size. Process '4' is activated, exceeds available core and is suspended at regular intervals which are synchronised to the scanning pointer. This is an unfortunate situation in that some process with a smaller parachor than process '4' could have utilised the core available more effectively.

To gain some feel for the improvement to be expected by replacing the FIFO page replacement by second-chance replacement the results in figure 5.19 marked HOARE (70,2) - SC were obtained. Here we assumed that there was a constant probability of 0.5 that a page belonging to an activated process had been used since the last scan. We make no attempt to justify this gross oversimplification other than to say that one would expect the drain rate of second-chance to be far less than that of FIFO, this was a simple way to obtain the desired effect. As would be expected there is a significant improvement in the results obtained.



Hoare's algorithm - core map for the standard mix

Figure 5.21

The simulations suggest that the performance of this algorithm might still be increased by further refinements. However, it would be difficult to assess further refinements due to our inability to simulate the second-chance aspect.

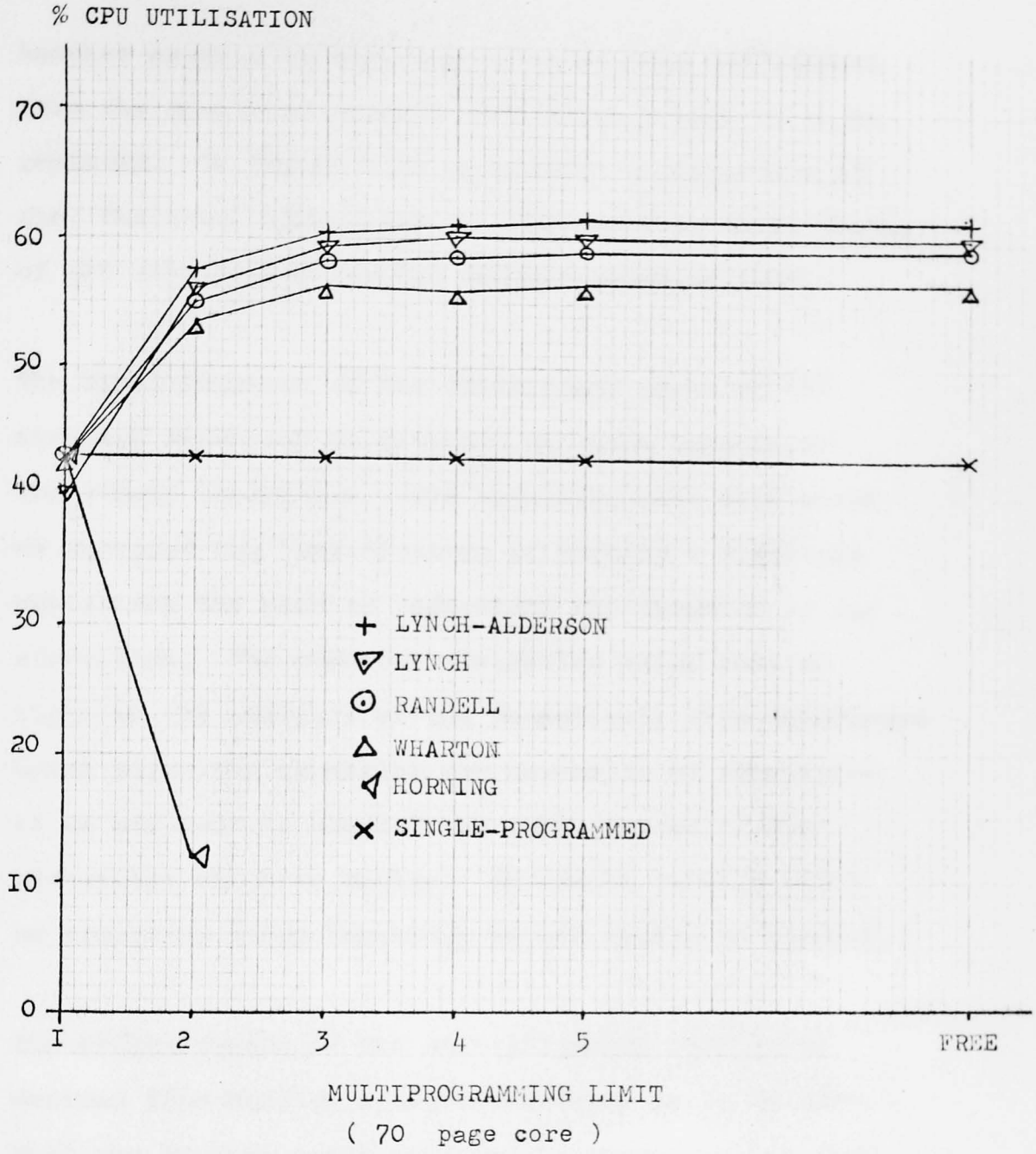
Our principal interest in this strategy derives from studying its settling time. The settling time in respect of any single stimulus, such as a sudden change of locality or process termination, is equal to the time taken for one cycle of the scanning pointer. This is the reason for synchronising suspension and activation of processes to that cycle. Thus Hoare's algorithm has the interesting property that its settling time may be changed by altering the presettable scanning rate. This has been of great value in our study of scheduling described in the next chapter.

5.9 Summary

It was the intention of this chapter to explore the manner in which feedback control could be applied to the design and understanding of core allocation. We were particularly interested in the avoidance of thrashing which as we have earlier discussed is caused by feedback instability in core allocation.

Starting with two stable but not very effective core allocation policies we have evolved a series of strategies. In figure 5.22 we present a comparison of these algorithms by superimposing their graphs of CPU utilisation against Multiprogramming Limit. These algorithms have been analysed and improved, sometimes, by appealing to arguments based upon feedback control notions. This process leads us to believe that the application of feedback control in this way is a useful and effective method of developing core allocation policies to avoid thrashing.

The process of development itself has provided some useful insights into the manner in which core allocation algorithms can be structured so as to simplify their design. Instances of useful structuring are provided by the concept of drain processes which may be superimposed upon an existing algorithm to improve effectiveness.



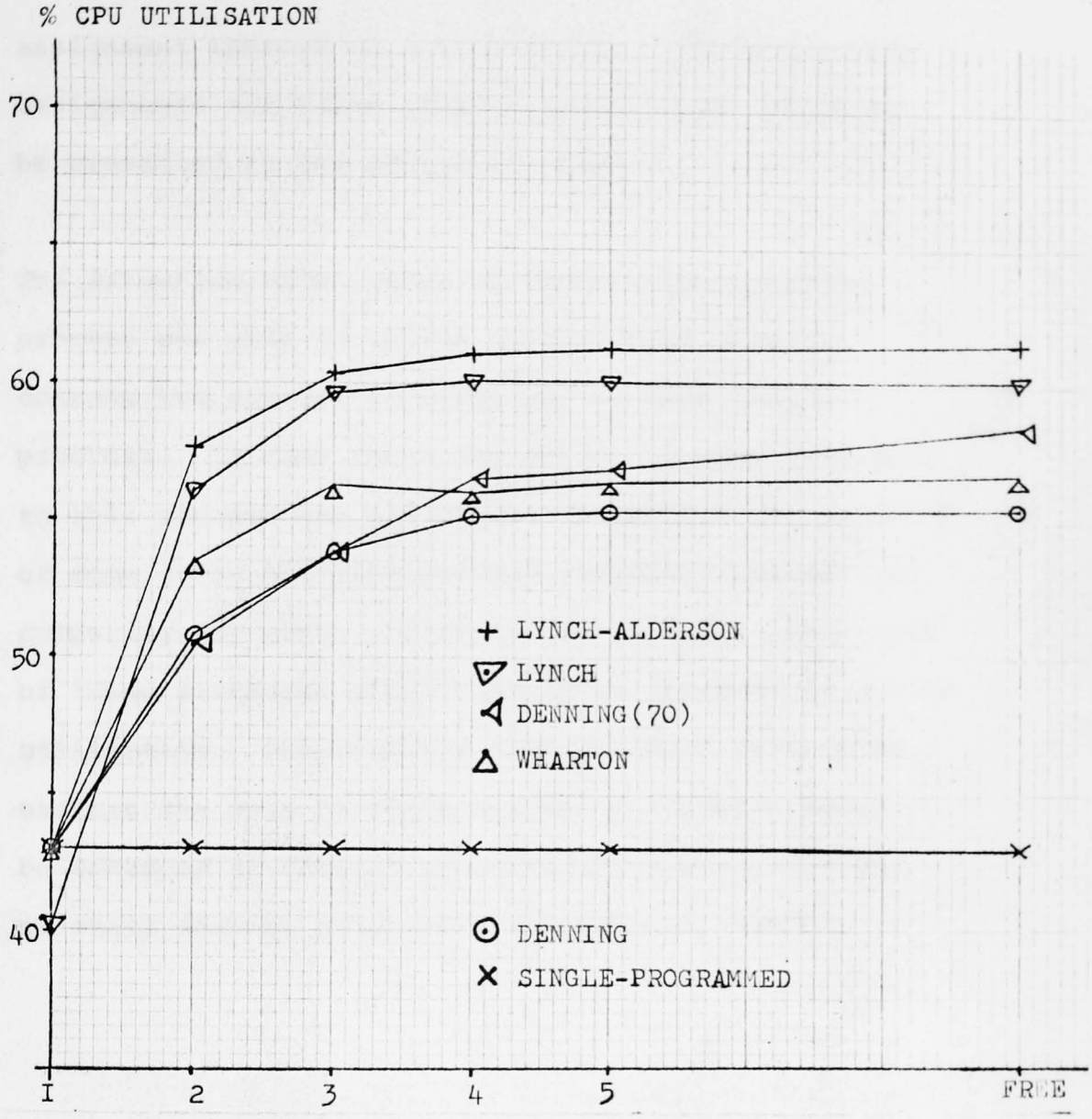
Summary of results - CPU utilisation against Multiprogramming Limit for the standard mix

Figure 5.22

Another example is the separation of page replacement from the choice of process from which a page is to be replaced. In figure 5.23 we present a comparison of the 'draining' algorithms by superimposing their graphs of CPU utilisation against Multiprogramming Limit.

The simplification of the constituent parts of the strategy which may be obtained in these ways is of the utmost importance. The appalling ease with which we accepted the justification of Horning's algorithm emphasises the need to understand the dynamics of our algorithms. The numerous subtleties which come to light in the analysis of the deceptively straightforward Lynch algorithm emphasise the necessity of simplicity if we are ever to predict the consequences of the mechanisms which we design. We cannot justify using an algorithm whose behaviour we are unable to predict.

The effectiveness of the anti-thrashing algorithms derived from Wharton's algorithm leads us to believe that any further gains obtained by improving the core allocation strategy will be small. It now seems clear that if further increases in system utilisation are to be achieved the problems of the 'mix' of processes which is presented to the system must be considered. In our simulations so far the composition of the mix has been totally determined by the initial external priority



Summary of results - CPU utilisation against Multiprogramming Limit for the standard mix

Figure 5.23

assignment imposed on all processes. These priority assignments are naive causing undesirable mixes to be presented to the system at times.

The situation often arose in which a low priority process was able to obtain insufficient core to contain its current locality and so made little progress. In many cases the amount of core available to this process was sufficient to contain the parachor of some lower priority process currently barred from competing for core. Altering the relative priorities of these processes would lead to an increase in system utilisation. Observations such as these lead us to explore the ways in which system utilisation could be enhanced by dynamic priority assignment schemes; we shall discuss this topic in the next chapter.

CHAPTER 6

Scheduling and Dispatching

6.1 Introduction

We have treated core allocation as the major problem in designing an efficient operating system. Where core is a scarce resource, as it commonly is, the occurence of thrashing will overshadow all other performance problems. However, having countered thrashing by employing a suitable core allocation algorithm, the allocation of the CPU and I/O processors becomes increasingly important.

We have indicated two levels at which the allocation decisions might be made with regard to these resources. These levels are scheduling and dispatching. The intention here is to examine the manner in which the scheduling and dispatching levels of our hierarchy affect the performance of an operating system. In particular we shall discuss the interactions of scheduling and dispatching with core allocation and the constraints which must be imposed upon the various levels if effective operation is to be maintained. Once more we shall appeal to notions of feedback to aid analysis.

6.2 Scheduling

The function of the scheduler in our hierarchy is to define a priority ordering upon the various processes which require system resources. Typically the priority order will reflect management policies to favour certain kinds of process and to provide certain levels of service to batch, interactive and real time processes. It may also reflect system decisions aimed at deadlock avoidance. These requirements must be combined with a strategy to ensure that the demands made upon the system do not cause overloads of individual resources resulting in diminished effectiveness. To implement such a strategy will require monitoring of the system so that the characteristics of the various processes utilising the system may be determined. (It is well known that external agencies such as the programmer are a very unreliable source of such data). Thus some form of feedback of information is required.

The two problems which arise are:-

a) What information will be required to determine the process characteristics so that the priority order may be modified appropriately, and how might that information be collected?

b) What effect will the priority reordering have upon system performance?

Answers to the first problem will depend to a great extent upon the role which the system is intended to fulfill, therefore we shall not discuss them. However, it is possible to discuss the effect of changing priorities in a general way within the context of our hierarchy.

6.2.1 The Effect of Dynamic Priority Reordering upon System Effectiveness

We have defined our hierarchy so that core is allocated to contending processes with respect to a priority order imposed by the scheduler. We have shown that it is possible to design core allocation strategies which will allocate core amongst the contending processes so as to give a stable division of the resource. Now if we alter the priority order which dictates the manner in which core is divided then the allocation will change accordingly and, with an appropriate allocation strategy, will settle once more to a stable division.

Ideally, we wish the system to operate in stable mode so that the paging overheads will be minimised. However, each time the process priorities are reordered a period of operation is incurred during which the core allocation is stablilising and higher overheads occur. Thus we must arrange that priority reordering occurs at a rate which is such that the time spent in the unsettled state does not represent a significant proportion of processing time. Of course, process completion and submission also cause reordering which is unplanned. If the rate of occurrence of these events is high we may have to curtail our planned reordering or reorder only when these events occur.

If our only requirement was to minimise paging overheads then we would choose to reorder priorities as infrequently as possible. However, there are advantages to be gained from rearranging priorities. In particular, ordering with respect to the characteristics of the processes can improve the parallel use of the CPU and I/O devices. Since these characteristics change with time we may require to make frequent observations of process characteristics and make related priority alterations. To this end we may be prepared to accept an amount of overhead which we would hope to offset by improved system utilisation.

Wulf (1969) has described a scheme of process monitoring and priority reordering implemented within the Chippewa operating system on a CDC 6600. Here an attempt was made to order the processes with respect to their observed characteristics and the observed performance of the system. Processes which make heavy use of a resource in high demand may be suspended, thereby decreasing contention. Suspended processes which have been observed to use currently underutilised resources may be activated, increasing the parallel utilisation of resources. Wulf reported greatly improved system utilisation in spite of the added overhead of the increased swopping of processes between core and backing store, and the overhead of the monitoring and process selection algorithms.

However, there is an upper limit to the rate at which priority changes can be made. If we alter priorities at intervals which are shorter than the settling time of the core allocation policy then a stable core division will never be obtained and overheads will increase rapidly. In fact rapid priority changes will tend to negate any anti-thrashing properties which the core allocation algorithm may have.

6.2.2 Time-slicing as a Simple Example of Priority Reordering

To demonstrate these ideas let us consider a very common requirement of multiprogramming systems, that of providing rapid processing of small interactive processes whilst providing acceptable background service to more complex processes.

If one is to ensure that processes will complete within an acceptable time then processes of any particular type must not be blocked for inordinately long periods by other processes. It must be ensured that processes requiring a great deal of computation neither monopolise the CPU nor are prevented from completing because of the precedence being given to short computations. A simple way to avoid such problems is to allot each process a quantum of CPU time - a time slice. If the process does not complete within an alloted time slice then it is given further time slices as required but its priority to obtain the CPU is reduced.

For example, in a system in which the CPU is allotted on a round-robin basis, allocating the CPU to the next process in the cycle is equivalent to demoting a process to lowest priority when it has used its time slice. The time slice is used here as a simple feedback of information to the

scheduler upon which decisions to reorder priorities may be made.

Since time-slicing was easy to implement in the simulation model it was possible to carry out a number of simple experiments which demonstrate the effect of priority ordering upon system effectiveness. In these experiments we chose to utilise the feedback of information so as to improve the response of the system to short computations. To this end we demote any process which utilises all of its time slice to the end of the scheduler queue. New processes enter the scheduler queue in first-come-first-served order.

To assess the effectiveness of the scheme a figure of merit was computed for each simulated system, (Hellerman, 1969). This figure of merit is larger the more successful the system is in providing better service to short computations relative to long computations in a mix containing both. Hellerman's figure of merit is defined as:

$$\mathcal{H} = \frac{n}{\sum_{i=1}^{n} (e_i/x_i)}$$

where n is the number of processes submitted to the system, x_i is the processor time (I/O and CPU) required by the i^{th} process and e_i is the elapsed time to completion of the i^{th} process. The term e_i/x_i will weight the figure of merit

more significantly if it refers to a short process than to a long one. The figure has a maximum value of one when $e_i = x_i$, $i = 1,2,3, \ldots n$, that is when each process experiences no contention for any resource which it requires.

Before discussing the results obtained we make one further observation. Consideration of response time requirements for short processes implies that the best value for the time slice is that which is just long enough to allow short interactions to complete within one time slice. If the time slice is shorter than this value short computations will experience delays due to being requeued at time slice end. If the time slice exceeds this value then the CPU will pass more slowly from process to process and short computations entering the system will incur a greater queuing time to first service than with the shorter time slice. In the simulation experiments using the standard mix the apparently optimal time slice value is one second.

The algorithms simulated were Hoare's algorithm scanning two pages every 70 milliseconds, Hoare's algorithm scanning two pages every 140 milliseconds and Wharton's algorithm. The simulations were of a 70 page core.

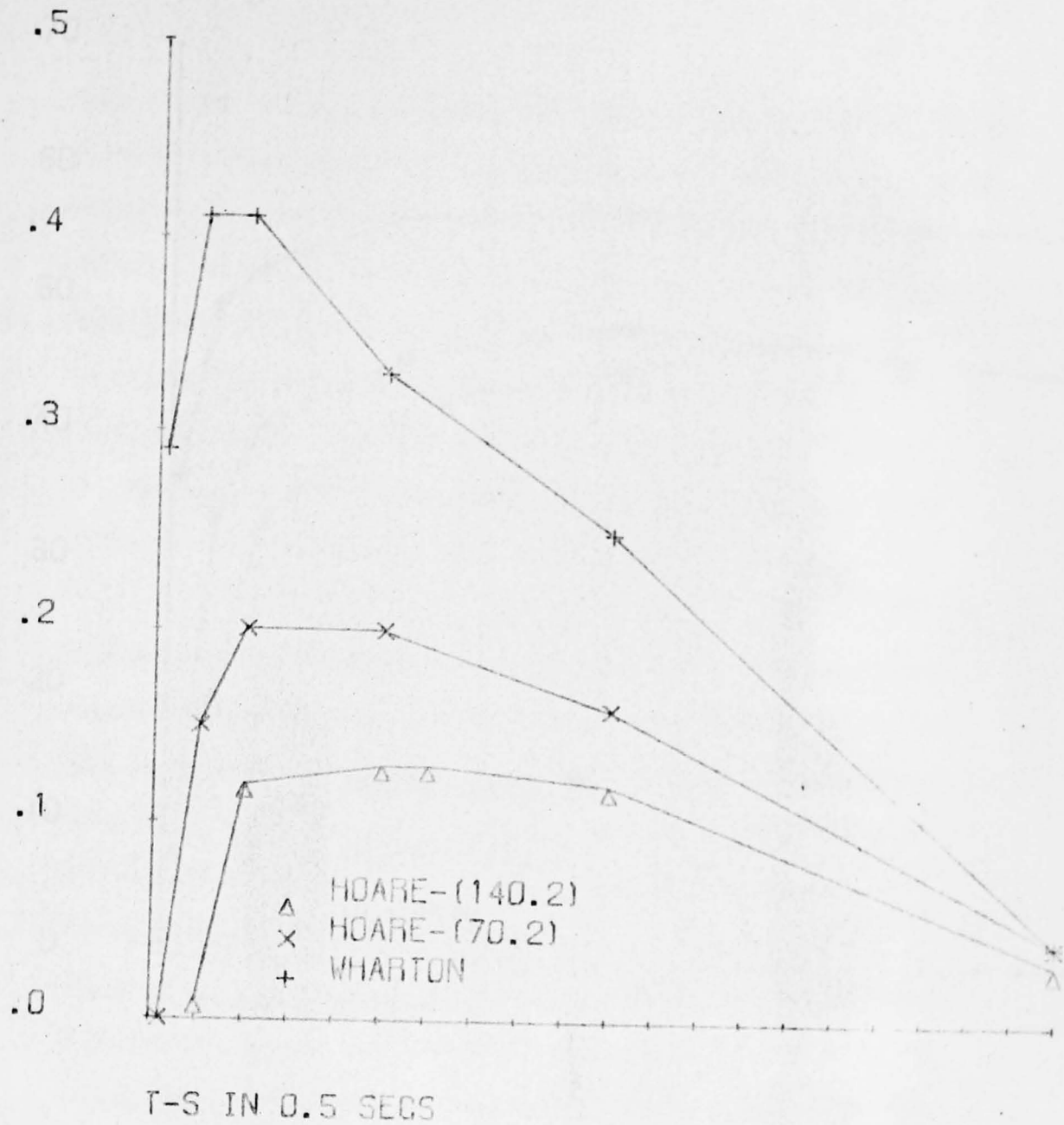
The settling times of the core allocation policies are approximately:

Hoare (2 pages, 70 milliseconds)	-	2.45 seconds
Hoare (2 pages, 140 milliseconds)	-	4.9 seconds
Wharton	-	0.6 seconds

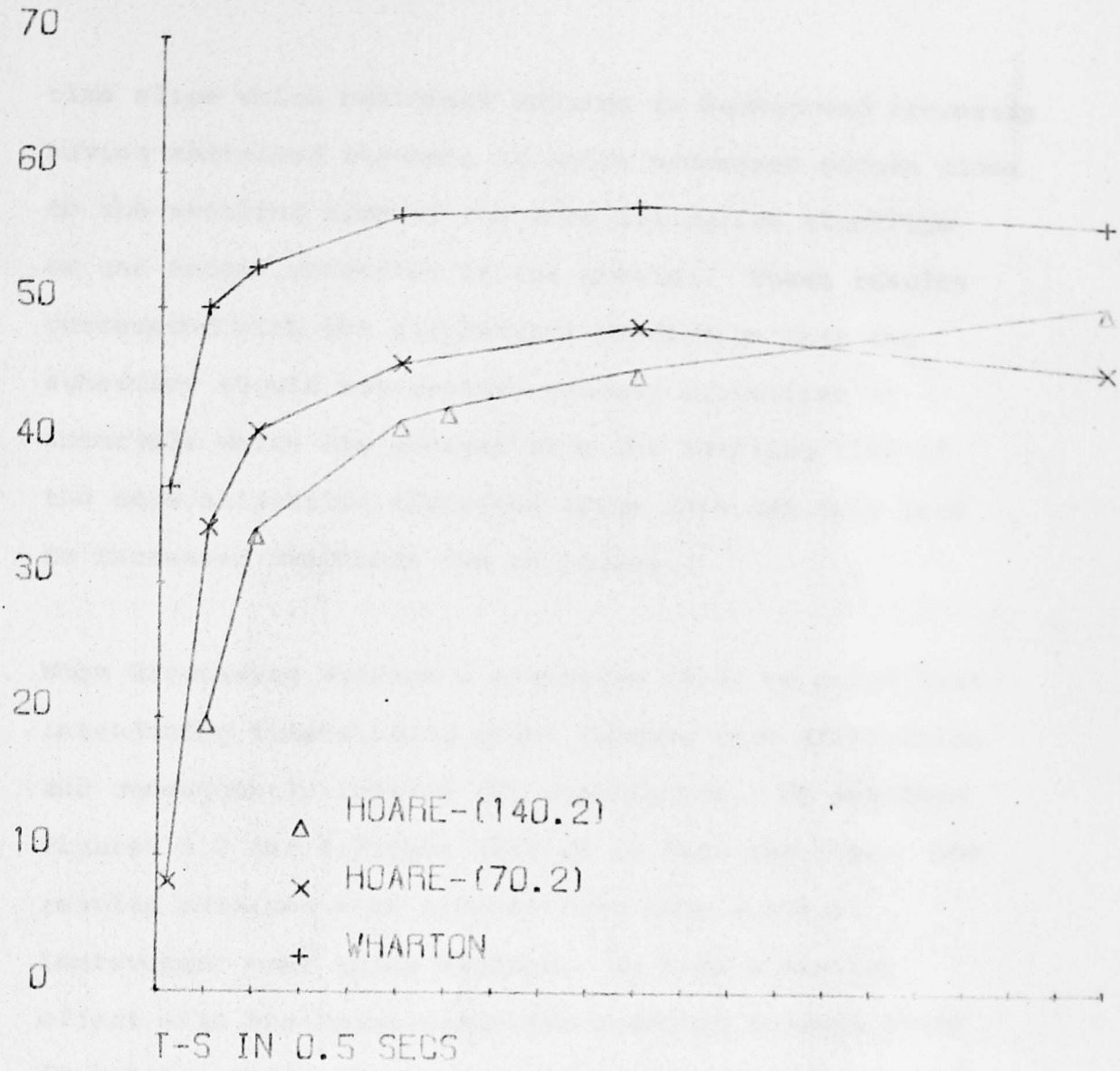
The settling time for Wharton's algorithm depends upon the number of pages which the process being demoted had in core at time slice end. The value of 0.6 seconds is obtained by assuming that half of the available pages (35) belong to the demoted process.

Let us now consider the simulation results obtained. Considerations of the mix presented to the system in the simulations leads one to expect maximum response for short processes when the time slice is in the region of one second. However, as the graph of Hellerman's figure of merit against time slice (figure 6.1) displays, only Wharton's algorithm with a settling time of less than one second shows this. Both of the Hoare algorithms with settling times exceeding one second attain maximum response with time slice values in excess of one second.

Combining the results shown in figure 6.1 with those of figure 6.2, the graph of CPU utilisation against time slice, we see that once a high level of response to short processes has been attained decreasing the time slice value only causes depleted service to background processes and increased paging overheads. In all cases the value of



Hellerman's figure of merit against time slice size
Figure 6.1



CPU utilisation against time slice size

Figure 6.2

time slice which maximises service to background processes having maximised response to short processes occurs close to the settling time of the core allocation algorithm or one second whichever is the greater. These results correspond with the theoretical prediction that the scheduler should not reorder process priorities at intervals which are shorter than the settling time of the core allocation algorithm since this can only lead to increased overheads due to paging.

When discussing Wharton's algorithm (4.2) we noted that introducing time-slicing might improve core utilisation and consequently improve CPU utilisation. We see from figures 6.2 and 4.3 that this is in fact the case. The results obtained with time-slicing show a slight improvement over those without. We note a similar effect with the Hoare algorithm scanning 2 pages every 70 milliseconds. However, we see that the maximum CPU utilisation achieved is for a time slice well in excess of the settling time of the algorithm. We would expect this for although we expect to maximise short process response by making the time slice small we noted that the system should operate most effectively when in the settled state, that is for time slice values greater than the settling time of the core allocation algorithm. As the time slice value decreases the unsettled state becomes increasingly dominant. Interaction between the

core allocation and scheduling components increases and paging overheads rise rapidly as the proportion of time during which the core allocation algorithm is settled becomes less significant. Again these results agree with the theoretical predictions.

Thus as predicted by considering the core allocation algorithm as a feedback controller experiencing external stimuli in the form of priority reordering caused by the scheduling component, system efficiency is sensitive to the rate of interaction between system components. The results display the importance of being aware of the interactions and the limits they place upon the values of the parameters of the various system components.

6.2.3 Bias in Scheduling Algorithms

Frequently decisions about the role of the system imply the necessity of biassing the scheduling algorithm towards a certain type of process. For example when optimising response to short processes it is natural to build in a bias to those processes. The beneficial effect of such biassing is demonstrated in this case by the results discussed above. However, introducing bias into an operating system may have undesirable consequences.

Let us consider the example of a system designed to support interactive processes and a background batch stream. By giving precedence to interactive processes it may be possible to exhaust the supply of such processes leaving only the background stream. Due to bias the mix is modified. Since the system is not designed to cope with a pure batch stream its algorithms may be less effective.

Care must also be taken that the scheduling algorithm does not inadvertently cause biasing towards certain types of process. Again this will lead to backlogs of other processes, the execution of which may require the system to work outside its operating region or in other ways cause excessive delays to certain process types.

An example of such biasing is given by Lehman and Rosenfeld (1968). Here processes requiring large amounts of memory were delayed excessively until they dominated the mix at which time the backlog was rapidly depleted and the cycle begun again.

6.3 Dispatching

In our hierarchy dispatching is concerned with the allocation of CPU and I/O processors. It should be the aim of a dispatcher to optimise the parallel usage of these resources in order to increase throughput. The most important component of a dispatching strategy is the algorithm which governs the utilisation of the CPU since all processes must obtain the CPU to some extent if they are to progress. The relative demands for the CPU made by various processes may vary widely and it has long been recognised that for some processes the limiting factor upon their rate of progress is CPU availability whilst others may be limited by I/O availability.

Sherman (1972) has confirmed by experiment that the parallel usage of CPU and I/O processors in a multi-programming environment is optimised by giving the CPU to the process which will compute for the shortest period before issuing an I/O request. Advantage has often been taken of this by giving I/O bound processes priority over CPU bound processes since on average I/O bound processes will compute for shorter periods between requests for I/O.

However the relative demands made by a particular process for CPU and I/O can vary dynamically with the progress of

the computation and the process may switch between being CPU bound and I/O bound. Therefore, it is necessary to be flexible in the assignment of priority to use these resources. A strategy of monitoring the demands of the process coupled with a dynamic selection of the next process to utilise the CPU and I/O devices may be fruitful.

6.3.1 Frequency of Choice of Process to Dispatch

An important consideration to be made before examining possible dispatching algorithms is the frequency with which there will be a choice of processes to dispatch. If the number of processes contending for any resource seldom exceeds one then there is no gain to be made over the simplest possible dispatching algorithm. The overhead incurred by more sophisticated methods could not be balanced by improvement in performance.

With the probable exception of the line printer individual I/O devices are certain to experience less contention than there will be for the CPU. Queues for the printer are not uncommon but they are often dealt with by a spooling system due to the great disparity in speed between the CPU and the line printer. It will seldom be necessary to make a choice between processes competing for an I/O device other than the line printer. Also the use of the CPU and I/O processors by a process is distinctly different. I/O operations are requested singly with each request being preceded by many CPU operations. These differing usage characteristics imply that the conditions necessary for a dispatching decision are different in the two cases. CPU and I/O devices may only be reassigned when the current operation has completed. However, it is possible that the CPU will

still be required by the current process at the end of the present operation whereas this will be unlikely for an I/O device.

It is possible then to make a choice for CPU assignment whenever another process other than that currently assigned the CPU becomes able to use it. That is, a decision may be made whenever the number of processes requiring the CPU is at least two. With I/O however, where the process currently receiving service is very unlikely to demand further service immediately upon termination of the current operation, two further processes must require service from that I/O device before a choice can be made. With the exception of the paging drum this will be a rare situation and may therefore prove inefficient to cater for.

The paging drum is somewhat exceptional since indirectly all processes require service from it because of the structure of the system. It is possible, by employing sector queuing techniques (Weingarten, 1966; Denning, 1967; Coffman, 1969), for the dispatching algorithm to initiate a chain of I/O commands to the drum which satisfy the needs of many processes in one I/O operation.

Results obtained by the system simulator using the standard workload for the 9 sector drum organisation

employed by MTS give average sector queue sizes of less than 0.1 for Lynch's algorithm. These results show that it would be infeasible to use the simulator to study the effects of applying a dispatching algorithm to requests for transfers from the paging drum. Measurements of MTS also showed short sector queues with the drum never loaded to more than 30% of its capacity even when thrashing was being experienced. Since the dispatcher for the paging drum, which employed a very sophisticated algorithm was using between 8% and 15% of available CPU time it was considered that a simple first-come-first-served algorithm would provide an equally effective dispatcher for greatly reduced overheads.

Thus we would expect to obtain the most benefit by providing simple dispatching algorithms for I/O processors and confining our attention to the dispatching of processes contending for the CPU. Average CPU queue lengths for Lynch's algorithm obtained from simulations rose from nearly one with 70 pages of core to approximately three with 130 pages, suggesting that a choice of processes would occur sufficiently often to warrent the application of a sophisticated dispatcher for the CPU.

6.3.2 Application of Feedback to CPU Dispatching

It would not be appropriate to attempt to catalogue the ways in which one might estimate the process most likely to compute for the shortest period before issuing an I/O request. Sherman (1972) has already made a study of a number of important theoretical and practical algorithms by simulation. In particular the study included some predictive dispatchers.

The basic technique is to predict the CPU time required by a process before its next I/O request, based on the past behaviour of the process. The CPU is given to the process with the least predicted value. Two different methods were tested by Sherman. The first developed an exponential smoothing predictor as follows. If x_{n-1} is the $(n-1)^{st}$ CPU service time for a process and $\hat{x}_{n-1}$ is the $(n-1)^{st}$ prediction then the prediction of x_n is

$$\hat{x}_n = \alpha . x_{n-1} + (1-\alpha)\hat{x}_{n-1}$$

where $0 \leq \alpha \leq 1$. The larger the value of α the more heavily weighted is the most recent past. The second method used the 'complete history', predicting the next CPU service time to be the mean of all past service times for that process, the formula being

$$\hat{x}_n = (x_{n-1} + \hat{x}_{n-1} \ (n-1))/n$$

These 'feedback' dispatchers were shown to be successful when compared against the theoretical best in which the correct process is always chosen. They chose correctly in up to 75% of cases.

The environment in which the feedback dispatchers were studied by Sherman did not include the external stimuli to the dispatcher from the core allocation component which would be found in a paging system. However, when time slicing was introduced creating a source of external stimuli a performance improvement was observed. The time slice values were of the same order of magnitude as the settling times of Wharton's and Lynch's algorithms and therefore we might expect these algorithms to perform equally well in a paging environment. However, dispatching algorithms in which processes are ordered according to some calculated statistic require a linear search of a list for each dispatching decision made and so entail a certain amount of overhead.

The dispatching algorithm of MTS overcomes the necessity for this linear search. Here the process which requested I/O most recently is predicted to be the process which will request the least CPU service before issuing a further I/O request. Thus whenever a process completes I/O service it is placed at the front of the dispatcher queue, thus ordering it correctly with respect to

its predicted value of CPU service requirement. The algorithm is also preemptive and so the process obtains the CPU immediately. The notion behind this scheme is that processes which make frequent demands will predominantly occupy the leading positions in the dispatcher queue. CPU bound processes will be demoted to the lower positions.

The queue need never be scanned either to insert a process in priority order or to select the next process. The algorithm does however have one distinct disadvantage. Under the algorithms described by Sherman the predicted length of next CPU request for a process can be retained during suspension of the process by the core allocation algorithm. This is not possible under the MTS scheme. Thus the 'explicit' predictor methods have an advantage when considering settling characteristics of the algorithms.

Since dispatching occurs at the lowest level of our hierarchy it is necessary that the algorithm be economical of CPU time for otherwise it would have a significant effect upon its own settling time. The settling time of the dispatching algorithm must of course be significantly less than that of the core allocation algorithms if the system is to be in a settled state for a sufficient proportion of the time. Thus the dispatching algorithm cannot be complex. The feedback dispatchers described by Sherman display that simple but successful algorithms can be constructed.

6.4 The Implications of the Control Hierarchy

In this and the previous chapters we have discussed the feasibility of the hierarchy of control which we proposed in 1.3. By describing the hierarchy in feedback terms we have shown that for an operating system of this design to function effectively certain constraints must be imposed upon the interaction of the various levels. The levels themselves can be claimed to perform logically separate functions and the function of a particular level is one which can logically be delegated by the immediately higher level. However, we have seen that this is not sufficient for the whole to work as an effective unit.

Dijkstra in designing the THE operating system (Dijkstra, 1968) created levels logically, building each level from the building bricks of lower levels. Each level of the THE operating system can be seen as delegating to lower levels tasks which need not be dealt with in detail at the current level. The structure enabled Dijkstra to infer the correctness of the implementation of the design and allowed simplification of the interfaces between levels. However, the THE system is liable to suffer gross performance degradation due to thrashing, and the operators have the responsibility of monitoring the system's behaviour, and of deciding whether one or more jobs should be postponed.

Dijkstra has said that on reflection one of the vital characteristics of the structuring of the THE system was that each level of abstraction concerned events that occurred at appropriately different 'time grains'. The lowest level concerned process switching (50 sec); the second virtual memory management (40 msec. drum revolution time); the third level operator messages (1-2 sec), the next peripheral assignments (a minute or so) and the top level process submission (many minutes). He believed that this sort of relationship was essential if a level of abstraction was to be able to safely ignore what was going on at lower levels. If it were otherwise each level would be required to involve itself with all sorts of complicated interactions.

This is sound reasoning and has played an important part in our own choice of control hierarchy. However, our studies have shown that the division of tasks upon the basis of time grain is not a sufficient condition for avoiding the gross performance degradation which occurs due to the uncontrolled interaction of the various levels of control. Not only must these time grains be suitably separate under normal conditions, but their separation must be maintained under varying conditions. As the time grains of two adjacent levels of control approach each other, increasingly complex interactions occur until the controls provided are no longer adequate and performance

degradation results. One must ensure that the algorithm used at each level is stable and settles to that stable level sufficiently rapidly.

If we take care to design an operating system in the way described above a further possibility arises besides the important ones of having confidence in its correctness and avoiding gross performance degradation. It may also be possible to analyse our design.

Simon and Ando (1961) studied a technique of variable aggregation and the concept of nearly completely decomposable systems applied to the theory of Econometrics. Variable aggregation is based on the recognition that in complex systems represented by a large number of state variables, the state variables can be classified into groups such that interactions within groups can be studied as if interactions amongst groups did not exist, and interactions amongst groups can be analysed without reference to interactions within groups. This is trivially true if variables within each group depend only upon variables within the same group. Such a system is said to be completely decomposable. Simon and Ando showed that the technique still yields good approximations when the interactions between groups are weak in comparison to interactions within groups. Such a system is said to be nearly completely decomposable. The theory of nearly completely decomposable systems also shows

that it may be possible to derive confidence limits upon the predictions of a model produced by this technique.

Courtois (1971) has studied the application of this work to the modelling of multiprogramming operating systems and the method should be applicable to the hierarchically structured system we have been discussing. Here the variables for each level of control would be aggregated separately. With the variables aggregated in this way, our hierarchic structure in which the separation of time grains between levels is maintained by stable controls at each level, ensures that the system is nearly completely decomposable.

Provided the algorithms of each level of the hierarchy were able to maintain the separation of the various time grains then the decomposition would be valid. Each level of the hierarchy could be analysed separately and then by combination of these analyses a model of the complete system might be achieved. We note that in cases in which the algorithms used are not stable the system would not be nearly completely decomposable in all circumstances and the results of an analysis by this technique would not be valid in all conditions.

CHAPTER 7

Conclusions

The intention of this thesis has been to apply the concepts of feedback control theory to the design and analysis of resource allocation algorithms in multiprogramming operating systems. In particular we have tried to emphasise the way in which feedback arises naturally within the resource allocation component of an operating system. This led us to explore the ways in which the notions of feedback control could be exploited in resource allocation, either to stabilise these naturally occuring feedback loops or to introduce further feedback effects, so as to improve performance. We have done this by examining resource allocation algorithms and their interrelationships within the framework of a simple control hierarchy.

A hierarchic control structure was chosen because it is possible to reduce the complexity of the resource allocation problem by dealing individually with the various resources and then with the interactions between the allocation algorithms. Furthermore, the interest which has been generated in hierarchic structuring by the work on program correctness, particularly its use in the design of the THE system by Dijkstra, enhances the possible value of the study. By studying this

control structure we were able to display the importance of the rates at which the resource allocation algorithms allowed events to occur at the various levels of the hierarchy. This showed by means of the concepts of feedback control theory the necessity for strict control of these rates.

We were also able to indicate that the construction of a resource allocation strategy in this hierarchic manner using stable algorithms would enable us to fulfill conditions required by the method of Courtois. This method might enable analysis of the strategy.

Detailed study of core allocation algorithms in a paging environment enabled us to demonstrate that stable algorithms could be designed and furthermore were capable of development to provide effective utilisation of core. The study of the anti-thrashing algorithms also provided us with useful insights into the ways in which thrashing can be precipitated.

Our studies have been confined to three areas - scheduling, dispatching and allocation of core in a demand paging environment. However, although our studies have been restricted the results obtained and the algorithms developed should be of wider application

and should be of value when adding further levels to the resource allocation hierarchy.

The most obvious further application of the results is of the core allocation algorithms to a segmented memory space. Basically we need only replace 'page' by 'segment' in the description of the algorithms. However, since segments vary in size, the fetch of a single segment may require multiple deletions. This would necessitate modification of the algorithm.

A further problem arises in the application of the drain or delete policies. Because of the varying segment sizes it is more difficult to predict the effect of the draining rate upon the processes. These problems however should be soluble.

Another possible application of the results arises where it has been found that the paging drum may not be able to hold the virtual memory of all the processes active in the system. This might prompt the incorporation of a further level of storage hierarchy, so that the virtual memory of certain processes is held on disc. Access from disc is of course much slower than from the paging drum and so there is much to be gained in ensuring that pages of the highest priority processes occupy the paging drum. This problem is very similar to that of

deciding which pages should be held in core and which on drum. Here, however, the time grain is more coarse. Similar solutions to those used in the allocation of core may be useful in the allocation of paging drum space. In view of the slower access rates it may be more effective to transfer segments rather than pages and we have already seen how such algorithms might be constructed. We should remember that adding this level of resource allocation may require adjustments in the time grain of the scheduling algorithm.

The necessity of ensuring that the time grains of the levels of the control hierarchy stay sufficiently separated applies to hierarchies other than that which we have described. We have shown how the values of the time grain are determined by the algorithms employed in the allocation process. Consequently we have shown the necessity of employing algorithms which have the property of maintaining their time grain within acceptable limits and whose time grain and setting time we are able to estimate. We can only achieve this however if we understand the algorithms concerned.

One by-product of our work has been to increase our awareness of the ease with which one can fall into the trap of developing an algorithm whose behaviour cannot

in practice be successfully predicted. The richness of variety and depth of subtlety of the action of the simple feedback control algorithms we studied stands as a warning that even slight complications may destroy our ability to comprehend the resulting algorithm. Yet the typical operating system contains many algorithms whose behaviour is far more impenetrable than these comparatively simple algorithms!

REFERENCES

BELADY, L.A. 1966
A Study of Replacement Algorithms for a Virtual Storage Computer.
IBM Systems Journal, Volume 5, Number 2, 1966. pp 78.

BELADY, L.A. and KUEHNER, C.J. 1969
Dynamic Space Sharing in Computer Systems.
Comm. ACM, Volume 12, Number 5, May 1969, pp 282-288.

BÉTOURNÉ, C., KAISER, C., KRAKOWIAK, S. and MOSSIÈRE, J. 1971
System Design and Implementation Using Parallel Processes.
IFIP Congress, Ljubljana, August 1971, pp TA-3 31-36.

BÉTOURNÉ, C., FERRIÉ, J., BOULENGER, J., KAISER, C. and KRAKOWIAK, S. 1970
Process Management and Resource Sharing in the Multi-access System ESOPE.
Comm. ACM, Volume 13, Number 12, December 1970. pp 727-733.

BRAWN, B.S. and GUSTAVSON, F.G. 1968
Program Behaviour in a Paging Environment.
Proc. AFIPS, Fall Joint Computer Conference, Volume 33, Part 2, Spartan Books, New York, pp 1019-1032.

BREED, L.M. and LATHWELL, R.H. 1968
APL\360 Implementation.
Interactive Systems for Experimental Applied Mathematics.
Klerer, M. and Reinfelds, J., Academic Press, 1968
pp 390-399.

BUNT, R.B. and HUME, J.N.P. 1971
Self-Regulating Operating Systems.
University of Toronto, Dept. of Computer Science.

COFFMAN, E.G. Jnr. 1969
Analysis of a Drum I/O Queue Under Scheduled Operation in a Paged Computer System.
Journal ACM, Volume 16, Number 1, January 1969. pp 73-90.

COFFMAN, E.G. Jnr. and KLEINROCK, L. 1968
Computer Scheduling Methods and their Countermeasures.
Proc. AFIPS, Spring Joint Computer Conference, Volume 32, 1968. Spartan Books, New York, pp 11-22.

COFFMAN, E.G. Jnr. and VARIAN, L.C. 1967
Further Experimental Data on the Behaviour of Programs in a Paging Environment.
Comm. ACM, Volume 11, Number 7, July 1968, pp 471.

CONTI, C.J., GIBSON, D.H. and PITKOWSKI, S.H. 1968
Structural Aspects of the System/360 Model 85: General Organisation. IBM Systems Journal Volume 7, Number 1, 1968. pp 22.

COURTOIS, P.J. 1973
Instabilities and Saturation in Multiprogramming Computer Systems.
MBLE Research Laboratory, Report R211, January 1973.

DENNING, P.J. 1967
Effects of Scheduling on File Memory Operations.
Proc. AFIPS, Spring Joint Computer Conference, Volume 30, Spartan Books, New York, 1967. pp 9-21.

DENNING, P.J. 1968a
Thrashing: Its Causes and Prevention
Proc. AFIPS, Fall Joint Computer Conference, Volume 33, Spartan Books, New York, 1968. pp 915.

DENNING, P.J. 1968b
The Working Set Model for Program Behaviour.
Comm. ACM, Volume 11, Number 5, May 1968. pp 323-333.

DIJKSTRA, E.W. 1968
The Structure of the 'THE' Multiprogramming System.
Comm. ACM, Volume 11, Number 5, May 1968. pp 341

FINE, G.H., JACKSON, C.W. and McISSAC, P.V. 1966
Dynamic Program Behaviour Under Paging.
Proc. ACM National Conference, 1966, pp 223.

GOODE, H.H. and MACHOL, R.E. 1957
Control System Engineering.
McGraw-Hill, 1957, Library of Congress 56-11714.

GRABBE, E.M., RAMO, S. and WOOLDRIDGE, D.E. 1958
Handbook of Automation, Computation and Control,
Part 1. John Wiley and Sons, 1958, Library of
Congress 58-10800.

HELLERMAN, H. 1969
Some Principles of Time-Sharing Scheduler Strategies.
IBM Systems Journal, Volume 8, Number 2, 1969. pp 94.

HOARE, C.A.R. and McKEAG, R.M. 1972
A Survey of Storage Management Techniques - Part Two.
Operating System Techniques. Ed. HOARE, C.A.R. and
PERROTT, R.H. Academic Press, London, 1972. pp 132.

JOSEPH, M. 1970
An Analysis of Paging and Program Behaviour.
Computer Journal, Volume 13, Number 1, February 1970.
pp 48.

KLEINROCK, L. 1970
A Continuumof Time Sharing Scheduling Algorithms.
Proc. AFIPS, Spring Joint Computer Conference,
Volume 36, Spartan Books, New York, 1970. pp 453-458.

LEHMAN, N.M. and ROSENFELD, J.L. 1968
Performance of a Simulated Multiprogramming System.
Proc. AFIPS, Fall Joint Computer Conference, Volume
33, Spartan Books, New York, 1968. pp 1431.

LIPTAY, J.S. 1968
The Cache
IBM Systems Journal, Volume 7, Number 1, 1968
pp 15-21.

LYNCH, W.C. 1967
Evolution of Computer Operating Systems.
IEEE International Convention Record, Part 10, 1967.

LYNCH, W.C. 1972
Operating System Performance.
Comm. ACM, Volume 15, Number 7, July 1972. pp 579-585.

MARSAGLIA, G. and BRAY, T.A. 1968
One-Line Random Number Generators and Their Use in
Combinations.
Comm. ACM, Volume 11, Number 11, November 1968, pp 757.

MEEKER, J.W., CRANDALL, N.R., DAYTON, F.A. and ROSE, G. 1969
OS-3: The Oregon State Open Shop Operating System.
Proc. AFIPS, Spring Joint Computer Conference, Volume
34, Spartan Books, New York, 1969. pp 241-248

RANDELL, B. and KUEHNER, C.J. 1968
Demand Paging in Perspective.
Proc. AFIPS, Fall Joint Computer Conference, Volume 33,
Spartan Books, New York, 1968. pp 1011

ROLFSON, C.B. 1968
Scheduling Work Flow in a Computer System.
Ph.D. Thesis, University of Toronto, Dept. of Computer Science, 1968.

ROSE, J. 1967
Automation: Its Anatomy and Physiology.
Contemporary Science Paperbacks, 8, Oliver and Boyd. 1967.

SHERMAN, S., BASKETT, F., and BROWNE, J.C. 1972
Trace Driven Modelling and Analysis of CPU Scheduling in a Multiprogramming System.
Comm. ACM, Volume 15, Number 12, December 1972.
pp 1063-1069.

SHILS, A.J. 1968
The Load Leveller.
Report RC 2233, IBM T.J. Watson Research Centre, Yorktown Heights, New York, 7 October 1968

SIMON, H.A. and ANDO, A. 1961
Aggregation of Variables in Dynamic Systems.
Econometrica, Volume 29, Number 2, April 1961.

SMITH, C.L. 1970
Digital Control of Industrial Processes.
Computing Surveys, Volume 2, Number 3, September 1970. pp 211-242.

WEINGARTEN, A. 1966
The Eschenbach Drum Scheme.
Comm. ACM, Volume 5, Number 7, July 1966.
pp 509-512.

WHARTON, R.M. 1971

A Page Replacement Strategy for Multiprogramming Systems.
Topics in Operating Systems. Ed. Tsichritzis, T.
Technical Report Number 23, January 1971.
Dept. of Computer Science, University of Toronto.
pp 23-34

WILKES, M.V. 1971

Automatic Load Adjustment in Time Sharing Systems.
ACM Sigops Workshop on System Performance Evaluation,
Havard. pp 308

WILKES, M.V. 1973

The Dynamics of Paging.
Computer Journal, Volume 16, Number 1, February
1973. pp 4-9.

WULF, W.A. 1969

Performance Monitors for Multiprogramming Systems.
Second Symposium on Operating Systems Principles,
Princeton University, October 1969. pp 175-181.