# ESTIMATION OF PRIMARY MEMORY REQUIREMENTS OF PROCESSES IN MULTICS

by

DAVID PATRICK REED

Submitted in Partial Fulfillment

of the Requirements for the

Degree of Bachelor of Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June, 1973

Signature of Author David P. Reed
Department of Electrical Engineering, May 25, 1973

Certified by _____
Thesis Supervisor

Accepted by David Adler
Chairman, Departmental Committee on Theses

# ESTIMATION OF PRIMARY MEMORY REQUIREMENTS OF PROCESSES IN MULTICS

by

DAVID PATRICK REED

Submitted to the Department of Electrical Engineering
on May 25, 1973, in partial fulfillment of the
requirements for the Degree of Bachelor of Science

## ABSTRACT

This thesis presents a simple, effective algorithm for predicting the primary memory needs of in execution on a demand paging, virtual memory computer system such as Multics. The algorithm is based on linear extrapolation of the relation between page fault rate and allocated primary memory to determine the approprriate amount of primary memory needed to sustain a specified "optimal" page fault rate in each process. Applications of the algorithm to primary memory resource management in such computer systems is discussed. Experiments with an implementation of the algorithm in a special version of the Multics system are presented,with results that indicate that the algorithm has some promise to be useful in optimizing throughput.

THESIS SUPERVISOR: Jerome H. Saltzer
TITLE: Associate Professor of Electrical Engineering

## Table of Contents



## Figures and Tables

## I. Introduction

A simple adaptive control mechanism for predicting the primary memory demands of an ongoing computation in a virtual memory environment can be utilized in several ways to manage memory resources in a system such as Multics [C1,C2]. These uses can be classified according to purpose in three classes:

1. System performance optimization.
2. Enforcement of system management policy goals related to control of primary memory resources.
3. Providing information to programmers about their primary memory resource utilization.

In this thesis I will discuss an algorithm which can be used in all of these ways, and show the results of some experiments in which I have attempted to demonstrate the effectiveness of the algorithm in "real" system tests performed on a version of Multics.

## II. Details of the Problem

In the Multics virtual memory implementation, primary memory is divided into equal sized blocks known as pages. A computation's apparent address space is then mapped into a set of blocks of storage, which are moved into main memory upon demand. Due to the fact that a fairly large delay is incurred while waiting for transfers of pages from secondary to primary memory, the Multics

operating system takes advantage of page waits to run other processes (this technique is called multiprogramming). Since enough pages in the newly started process must be present in primary memory to run that process at all, and due to the finite size of primary memory, only a small subset of the processes ready to run at a particular time can be made eligible for multiprogramming at once. The number of processes thus made eligible is called the degree of multiprogramming.

In the current Multics implementation, the degree of multiprogramming is limited to a fixed number, which has been determined by experiment to give good system percentage throughput[1] and good average response time. It has been clear for a long time that this is not a sufficient mechanism -- particularly because the primary memory demands of individual compuations vary widely. For example, five processes performing editor requests do not generate the same demands on primary memory which five large data base inquiry systems do.

An algorithm which predicts the primary memory demands of a computation can be used to automatically control the degree of multiprogramming so that performance is optimized. In the current Multics implementation, in fact, there is an attempt to

---

[1] Percentage throughput is a measure of system effectiveness computed by determining what percentage of the processors' time is spent processing user computations, as opposed to automatic operating system management functions not directly requested by the user.

do this, by predicting a working set size for a computation, but the algorithm used for this prediction has never worked at all well. I will discuss some of its problems in a later section. The algorithm developed in this thesis is also an attempt to predict the primary memory needs of a process, and seems to work better than the already existing algorithm.

The first two purposes noted in section I, performance optimization and enforcement of policy goals are normally closely related. One normally talks about optimizing system performance within the constraints of a particular system management policy. Consequently, I would also like to briefly discuss the problems of enforcing system management policy as related to memory resource management. Policy goals can be distinguished from simple performance goals by the fact that policy goals are normally preferential -- i.e., certain resource usage patterns are encouraged while others are penalized. These preferences are usually based on considerations external to the computer system, such as the relative importance of certain types of computer usage at a particular installation.

Having an effective algorithm for predicting memory demands of computations allows certain system management policies regarding memory resources to be enforced. For example, processes with small memory needs might be given scheduling priority over processes with large memory needs. Another possibility is a

policy of allowing some users to obtain more use of primary memory than others -- this could be gotten by giving them more memory than the system determines as their fair share when they are scheduled. A particular management policy which I will take as necessary in the development of this thesis is the policy of equal percentage overhead for all classes of users. I have already noted the necessity of automatically controlling the degree of multiprogramming in order to prevent devotion of excessive resources to page fault processing. Control of multiprogramming is important from the operating system's point of view in order to maximize the effective use of the available hardware resources. Were this the only goal, however, the task would be much simpler; a very preferable policy goal for a multiprogramming control is to allocate primary memory resources fairly among user processes. It is easy to ignore the variation in usage patterns among user programs and design a scheme which works for the average programs on a particular system, and performs poorly for exceptional classes of users. The exceptions which I am concerned with now are the ends of the memory usage scale: the small users, such as the SIPB [1] Experimental Calculator Service, which offers a cheap, interactive subset of BASIC to all MIT students, and the large users, such as data-base

---

[1] The MIT Student Information Processing Board, which is a student organization at MIT dedicated to the purpose of supplying computer time to students in order to make it relatively simple for such students to become used to using the computer as an everyday tool.

management systems, automatic programming systems, and so on. A fair scheme for allocating primary memory resources should allocate primary memory to users in proportion to their need for primary memory, so that large users of memory will have enough memory to run with an acceptable level of overhead due to page faults. On the other hand, a small memory user should not be allocated more memory than is necessary to run him efficiently, for any excess resources might as well be assigned to others waiting in the scheduling queues to run.

In order to manage primary memory effectively, taking into account the variations among classes of usage, the traffic controller must be able to predict the memory requirements of processes and use this prediction in its scheduling decision. Unfortunately, until recently there was no easily tractable model of a process's memory reference behavior which could be invoked to assay a process's memory needs at a particular instant of time. Consequently, in the current design of the Multics primary memory scheduling algorithm, the control of multiprogramming is achieved by an algorithm which has been observed to show several anomalies (in particular, it performs very poorly for some programs with larger than average "working sets"). I believe, however, that Saltzer's linear model of demand paging performance[S2] does lend itself to a simple, easily implemented algorithm for dynamic multiprogramming control. For my thesis, I have developed such an algorithm, and determined experimentally

its performance.

## III. Historical Perspective

As far as I could determine, there is very little information in the literature which deals directly with the subject of dynamic control of the degree of multiprogramming in a demand-paged virtual memory system. There is, however, a great deal of information on closely related subjects such as the analysis and measurement of performance in such computer systems and the analysis of program behavior with regard to memory reference patterns. I survey here some of the more interesting literature in those areas which bear closely upon the topic under discussion in this thesis.

Smith [S4] gives a fairly primitive model of multiprogramming in a demand paging computer system. He speaks briefly about optimizing the level of multiprogramming to prevent overloading the I/O queues for secondary storage media. He does not discuss any means of dynamically adjusting the level of multiprogramming based on instantaneous program behavior. Wallace and Mason [W1] speak about obtaining optimal degrees of multiprogramming under a kind of "linear model". However, again no attempt is made to discuss dynamic control of the level of multiprogramming; Wallace and Mason discuss only static optimization of the level of multiprogramming. This paper also gives some attention to the

effect of the bursts of paging activity begun upon activating a computation.

Saltzer [S1], in his doctoral thesis, deals with some of the problems of scheduling processor and memory resources. The basis of the current implementation of the Multics traffic controller is discussed, although no really relevant information related to control of the degree of multiprogramming is discussed.

Sekino [S3], in his doctoral thesis, develops a comprehensive model of a demand paged, virtual memory computer system, which includes some discussion of static optimization of the degree of multiprogramming. For the case of fixed degrees of multiprogramming, he develops analytic models which closely approximate the behavior of the current implementation of Multics.

Saltzer [S2], in an internal Project MAC document, describes a simple linear model of the paging behavior of a demand paged computer system, in which he directly relates primary memory size and system page fault rate. Experimental results which indicate the validity of this model over a wide range of primary memory sizes are presented. He indicates that this might be generalized to cover individual processes, and shows how this model can be useful in predicting system performance.

Frankston [F1], in work on the Multics system, deals with a

method for allocating costs of primary memory among users competing for that memory by using Saltzer's linear model.

Hunt [H1], in work on the Multics system, has evaluated the predictions of Sekino's model with respect to optimal level of multiprogramming and found them to be fairly accurate. He also pointed out the fact that the currently implemented working set estimation algorithm fails to work to stabilize the paging behavior of the system. In doing this work he has developed a standard benchmark load for Multics, which I have used to conduct the experiments described in a later section.

## IV. Yardsticks for Evaluation of the Algorithm

Before discussing particular algorithms to achieve the purposes hinted at above, it seems important to enumerate several criteria which I think must be met by any primary memory scheduling algorithm. First, the variables most important to heavily loaded interactive systems, response time and system throughput, must be optimized. Second, the algorithm must properly respond to the variation in memory requirements among different classes of users -- not penalizing large users excessively, and giving the small user his fair share of resources. It is under this criterion that the current Multics memory management algorithm seems to be deficient. Thirdly, the algorithm must be stable under fluctuating load conditions. Although the algorithm is primarily used to prevent excessive

page fault overhead, a condition occuring only under a significant load, light loads should not cause the algorithm to operate incorrectly, and transient loads due to interactive processes should not cause the algorithm to oscillate, or otherwise perform suboptimally. Finally, the parameters by which the algorithm is tuned should be independent of system configuration as much as possible. In other words, once a system is tuned for one configuration, it should be correctly tuned for any other configuration. I am particularly interested in the configuration with respect to the size of primary memory and the number of CPU's. It is interesting to note that the current algorithm on Multics causes the following anomaly: if the parameters are set for a 1 CPU, 256K system, then the performance of a system such as a 2 CPU, 256K system is very poor. This probably means that the second CPU is causing thrashing, and should remain idle more of the time, since a second CPU should not cause degradation of system performance on the same amount of memory. [1]

---

[1] Barring such effects as memory interference and data-base interference, both of which can be minimized by idling the additional CPU more frequently.

## V. Simplicity

I devote a separate heading to this topic simply (!) because it is worth emphasizing. It is all too easy to design algorithms which are more complex than our understanding of the problems they intend to solve. Since the difficulty in maintaining programs is primarily due to the time required to understand them, simplicity is an operational virtue.

In the context of a discussion of primary memory management algorithms, simplicity has several applications. Our understanding of programs' memory reference behavior is limited, so it would be unjustifiable to assume that a complicated algorithm has any better chance of working well than a simple algorithm would. A simple algorithm is much easier to explain, thus allowing others who must understand the algorithm (such as specialized Multics installations who have need to tune their system for their own purposes) to achieve the necessary knowledge with a minimum of difficulty. Also, since the multiprogramming algorithm is a basic part of the operating system, a simple multiprogramming algorithm has the virtue of easier verification, an important consideration in a system striving to be provably correct.

## VI. Basic Principles

For most of this thesis, I will sketch the development of an algorithm based on Saltzer's linear model, which I hope will satisfy the above criteria, along with the criterion of simplicity. Towards the end of the thesis, I will discuss the techniques which I used to investigate the performance of the algorithm experimentally.

Before I begin describing the details of my proposed algorithm, I would like to note the basic concepts of the current multiprogramming control mechanism. First of all, since we are primarily concerned with obtaining effective use of main memory, the degree of multiprogramming depends directly on the size of primary memory available for paging (I will henceforth use the symbol M for this). The number of processors is not germane to this discussion, since addition of a processor has the effect of increasing the rate of references to memory, which effect will be factored out by considering time as an axis measured in memory references.[1] In order to characterize a process's memory requirements, the currently implemented algorithm attempts to compute a value, called the working set estimate, which indicates the amount of primary memory which must be available to that

---

[1] Of course, I oversimplify here...in fact there are important effects such as sharing of pages and overloading of I/O channels which do come into play in determining the load on primary memory. For a first order conceptual model, I will explicitly ignore these effects.

process to prevent excessive paging. This is basically computed from the size of a subset of the recent page exceptions (those which correspond to pages which have been referenced recently, according to the hardware and software reference bits).

The scheduler then makes processes eligible until the sum of their computed working set estimates is as large as possible, but not exceeding the amount of primary memory available. An additional constraint is added which places lower and upper bounds on the number of eligible processes. This constraint is present to counteract the inconsistency of the results provided by the working set estimator, and in fact the major control of the degree of multiprogramming is currently obtained by the setting of the upper bound of eligible processes.

In practice, the working set estimation algorithm has not been made useful, as is shown by the fact that the computed working set estimate is multiplied by a fraction, called the working set factor, before summing the eligible processes' working set estimates, and that this fraction has in practice been set to a value which is too small to allow the working set estimates to influence the scheduling decision at all (increasing the working set factor causes performance to degrade significantly -- indicating the algorithm does not calculate a working set estimate which is useful in scheduling).

This algorithm has been observed to have several deficiencies.

The first is poor parameterization, which does not carry from one configuration to another. More importantly, the working set estimate has shown some obvious incorrect behavior for certain programs with large working sets. The symptom of this is that the working set estimate remains very small even though the user's program is taking over 100 page faults per cpu second (a mean headway between page faults of several instructions). This has been observed by people in the LISP project[1] and by Gumpertz[G1] at the Multics installation at Honeywell Bull, Paris, France.

Finally, the currently implemented algorithm fails on the criterion of simplicity. It bases its decision of whether to include a page in the working set on a complicated calculation based on 5 bits of information about each recent page fault. The time eligible is not taken into account, and page faults which are too frequent to be recorded in the page fault trace table are not taken into account. In addition, there are complicated interactions with other processes, the core allocation algorithm, and the system call, hcs_$reset_working_set,[2] all of which

---

[1] A project, currently in progress, to implement a version of the LISP programming language on the Multics system. This version of LISP is specifically designed to handle large Data Base applications, requiring large amounts of primary memory to operate efficiently.

[2] This call is supposed to allow the user some control over his paging performance. It allows him to notify the system of large discontinuities in his page reference pattern.

modify the page usage bits which the working set algorithm depends on.

## VII. The Proposed Algorithm

I have tried a somewhat different approach in synthesizing an algorithm here, with the hope that the resulting algorithm will be simpler, and more amenable to analysis and verification. I have also tried to define a set of less ambiguous terms for describing concepts, since such terms as "working set estimate" are apt to be confusing, because they have many possible interpretations in current usage.

The following analysis is not intended to be mathematically precise; instead, it is intended only as an argument to justify the plausibility of the resulting algorithm. The correct analysis would depend on elaborate statistical analysis of the algorithm's interaction with a complex model of general program behavior. Since such sufficient models do not really exist, this type of study is beyond the scope of this thesis.

To begin the discussion of my proposed algorithm, I would like to define a function, p(h,t), which is called the partition size of a process with respect to a particular mean headway between page faults (mhbpf, measured in memory references), h, at a particular time t. This partition size function is defined to be the amount of primary memory required for that process to run at time t in

its execution with a specified local mhbpf, h. Given this function for all processes, a near-optimal multiprogramming level can be determined.

One way of preventing degradation of each process's performance due to excessive page faulting is to place a lower bound on each process's mhbpf, h_opt. Given h_opt, a near optimal multiprogramming strategy is to schedule as many processes as possible under the constraint that the sum of their instantaneous partition sizes is $\leq$ M:

$$\sum p(h\_opt,t) \leq M.$$

Of course, if the load is sufficiently heavy, the inequality becomes an equality, and the page fault rate is held to the level determined by the mhbpf, h_opt. On the other hand, if there is not a sufficient load, the mhbpf for each process will be greater than h_opt. Of course, fixing h to h_opt does not guarantee that we will prevent bursts of excessive page faults; this depends also upon how steady p(h,t) is with respect to small changes in t.

The problem of implementing this strategy is that p(h_opt,t) cannot be computed without foreknowledge of the process's behavior. Consequently, a practical algorithm must use a function which is capable of estimating p(h_opt,t). The working set estimate above can be seen to be an attempt at this, however, it

is not a very effective estimate for the reasons noted above. In the following analysis I will attempt to create an estimator for p(h_opt, t) which is better at tracking the actual value of p(h_opt,t) than the working set estimate currently used.

Saltzer's linear model[1] states that p(h,t) can be approximated closely by a linear function of h:

$$p(h,t) = k(t)*h,$$

where k is a function only of the program behavior around t. As a consequence, if we know the value of p for some specified value of h, h0, we can determine p for all values of h:

$$p(h,t) = \frac{h}{h0} * p(h0,t)$$

Figure 1 illustrates this model of partition size behavior. In the figure, the solid curve represents the actual relation of p(h,t) and mean headway between page faults. The dashed line is the linear approximation to this curve obtained by taking the actual partition at time t, p(h0,t), and the observed mean headway between page faults, h0, and constructing a line from this point through the origin. The approximation to p(h_opt, t)

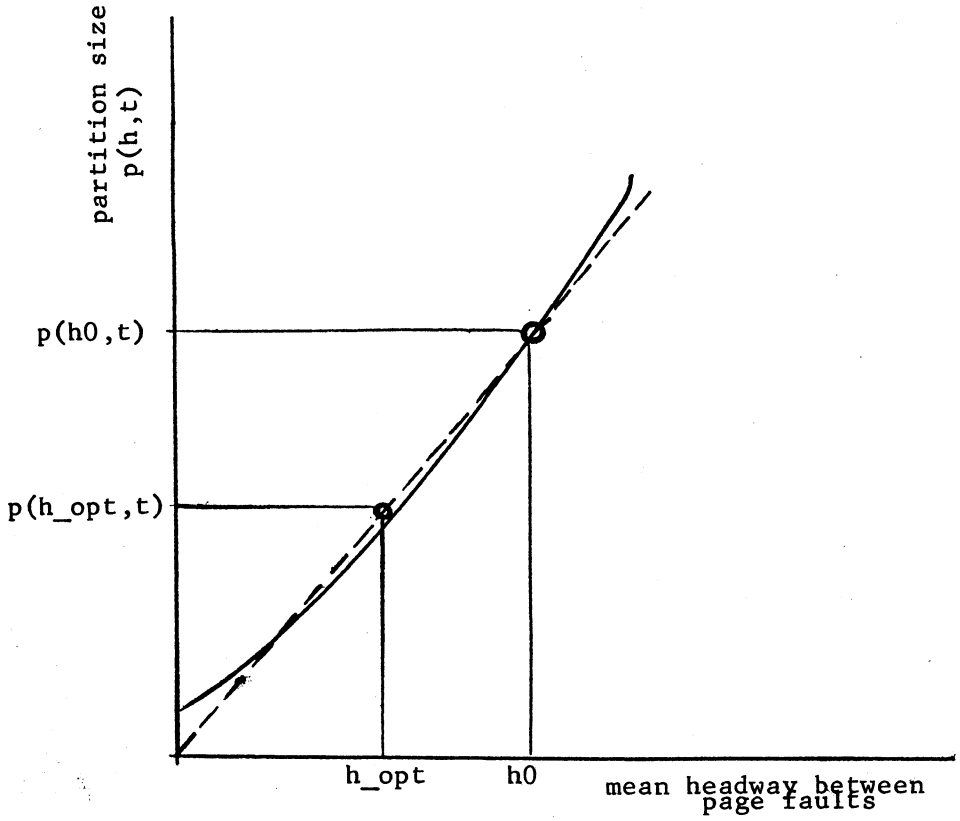
---

[1] I am assuming that Saltzer's model can be applied to single processes, an assumption which is plausible but not yet experimentally verified. The linear model does not have to apply exactly in order for this scheme to work, but I develop the argument from this statement of the linear model for the purpose of conceptual clarity.

can then be obtained by interpolating to the required mean headway between page faults on this line, as shown in the figure.

Figure 1

Using the Linear Model to Estimate p(h_opt, t)




In particular, assuming that k(t) has a bounded first derivative, and that we have the value of p(h0,t), we can get a good estimate for p a short time later by assuming that k(t) does not change significantly over that short period:

$$p(h,t+dt) = \frac{h}{h0} * p(h0,t)$$

It can easily be seen that this last equation can be used as the basis for a rather simple multiprogramming control. If we take p(h0,t) to be the partition size which we last allowed the process to run with, and h0 to be the observed mean headway between page faults for that period, than we can easily obtain an estimate for the immediate future of that process of p(h_opt,t) by that equation, setting h = h_opt. We thus have the following iterative algorithm which tracks the partition size of a process:

$$pe = \frac{h\_opt}{h\_obs} * pa \qquad (A)$$

where pe is the new estimate of the required partition size of the process for an mhbpf of h_opt, h_obs is the observed recent mhbpf, and pa is the amount of main memory which was assigned to the process during the recent past (over which we measured h_obs).

This algorithm is actually more powerful than the assumptions indicate. Even if Saltzer's linear model does not hold for a single process, this iterative algorithm will track p(h_opt,t) as long as the rate of change of p with time is small, and p is a relatively well-behaved function.

The only problem with implementing this algorithm is that it is somewhat difficult to determine a value for pa, the amount of memory actually assigned to the process, since several processes are competing for main memory, and pages do not inherently belong

to particular processes because of sharing. Under a sufficient load, of course, the constraint that the scheduler tries to make as many processes eligible as possible will force pa to be approximately equal to the previous estimate of pe. However, under lighter or more transient loads, this will not be the case. Since we would wish pe to be relatively independent of load fluctuations, we have to handle this case, and also the case in which for some reason we are forced to multiprogram more processes than will effectively fit into primary memory. This latter case could occur for individual processes whose partition sizes are too large to fit in main memory, or under some policy decision which requires a minimum number of eligible processes at all times.

Under a light load, the scheduler will not have enough processes to wholly utilize main memory resources. Consequently, each process's pa > pe. When there are processes whose partition sizes, pe, do not fit in primary memory, it is possible to have situations where a process's assigned memory, pa, is less than pe. This latter condition can also arise from constraints, such as the current lower and upper bounds on the number of processes the scheduler can make eligible. In order to keep the partition size estimates, pe, stable through transient loads and independent of time, it is thus important to get an accurate estimate of the memory assigned to the process to compute new

values of pe.

Making the assumption that the effective partitioning of memory among the eligible processes is proportional to their partition size, we can say then that

$$pa = \frac{pe}{\bar{u}} \qquad (B)$$

where $\bar{u}$ is the average fullness of primary memory. This fullness is determined by the function u(t), which is the ratio of the sum of all eligible processes' partition size estimates, and the size of main memory:

$$u(t) = \frac{\sum_{i \in \{\text{eligible processes}\}} pe(i,t)}{M}$$

This is basically a primary memory usage factor. Combining equations A and B, we get the iterative algorithm for computing the next pe value, pe':

$$pe' = \frac{h\_opt * pe}{h\_obs * \bar{u}}$$

I have not yet specified over what sample $\bar{u}$ is an average of u. For this purpose, an approximation to the average will suffice, since the value is not particularly critical. Consequently, the system will sample the memory usage fraction at each page fault

since the previous computation of the partition size estimate, computing a running average over time.

For the purposes of my experiments, the computation of pe is iterated each time the process can lose eligibility, since the value is needed no more frequently than that.

## VIII. Parameterization

Parameterization of h_opt and pe_init is a more difficult problem. Determination of h_opt's relationship to memory size is really a policy decision. If we wish the level of overhead seen by individual processes to reflect the added memory, then setting h_opt proportional to memory size,

$$h\_opt = ch * M$$

where ch is a tuning parameter, will have the desired effect. On the other hand, system management might want added memory to allow the support of more processes, while maintaining the same level of paging overhead, in which case, h_opt should be set independent of primary memory capacity. Some strategy in between these extremes should be used,

$$h\_opt = ch * M + ch1$$

where ch and ch1 are tuning parameters. ch1 can be viewed as a basic minimum mean headway between page faults, while ch represents a measure of the rate of improvement of individual processes' performance as the system configuration increases in

size.

The most important thing the algorithm offers here is that it can be used to control the paging rate of the system over a fairly wide range of values, thus effectively controlling the paging overhead seen by the system as a whole.

In contrast, pe_init is much more easily parameterized, since it just reflects as estimate of the average process's partition size. We can thus easily compute pe_init from the optimal mean headway between page faults by the linear model:

$$pe\_init = cp * h\_opt$$

where cp is a tuning parameter. It could be possible to set cp from a global long term average of system page fault behavior, but I could not test this out very well in the time allotted for the thesis.

An important factor in the parameterization of h_opt is the limiting I/O channel capacity for page transfers. An optimal policy might be chosen to maintain the peak channel demands at a sufficiently low level to prevent frequently exceeding the maximum channel capacity.

## IX. Implementation of the Algorithm

In order to test the partition size estimation algorithm detailed in the last section, I have implemented it in a special version

of Multics hardcore system 18-11. This system runs on the Honeywell 645 processor, and does paging I/O from a multi-level secondary storage hierarchy consisting of a "firehose drum" and several kinds of slower disk storage. Several modifications to the software were required, most of which were simple clerical changes. The interesting software changes will be detailed here.

First of all, the pre-paging mechanism was removed from the system. This algorithm attempts to guess which pages will be needed by a process which is being made eligible for multiprogramming, and causes I/O to be initiated to bring those pages into primary memory. I removed the mechanism for two reasons. First of all, the pre-paging mechanism is obsolescent in Multics since it is quite dependent on the characteristics of the "firehose drum" for its operation. Secondly, since the pre-paging algorithm is yet another algorithm trying to guess the primary memory needs of a process, its operation would very likely affect the operation of my partition size estimator very strongly. For these reasons, I felt that the work involved in trying to take pre-paging into account in the implementation of the algorithm was not justified.

The second change made was to modify the Multics traffic controller[1] to compute the partition size estimate at each time quantum, and to use the partition size thus computed to determine

---

[1] See Saltzer's paper on traffic control[S1].

if the process can be made eligible for multiprogramming. The algorithm thus implemented is a slightly rewritten form of the algorithm described in section VI. The one particular modification made is to count the end of the time quantum as a page fault. By doing so, two important goals are accomplished: first, the limiting case in which no page faults are taken during the time quantum is made to work (h_obs would be undefined otherwise), and second, the likelihood of overestimating h_obs with consequent gross underestimation of pe is reduced. This last can result from the large probability of error in estimating mhbpf caused by the relatively small sample of page faults.

The page fault handler was modified to add code to compute the running average of memory fullness, $\bar{u}$, on each page fault. This average is kept on a per process basis to save problems with attempting to use a system value which might not quite correspond to the time quantum under observation.

Other changes were made in system initialization and process initialization to set up the algorithm in each process. Some metering commands were changed.

## X. Tests of the Algorithm

For my thesis, I investigated the characteristics of this algorithm experimentally, using Hunt's standard Multics load programs[H1], and compared performance with the current

multiprogramming algorithm. The basic test performed was the comparison of fixed degree of multiprogramming as a multiprogramming control, as in the current Multics system, and the automatic control based on the partition size estimator. By comparing the stability of the paging rate in both tests, and the system throughput and response times under both sets of conditions, a reasonable estimate of the feasibility of the proposed partition size estimator is obtained.

The first tests performed were a series of tests of the consistency of the partition size estimates under varying loads. This was accomplished by running various commands with standard data repeatedly, and obtaining the partition size estimate at the time of completion of the command. The commands used were the following three:

pl1 get_ps -- which was a small PL/I compilation.

submit_abs_request -- which touches a fair amount of data in its operation.

lisp -- the lisp subsystem was entered and then immediately exited.

The results of these tests seem to indicate that the estimator is pretty consistent. Table I gives these results in tabular form. More extensive tests would be desirable, but these tests seem to be sufficient to prove the effectiveness of the estimator.

Table I

| Test: | pl1 | submit_abs_request | lisp |
|---|---|---|---|
| Number of samples: | 7 | 7 | 3 |
| Mean partition size: | 271.6 | 109.1 | 26.0 |
| Standard Deviation: | 160.0 | 8.4 | 1.0 |

The results of the PL/I test are somewhat suspect here because the PL/I command takes a fairly long time to complete. The history of the partition size is thus more prone to being upset by variations in the size of the last "quantum" of time assigned by the system. Since I am interested in the value during the last quantum in the results noted above, this may explain the large variability of the partition size. This large variability is accounted for almost entirely in one particular extreme sample.

The second set of tests consisted of running a standard, light-to-medium benchmark load on a large hardware Multics configuration. The partition size algorithm was turned off, by setting h_opt to zero, and control over the degree of multiprogramming was maintained by the normal means of limiting the number of eligible processes to a number, max_eligible. This number was varied to obtain a number of sample experiments, and system throughput and page fault rate were measured, to obtain a

graph of system performance related to max_eligible. After completing this first phase of the experiment, max_eligible was set to 10 (a very large, essentially infinite value), and h_opt was varied to obtain a second graph relating system performance and h_opt. The graphs are presented in Figure 2.

Figure 2

System Performance under Each Multiprogramming Control

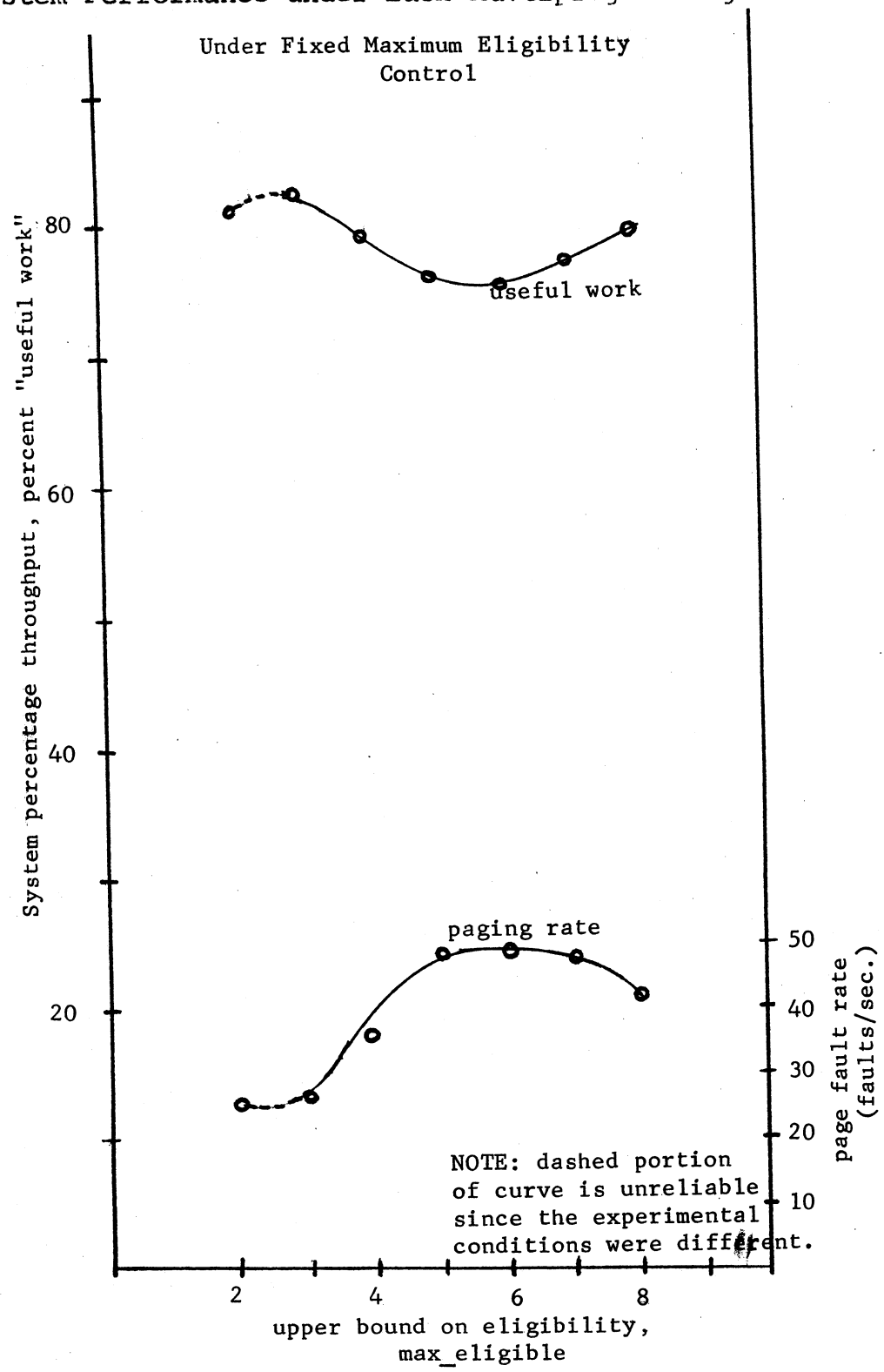
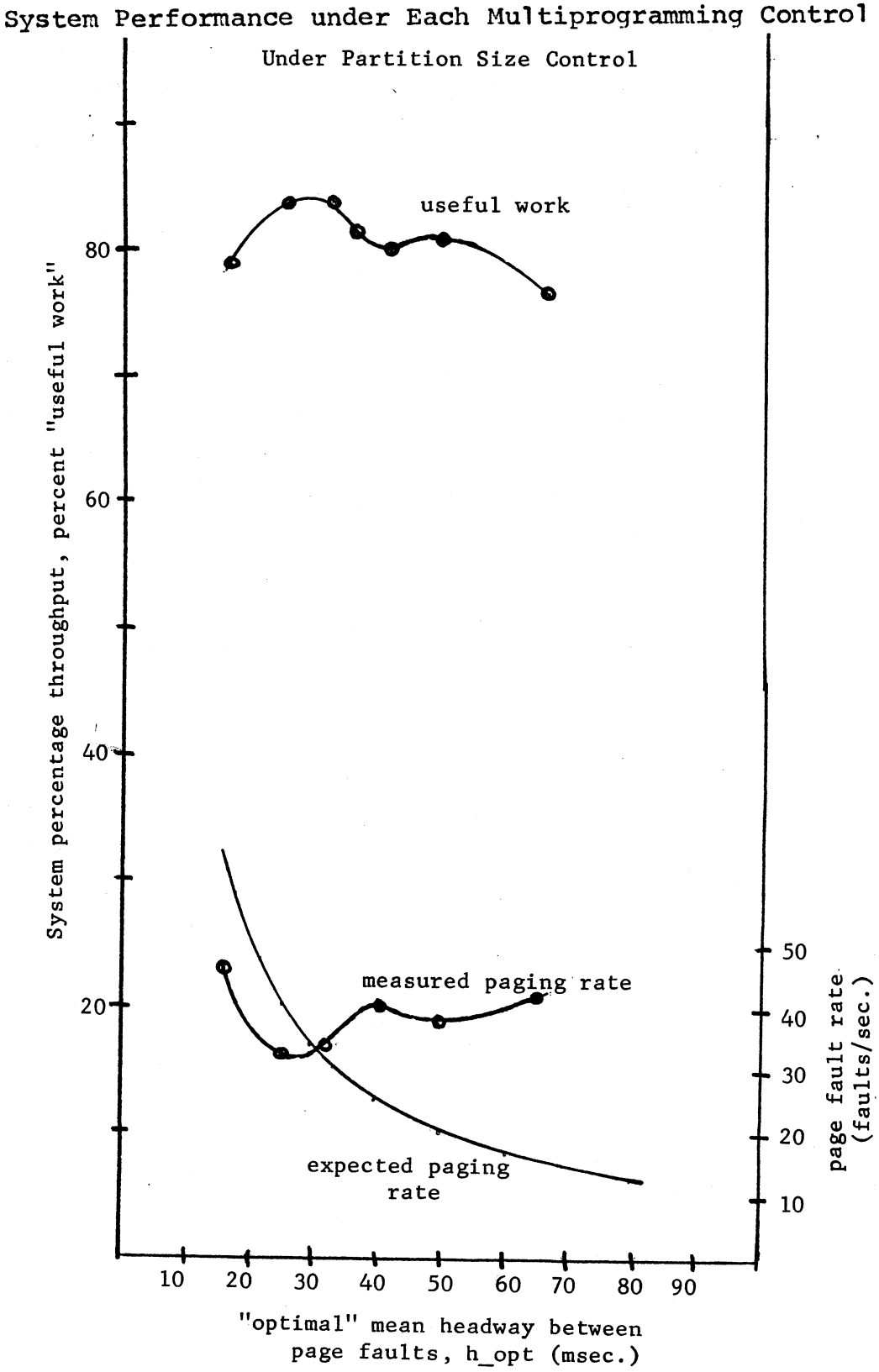


Figure 2 (cont.)



I must admit that the results given by the graphs are somewhat confusing. The graphs do reflect system performance pretty well, as the variation in system performance over the period of a particular setting of h_opt or max_eligible is approximately 2-3%. Each test lasted for 10 minutes, with a sample of the page fault rate and throughput taken each minute.

In Figure 2, there is a noticeable improvement in the "optimally tuned" system under the partition size control of about 5%. This performance improvement is just above the level of noise, but does seems significant, especially since the benchmark load programs do not include much variation in their demand on primary memory.

The page fault rate graphs do not seem to reflect my model however. In particular, though the page fault rate under the partition size control seemed to be slightly more stable, it did not stabilize to the rate prescribed by h_opt. More experiments will be necessary to determine why this is not so.

## XI. Summary and Comments on Further Work to be Done

In this thesis, I have presented a justification for a simple, effective estimator of the partition size or primary memory "competitiveness" of a computation. The algorithm proposed has been discussed from a theoretical viewpoint, and some experiments

with an implementation of the algorithm have been tried and noted here. The success of the implementation must be qualified by the fact that certain of the experimental data seem to be unexplainable; in particular, the stabilization of the page fault rate seems to occur at the wrong value.

Since these tests have not been a total failure, I think it will be interesting to find out if it is possible to understand why the results obtained did occur, and perhaps modify the implementation of the algorithm slightly to make it work better.

A test which was not carried out due to time limitations was one in which the benchmark load was changed to include processes with extremely light and extremely heavy demands on memory. Presumably, the partition size control algorithm will optimize system performance significantly in this case.

A major stumbling block in implementing the algorithm for the new version of Multics to be run on the Honeywell 6180 system is the fact that the primary paging device used is so fast that it is not worthwhile to multiprogram while doing I/O to that device. Since this will drastically reduce the amount of multiprogramming being done, the conditions under which the partition size algorithm operates will be quite different from those on the Honeywell 645 Multics. Tests of the operation of this algorithm on such a system will have to be done before anything can be known about the performance change obtainable by using a

partition size control algorithm.

Another set of tests which seem to be worthwhile would be an attempt to determine exactly how the system reacts to various types of memory usage, from the user's viewpoint. In these tests, the percentage overhead incurred by various classes of memory usage in computations would be compared, under the current max_eligible control, and then under a partition size control. It is hoped that the partition size control would reduce the inherent discrimination against large memory requirements which is currently incorporated into the system.

The estimator as described here is potentially capable of generating extreme values of partition size estimates on occasion. The primary cause for this behavior is a too short time quantum used in calculating the recent mean headway between page faults. This can cause a few closely spaced page faults to indicate a rather high paging rate, thus making the algorithm believe that the process needs a large partition to operate efficiently. To partially counteract this effect, some kind of damping could be used; for example, one might take the average of the last two estimates to be the next estimate used in scheduling the process. Carrying this to an extreme would reduce the effectiveness of the algorithm drastically, however, since the estimate would not really respond well to changes in the memory requirements of the program.

## BIBLIOGRAPHY


C1 Corbató, F. J. and Vyssotsky, V. A., "Introduction and Overview of the Multics System," Proc. AFIPS FJCC 27, 1965, pp. 185-196.

C2 Corbató, F. J., Saltzer, J. H., and Clingen, C. T., "Multics--The First Seven Years," Proc. AFIPS SJCC 40, 1972, pp. 571-583.

F1 Frankston, R. M., and Saltzer, J. H., internal memoranda.

G1 Gumpertz, Richard H., personal communication.

H1 Hunt, D. H., internal Project MAC memo.

S1 Saltzer, J. H., "Traffic Control in a Multiplexed Computer System," Sc.D. Thesis, MIT Project MAC Report MAC-TR-30, July 1966.

S2 Saltzer, J. H., "A Simple Linear Model of Demand Paging Performance," submitted to Comm. ACM.

S3 Sekino, A., "Performance Evaluation of Multiprogrammed Time-Shared Computer Systems," Ph.D. Thesis, MIT Project MAC Report MAC-TR-103, September 1972.

S4 Smith, J. L., "Multiprogramming under Page on Demand Strategy," Comm. ACM, Vol. 10, No. 10, Oct. 1967, pp. 636-646.

W1 Wallace, V. L., and Mason, D. L., "Degree of Multiprogramming in Page-on-Demand Systems," Comm. ACM, Vol. 12, No. 6, June 1969, pp. 305-308.