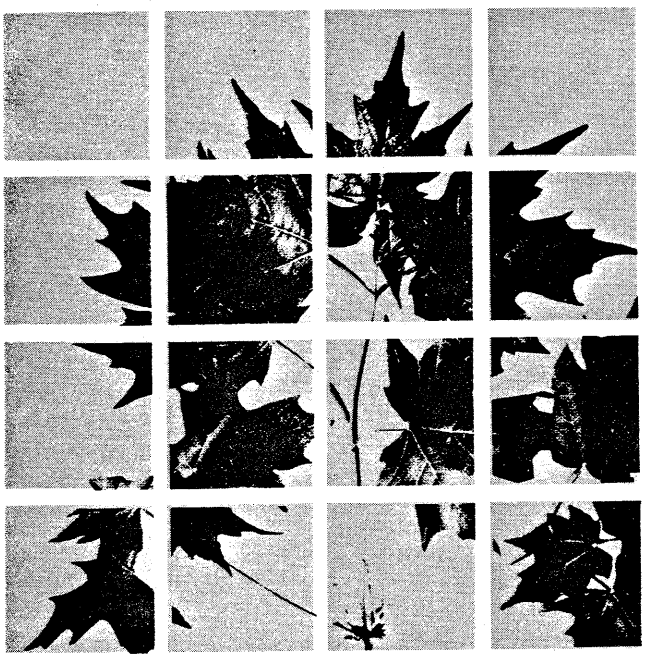


Program Behavior and the Page-Fault-Frequency Replacement Algorithm

Wesley W. Chu
University of California, Los Angeles

Holger Opderbeck
Telenet Communications Corp.

Introduction

Virtual memory is one of the major concepts that has evolved in computer architecture over the last decade. It has had a great impact on the design of new computer systems since it was first introduced by the designers of the Atlas computer in 1962. A virtual memory is usually divided into blocks of contiguous locations to allow an efficient mapping of the logical addresses into the physical address space. In this paper, we are concerned with paging systems, that is, systems for which the blocks of contiguous locations are of equal size. The memory system consists of two levels: main memory and auxiliary memory. The occurrence of a reference to a page that is currently not in main memory is called a page fault. A page fault results in the interruption of the program and the transfer of the referenced page from auxiliary to main memory.

Since main memory has only a limited capacity, pages already in main memory must continually be removed to make room for pages entering from auxiliary memory. Decisions as to when and what pages are to be removed from main memory are critical for the efficient operation of the system. The replacement algorithm is that part of the system which makes those decisions. The objective of a replacement algorithm is twofold. First, it is to keep those pages in main memory that are currently being used. This is necessary to keep the page fault frequency as low as possible. Second, the replacement algorithm is to free page frames as soon as there is a low probability that they will be referenced in the near future. This is a requirement for the efficient utilization of main memory by all processes.

In this paper, we first describe the behavior of several measured programs in terms of their stack distance probabilities and program reference patterns. Next, we describe the page-fault-frequency (PFF) replacement algorithm and its performance when operated in a uniprogramming and a multiprogramming environment. Finally, the implementation of the PFF replacement algorithm is discussed.

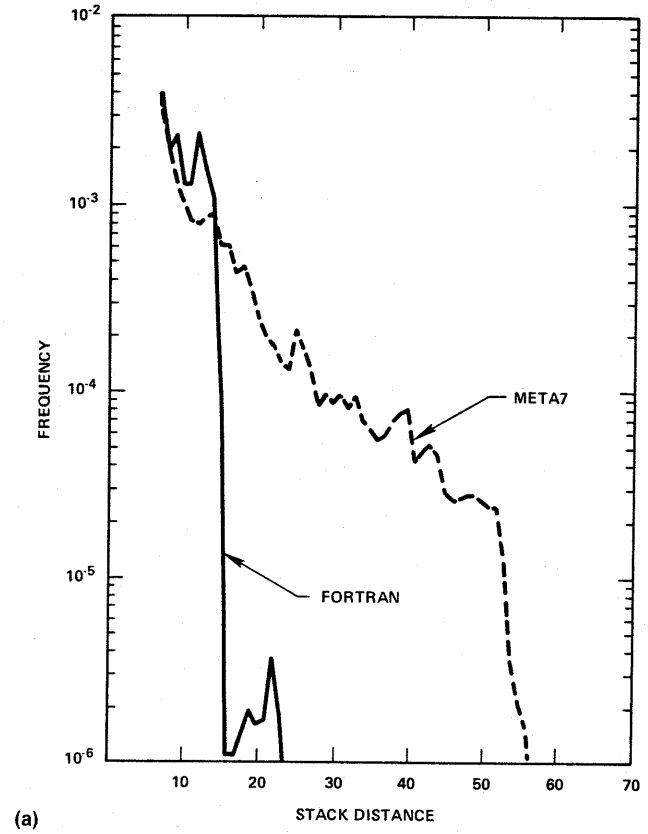
Measurement of program behavior

In order to measure dynamic program behavior, an interpreter was developed for the UCLA Sigma-7 timesharing system. This interpreter is capable of executing Sigma-7 object programs by handling the latter as data and reproducing a program's page reference string. The page size of the Sigma-7 timesharing system is 512 32-bit words. The page reference string can then be used as input to programs which simulate various types of replacement algorithms. For convenience in presentation, we let the time required for 1000 page references correspond to one millisecond.

Four different programs of various characteristics were interpretively executed. A Fortran program and a Fortran compiler (Fortcomp) were chosen as representatives for programs with small localities.* A Meta7 compiler and a DCDL compiler represent programs with large localities. Meta7 translates programs written in Meta7 to the assembly language of the Sigma-7. The DCDL (Digital Control and Design Language) is written in Meta7. It translates specifications of digital hardware and microprogram control sequences into machine code. To illustrate the behavior of these programs, Figures 1a and 1b display the stack distance frequencies as defined by Mattson et al.[1] The frequent occurrence of large stack distances (20 and more) for Meta7 and DCDL indicates that the localities for these programs are larger than the localities of Fortran and Fortcomp.

Table 1 shows some characteristic properties of these programs. The column "size" is divided into two parts. "Static" refers to the number of pages s_0 necessary to store the program as an executable file on a disk where one page consists of 512 32-bit words. "Dynamic" indicates the number of different pages r_0 actually referenced while processing the given input data. There are two reasons why r_0 is not equal to s_0: first, not all the

*_Locality_ is defined as a subset of program's segments (in pages) which are referenced during a particular phase of its execution.



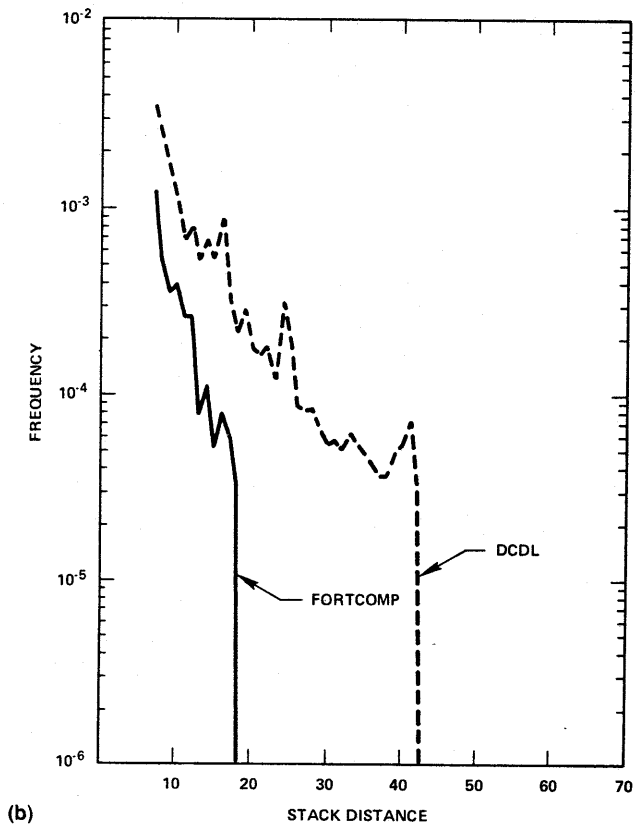


Figure 1. Stack distance frequency for the four measured programs: (a) Fortran and Meta7; (b) Fortcomp and DCDL.

TABLE 1. Characteristics of measured programs.

PROGRAM	SIZE		NUMBER OF PAGE REFERENCES
	STATIC s_0	DYNAMIC r_0	
FORTRAN	24	38	4,870,000
FORTCOMP	24	39	3,810,000
DCDL	44	71	3,010,000
META7	84	165	2,590,000

pages which make up the program may be referenced while processing a particular set of input data; second, a number of data pages are created and accessed during execution to provide for working storage space, buffer areas, etc.

The number r_0 is of special interest because it is equal to the minimal number of page faults which will be incurred by every replacement algorithm based on demand paging. Actually, r_0 page faults will occur even if not a single page is replaced. In this case, all page faults are caused by the very first reference to a page. Figures 2a-d display the reference patterns of the sample programs. The horizontal axis represents virtual processing time measured in units of 50,000 page references, while the vertical axis represents the virtual memory space at 512-word resolution. The dark areas show what pages have been used during a given time interval. This reference pattern illustrates the sudden change of program behavior as observed during the execution of the program. We notice that the Meta7 compiler has more sudden changes of referenced pages than the other three measured programs.

The page-fault-frequency replacement algorithm

Description of the PFF algorithm.[2] An "ideal" replacement algorithm should not make use of prior knowledge about program behavior; instead, all of the information needed to assure efficient memory allocation should be gathered during program execution.

The page-fault-frequency algorithm uses the measured page fault frequency as the basic parameter for the memory allocation decision process. In general, a high page fault frequency indicates that a process is running inefficiently because it is short of page frames. A low page fault frequency, on the other hand, indicates that a further increase in the number of allocated page frames will not considerably improve the efficiency and, in fact, might result in waste of memory space. Therefore, to improve system performance (e.g., the space-time product) one or more page frames could be freed.

The basic policy of the PFF algorithm is: whenever the page fault frequency rises above a given critical page fault frequency level P, all referenced pages which were not in the main memory—therefore causing page faults—are brought into the main memory without replacing any pages. This results in an increase in the number of allocated page frames, which in turn usually reduces the page fault frequency. On the other hand, once the page fault frequency falls below P, those page frames which have not been referenced since the last page fault occurred are freed. The same operation will be repeated whenever the page fault frequency rises above P again. Thus the PFF replacement algorithm provides very fast response to a sudden increase or decrease in memory

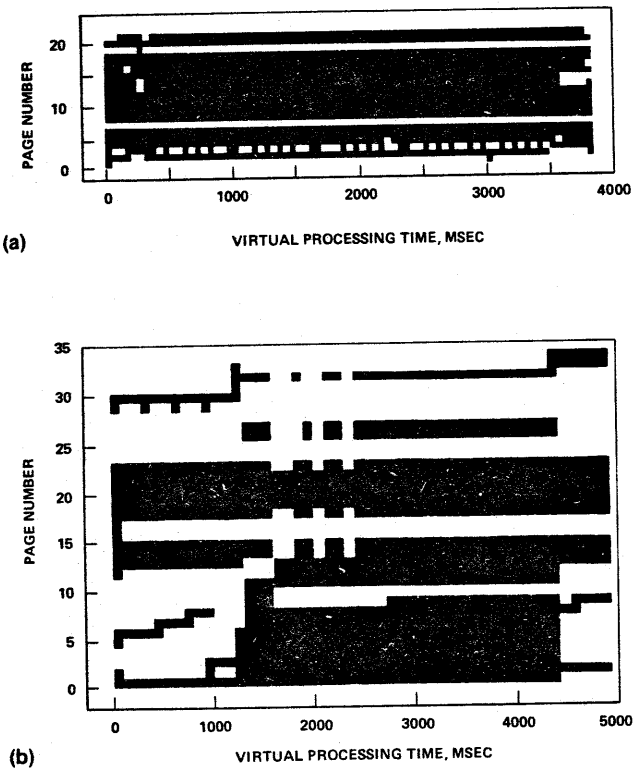


Figure 2. Reference patterns of the (a) Fortcomp program, (b) Fortran program, (c) DCDL program, and (d) Meta7 program.

requirements. The PFF parameter $P = 1/T$ is measured in the number of page faults per millisecond (1 msec = 1000 page references), where T is the critical interpage fault time.* The reciprocal of the interpage fault time is used as a running estimate of the page fault frequency. We note that T is somewhat similar to the working set parameter τ. However, there is an important difference. While τ indicates when a page should be freed, T represents only a lower limit. Furthermore, in contrast to the working set algorithm,† [3] page frames in the main memory are only freed at the time of a page fault. Therefore, the time at which a page frame is freed depends not only on T but also on the page fault frequency. The PFF replacement algorithm may therefore be considered as a working set algorithm with variable τ, where the value of τ is determined by the page fault frequency and the lower limit of τ is $T = 1/P$. The PFF algorithm may also be viewed as a LRU replacement algorithm with variable size memory allocation where the size is determined by T and the interpage fault times.

Performance of the PFF algorithm.

Performance measures. The page fault frequency and the space-time product are commonly used measures for the study of the performance of replacement algorithms. In the following we shall define these performance measures.

For a given page reference string ω and a given replacement algorithm the page fault frequency $f(\omega)$ is defined as the ratio of the number of page faults during processing ω to the total number of references in ω:

$$f(\omega) = r/t\,,$$

*A variation of the PFF algorithm is to use several neighboring interpage fault times to estimate the current page fault rate. Because of the adaptive nature of the PFF algorithm, our study of this modified scheme did not show improvements in performance over the PFF algorithm reported here.

†This is, of course, only true for the strict implementation of the working set algorithm which has been simulated.

where r is the total number of page faults and t the total number of page references.

Another parameter to measure performance is the space-time product which can be considered as being proportional to the cost of storage. Belady and Kuehner[4] define the space-time product C during the real-time interval $(0, t)$ as

$$C = \int_0^t S(z)\, dz\,, \tag{1}$$

where $S(z)$ is the amount of storage occupied by the process at time z. The real-time occupancy of information in main memory can be much longer than the virtual processing time. This occurs because of multiple processes being multiprogrammed and because of page-wait times. In a uniprogramming environment, only page-wait times need to be considered.

If we consider the execution of a program as a discrete process, the integral in (1) can be replaced by a sum which consists of two parts. The first part is the space-time product due to the virtual processing, while the second part is due to the total page-wait time. Thus the space-time product C can be rewritten as

$$C = \sum_{i=1}^{t} S_i T_{\mathrm{m}} + \sum_{i=1}^{r} S_{t_i+1} \cdot R \cdot T_{\mathrm{m}}\,, \tag{2}$$

where t is the total number of references; r the total number of page faults; S_i the number of allocated page frames prior to the ith reference (i is called the number of the reference); T_{m} the access time of main memory or the time of one page reference (10^{-3} msec); t_i the number of the reference which causes the ith page fault (since we do not preload any pages $t_i = 1$); S_{t_i+1} is therefore the number of page frames which are allocated during the ith page-wait time; and R is the speed ratio of a particular combination of auxiliary and main memory.

Since $\Sigma_{i=1}^{r} S_{t_i+1} \cdot T_{\mathrm{m}}$ and $\Sigma_{i=1}^{t} S_i T_{\mathrm{m}}$ are independent of R, C is a linear function of R. If we know C for $R = 0$ and C for another R ($0 < R < \infty$), then we can compute $\Sigma_{i=1}^{r} S_{t_i+1} \cdot T_{\mathrm{m}}$ from (2), and thus C for any R for $0 < R < \infty$.

Performance of the PFF algorithm in a uniprogramming environment. Measurement results from simulation of the PFF Algorithm for four different programs reveal that the performance (in terms of space-time product) of this algorithm is better than the performance of the best LRU replacement algorithm[1] (for which the optimal memory allocation that yields minimum space-time product is known *a priori*), and is comparable to the working set replacement algorithm (Figures 3a and 3b).[2] Further, the performance is relatively insensitive to changes in the PFF parameter P as shown in Figure 3b.

In Figure 4, the number of allocated page frames is displayed as a function of virtual processing time. As can be seen, the memory allocations for the four programs are quite different, and the number of allocated pages varies during execution, particularly for Meta7 and Fortran. This clearly demonstrates the adaptive capability of the PFF algorithm. The area below the four curves corresponds to the space-time product due to virtual processing time. For simplicity in representation, only major changes in the number of allocated page frames are indicated in Figure 4. Nevertheless, the figure shows clearly that the majority of page faults which resulted from changes in program locality occurred during relatively short time intervals.

Performance of replacement algorithms with different page sizes.[5] To study the effect that changing the page size has on the performance of a replacement algorithm, we again use reference strings as input to the programs which simulate various types of replacement algorithms. For the recording of reference strings of page sizes smaller than 512 words, we partition the address space into page sizes less than 512 words.

Our study reveals that for the LRU replacement algorithm, the influence of page size on the page fault frequency is highly dependent upon the size of the allocated memory space. If the program is forced to run in a relatively small memory (compared with the total number of referenced pages) the page fault frequency is smaller for small page sizes. If enough space is available so that only a few replacements are to be made during processing, this relationship is reversed and the page fault frequency is mainly determined by the number of references pages. But this number is clearly smaller for large page sizes (Figure 5a). This empirical observation has been recently verified theoretically by Fagin and Easton.[6] The space-time product for all page sizes depends critically on the size of the allocated space. The minimum space-time product for $R = 10{,}000$ tends to be smaller for large page sizes, and the minimum space-time product for small page sizes occurs at a smaller memory size than the minimum space-time product for the larger page sizes.

For the PFF replacement algorithm, larger page sizes yield lower page fault rates than smaller page sizes (Figure 5b). The space-time product for the larger page sizes is always lower than that of the smaller page sizes if the critical interpage fault-time T is relatively small. As T increases, the large page sizes do not necessarily yield the best performance (Figure 5c). In general the performance of the PFF replacement algorithms does not change as drastically with the page size as it does in the case of the LRU algorithm.

For the larger page sizes and $R = 10{,}000$, the performance of the PFF replacement algorithms (in terms of the space-time product) is better than the performance of the LRU algorithm.

As the speed ratio R decreases (which is the current trend), small page sizes gain more performance improvement than large page sizes. For example, if R is decreased by an order of magnitude, the performance of small page sizes is already much better than that of large ones. For a more detailed discussion of this topic, the interested reader should refer to the work of Chu and Opderbeck.[5]

An analytical model for the PFF algorithm. The performance of a replacement algorithm depends largely on the behavior of the running program, which for our purposes, will be described by its reference strings. These reference strings can be obtained in two ways. First, a program is interpretively executed and its reference string is recorded; simulation techniques are used usually to evaluate replacement algorithms as discussed in previous sections. Second, the reference string is generated by a model of program behavior. In this case, the reference string is only described in terms of its statistical properties. These properties are then used to evaluate the performance of replacement algorithms, thereby considerably decreasing the overall effort in terms of cost and time. For this purpose, we use the LRU stack model[7] for representing program behavior, and we use a semi-Markov model for modeling the PFF algorithm. The LRU stack model is based on the memory contention stack generated by the LRU algorithm. To represent a program by this model, we assign to each position of the stack a time-invariant fixed probability. At any time,

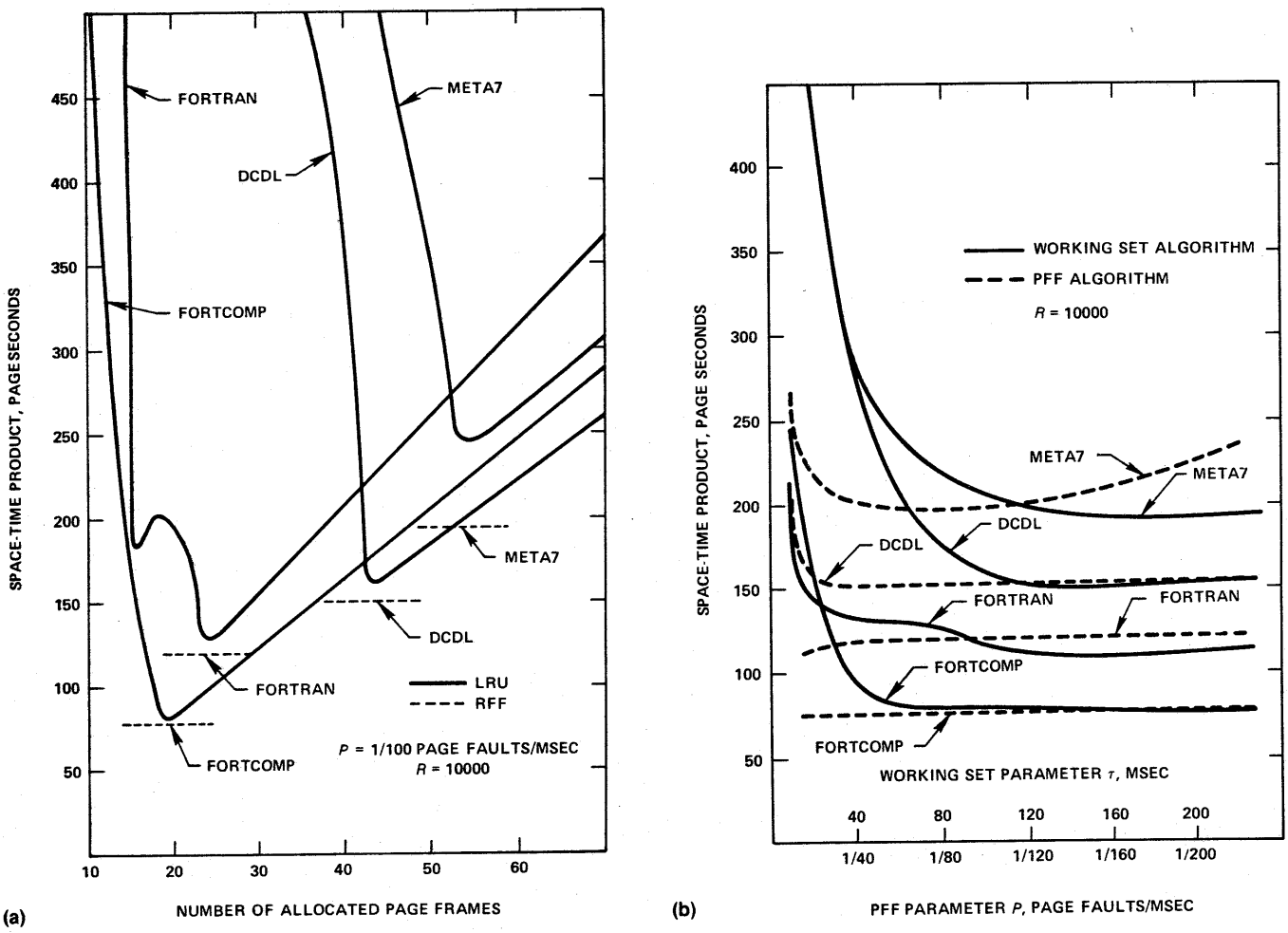


Figure 3. Performance comparisons between (a) the LRU and PFF algorithms and (b) the working set and PFF algorithms.

the stack position of the next reference is chosen with this probability. If stack position j is chosen, the page in that position is moved to the top of the stack. The pages at stack positions 1 through $j - 1$ are pushed down one position.. The consecutive stack positions are chosen independently of each other.

To represent the PFF replacement process, we let the number of allocated page frames represent a state in the page replacement process. Thus, if the page replacement process is in state j ($0 \leq j \leq M$), then j page frames are allocated, where M is the maximum number of pages that can be stored in the memory system. Since the number of allocated page frames (state) can only be changed at the time of a page fault, the page replacement process makes a transition whenever a page fault occurs. Since there is a finite but variable holding time (the number of pages referenced since the last page fault) associated with each state, we have to use a discrete time semi-Markov process to represent the PFF replacement process.

For the detailed development of the analytical model, the interested reader should refer to the work of Chu and Opderbeck.[8] Here, we shall provide the allocation state distributions (the number of pages allocated in the memory) of the Fortcomp and Meta7 programs for the PFF algorithm which were computed from the analytical model. We notice that the allocation state distribution is asymmetrical. Further, as the PFF parameter decreases, the mean of the allocation state increases, and its variance decreases as shown in Figures 6a and 6b.

Next we validate the analytical model by comparing the predicted page fault frequency and space-time product with that of measurement results. The measurement

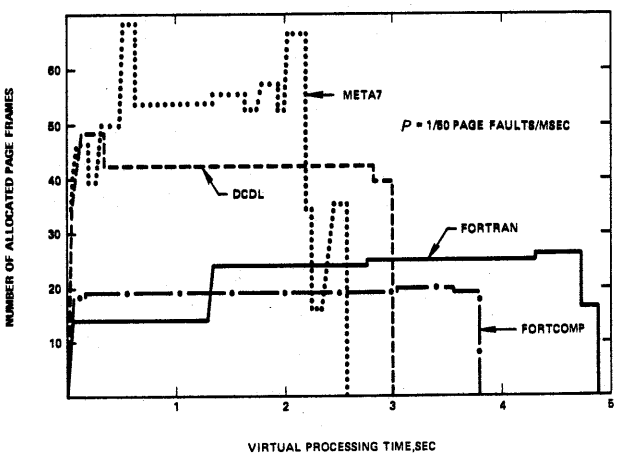
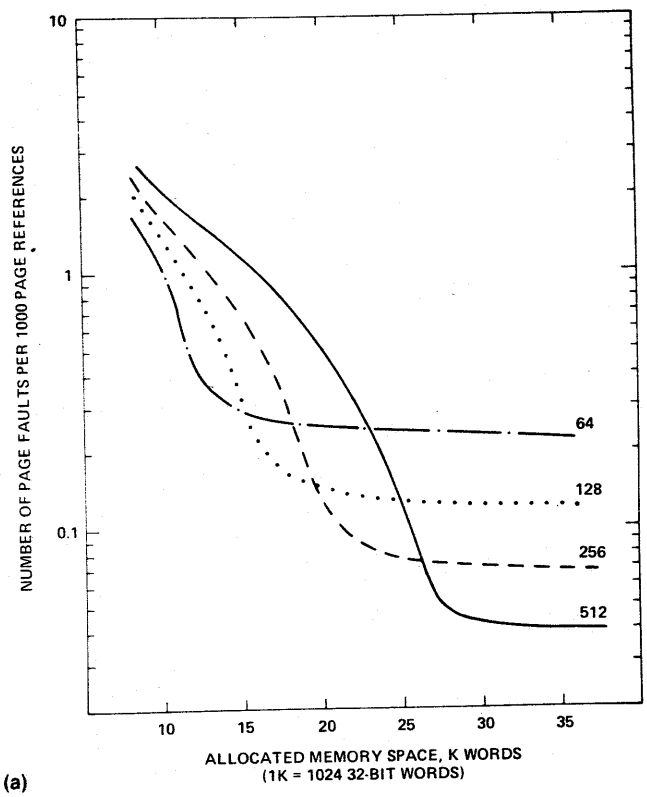
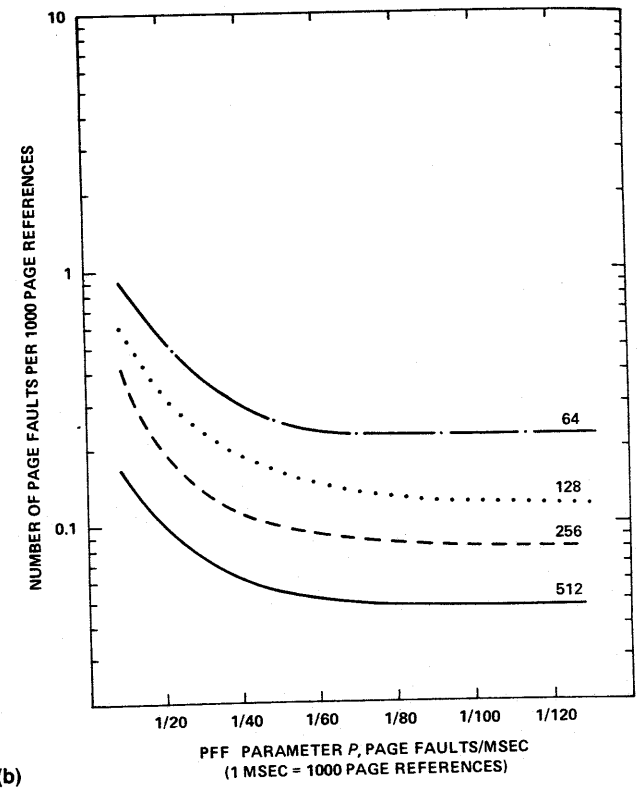


Figure 4. Dynamic changes in memory allocation of the PFF algorithm.





results were generated by using the page reference string as input to a program which simulated the PFF replacement algorithm. These results, as shown in Figures 7 and 8, reveal that the analytical model provides a good prediction of the performance for the PFF algorithm. We notice that the analytical model yields a better prediction for the Fortcomp program than for the Fortran program. This is mainly due to the more sudden (nonstationary) changes in the number of allocated pages for the Fortran program when compared with the Fortcomp program (see Figures 2a and 2b).

Application of the semi-Markov model for simulating the PFF algorithm. In a multiprogramming environment, we would like to study the interaction of various processes. Simulation based on every memory page reference becomes so costly as to be prohibitive. Since the PFF algorithm changes the allocation state only at the time of a page fault, we can estimate the interpage fault time by taking a random sample from the interpage fault time distribution. Simulating the replacement algorithm performance based on interpage fault time greatly reduces the simulation time. Further, if the sampled interpage fault time is less than the critical interpage fault time T, we increase the allocation state by one and no further computation is required. According to the PFF algorithm, all the pages that have not been referenced since the last page reference will be released. Therefore we must use the semi-Markov model to calculate the distribution of the number of distinct pages being referenced since the last page fault. For more detailed discussion on the model the reader should refer to the work of Chu and Opderbeck.[8]

Since the page fault rate of a system is around 10^{-4}, simulation of page fault events rather than simulation of each page reference represents a reduction of 10^4 events. Because we need to compute the number of distinct pages being referenced since the last page fault for an interpage fault time greater than T, the computation for the simu-

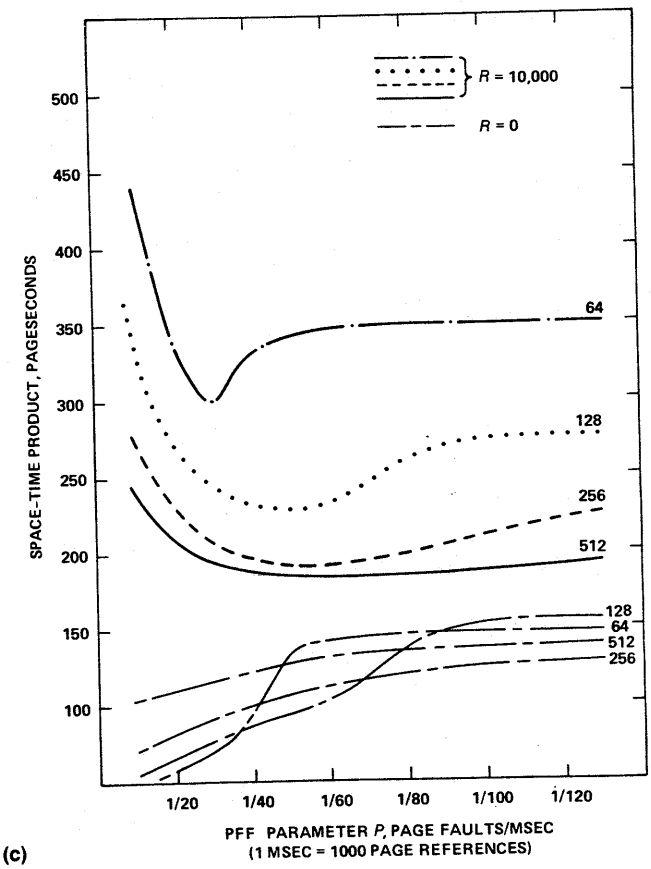


Figure 5. Performance of Meta7 program with different page sizes: (a) page fault frequency for the LRU replacement algorithm; (b) page fault frequency for the PFF replacement algorithm; and (c) space-time product for the PFF replacement algorithm.

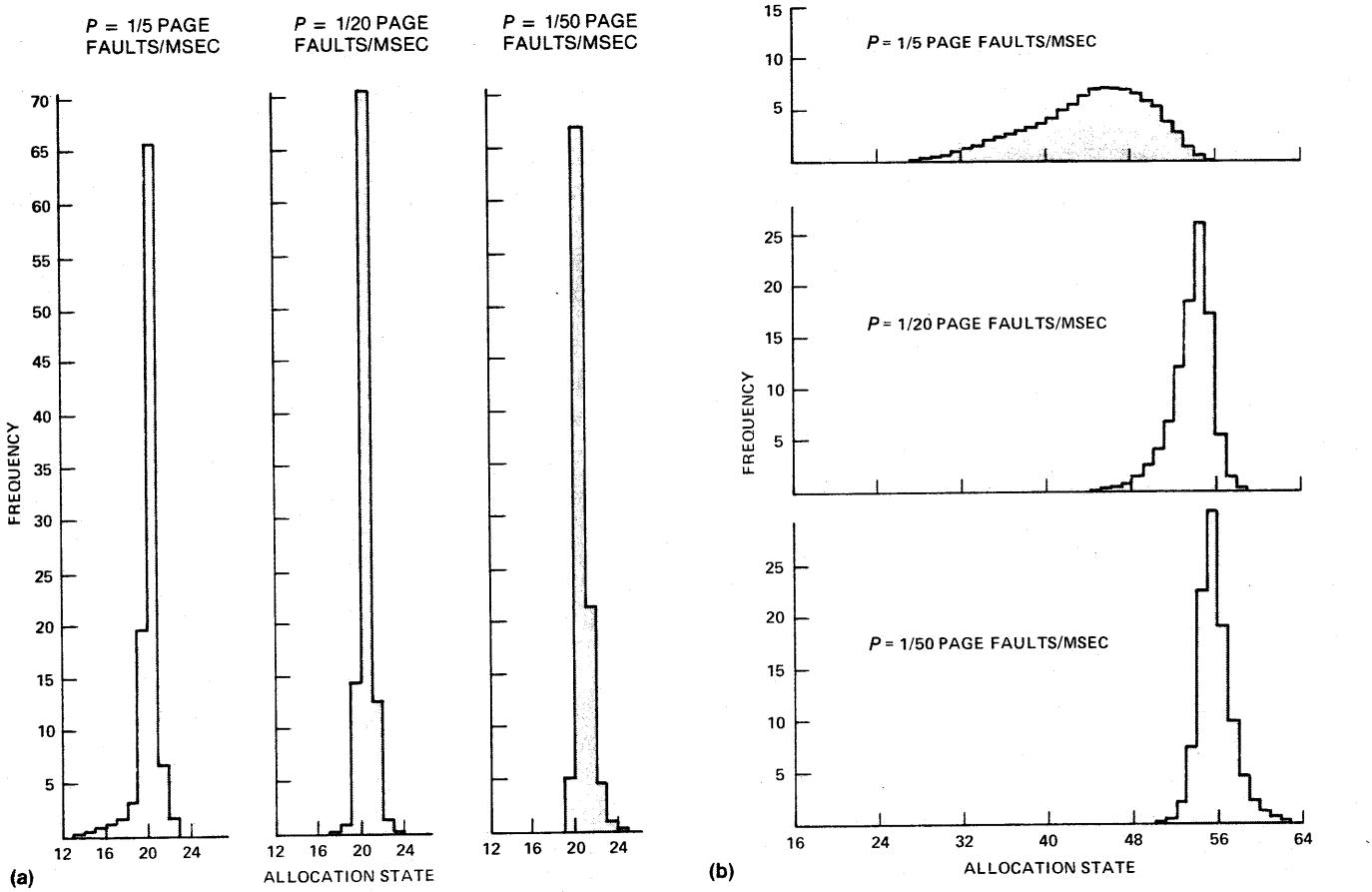


Figure 6. Allocation state distributions for (a) the Fortcomp program and (b) the second phase of the Meta7 program.

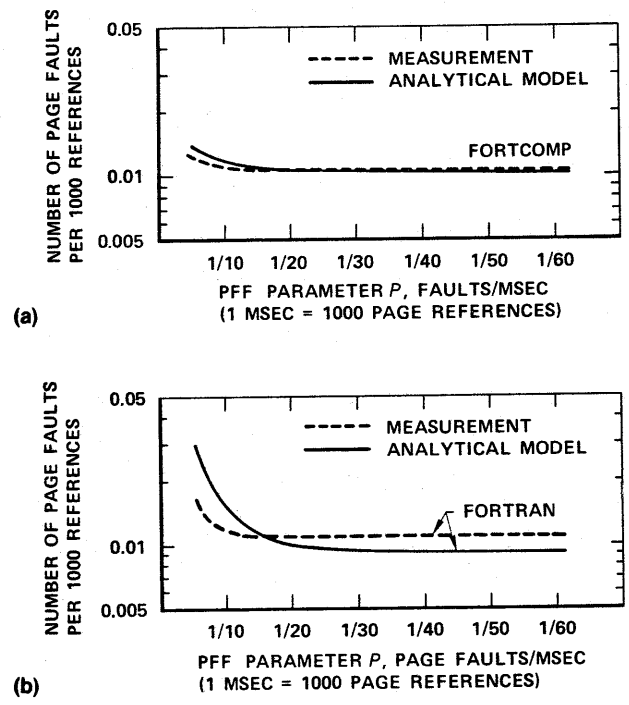


Figure 7. Comparison of the page fault frequency between analytical model prediction (solid line) and measurement (dashed line) for the PFF algorithm. (a) Fortcomp program; (b) Fortran program.

lation of one event is about ten times more costly than that of the simulation of a single page reference. Therefore, simulation based on page faults rather than every page reference represents a reduction of 10^3 in computation time.

We have used the semi-Markov model in studying the performance of the PFF algorithm in a multiprogramming environment (results will be presented in the next

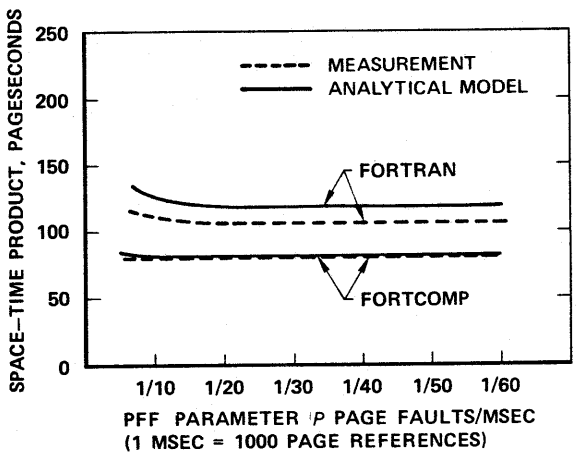


Figure 8. Comparison of the space-time product between analytical model prediction (solid line) and measurement (dashed line) for the PFF algorithm.

section). For simulating 1000 page faults for a given set of parameters, the required computation time is less than one minute on the IBM 360/91 system at a cost of about $20. Using the semi-Markov model to estimate analytically the number of distinct pages being referenced during the page fault interval permits us to evaluate the performance of the PFF algorithm based on the page fault intervals. As a result, we were able to complete our study with about two hours of computation time. Had we used every page reference in our simulation, it would have required about 2000 hours of computer time and the cost would have become prohibitive.

Performance in a multiprogramming environment. In a multiprogramming environment, the obvious advantage of the fixed storage partitioning (e.g., LRU) scheme is its easy implementation. The dynamic storage partitioning scheme (e.g., PFF), on the other hand, is more flexible. The number of allocated page frames may grow and shrink according to the dynamically changing memory requirements. Coffman and Ryan[9] used a mathematical model of locality to compare the two methods of storage partitioning. They report that the total memory size required for the dynamic partitioning scheme can be up to 30% less than that required by the fixed method for a given performance level if the variations in working set sizes are relatively large.

For the fixed storage partitioning scheme, new processes are usually started when a block of memory becomes free. For the dynamic storage partitioning scheme, on the other hand, a new process is not automatically started after the termination of a process. There is a separate activation decision which depends on the current state of the system. Further, a process may be deactivated when the pool of available page frames is empty and an additional page frame is demanded by some process.

Activation and deactivation policies. It is a well-known fact that when a process is first started, it rapidly demands page frames. At least some of these page frames should be available in the pool of available page frames. We therefore define a critical pool size N_a such that no new process is activated unless the current number of page frames in the pool of available page frames is greater than or equal to N_a. Once the new process is started, it demands page frames and thereby reduces the size of the pool. If the size of the pool drops below N_a, no new processes are started. We also define a critical activation time interval T_a such that a process is only activated if the current virtual processing time of the most recently activated process is at least T_a. In short, a new process is activated if and only if the size of the pool of available page frames is greater than or equal to N_a and the current virtual processing time of the most recently activated process is at least T_a. Neither of these activation decisions require prior knowledge about program behavior.

If the pool of available page frames is empty a process must be deactivated. We studied the following five types of deactivation:

Type 1: the process that demands the additional page frame (and thereby initiates the deactivation) is deactivated;
Type 2: the process which the currently smallest number of allocated page frames is deactivated;
Type 3: the process with the currently largest number of allocated page frames is deactivated;
Type 4: the process with the currently smallest virtual processing time is deactivated; and
Type 5: the process with the currently largest virtual processing time is deactivated.

We also studied the case where the least recently used page of the page faulting process is replaced. For notational consistency, we shall call this a Type 0 deactivation.

Note that none of these deactivation types requires prior knowledge about program behavior. They are therefore in agreement with our general philosophy: memory allocation and process scheduling should not depend on information which is provided from outside (e.g., the user); instead, the system should gather all the information needed for scheduling dynamically during the execution.

In a multiprogramming environment, the performance measure we use is throughput, which measures the number of processes completed per unit of time. Our studies reveal that the performance of the PFF algorithm is rather insensitive to the values of the parameters of the activation policies. Among the deactivation policies, we found the deactivation of the process with the currently smallest virtual processing time (Type 4) to be a good deactivation rule. If this deactivation type is chosen, then usually that process is deactivated which has been started most recently (and is still collecting the pages for its initial working set). This policy appears to be a good deactivation rule. Our observation is supported by the good results for deactivation of type 2, i.e., the deactivation of the process with the currently smallest number of allocated page frames, since the process with the smallest number of page frames is in many cases the one which has been started most recently. For further detailed information, the interested reader should refer to the work of Opderbeck and Chu.[10]

Throughput comparisons between the LRU and the PFF algorithm. In our study, we used a mixed system of programs, i.e., a system with a mixture of Meta7-like and Fortcomp-like programs, as our workload. We associate with each process which is about to be started a probability α that it is a Fortcomp-like program and a probability of $1 - \alpha$ that it is a Meta7-like program. For the LRU replacement algorithm, a process is started whenever one of the equal-sized memory blocks becomes free. For the PFF replacement algorithm, the starting of a new process is determined by the activation policies. Thus, neither algorithm assumes any knowledge about the type of the started process.

The most important overhead function of the PFF algorithm is the handling of page faults. Let γ be defined as

$$\gamma = \frac{\text{overhead per page fault (PFF algorithm)}}{\text{overhead per page fault (LRU algorithm)}}.$$

Since the overhead for page fault handling is directly proportional to the average number of page faults per process, we can determine the throughput for all values of γ from a single simulation experiment.

Figure 9 displays the throughput for such a mixed system with a mix ratio $\alpha = 0.5$. The results for the mixed system are similar to those for systems with Meta7-like or Fortcomp-like programs only. The PFF algorithm gives a much better throughput between 60 and 100 page frames than the LRU algorithm for any degree of multiprogramming. The degree of multiprogramming for the LRU algorithm must be chosen very carefully since too large a number of processes may drastically decrease the performance, whereas too small a number of processes results in inefficient use of system resources. For example, for 120 page frames the throughput for two processes is about 32% better than the throughput for one process. For 80 page frames, on the other hand, the throughput for one process is more than 32% better than the throughput for two processes. The optimal degree of multipro-

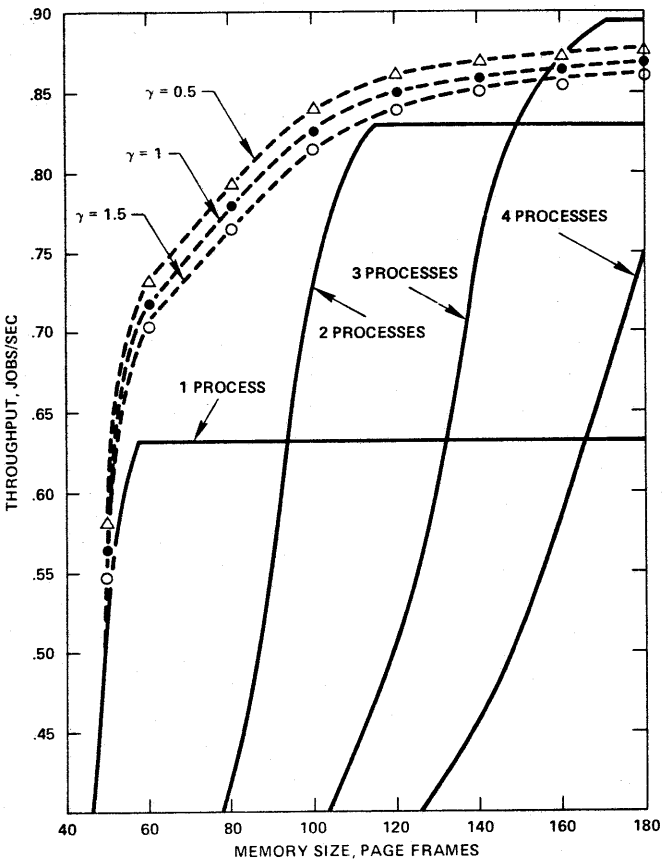


Figure 9. Throughput comparison between LRU (solid lines) and PFF algorithm (dashed lines).

gramming that yields maximum throughput is therefore an important design parameter for the LRU algorithm. The determination of this number is complicated by the fact that different programs usually have different memory requirements. Therefore this optimal number is usually a function of the current load of the system. For the PFF algorithm, because it is a dynamic memory management policy, the number of optimal degree of multiprogramming varies by its activation and deactivation policies and is adaptive to the current load of the system.

The results for the execution of single programs (Figures 3b and 5b) showed that the page fault frequency and the space-time product are rather insensitive to changes of the critical interpage fault interval T if T is sufficiently large. It was pointed out that this is an appealing feature of the PFF algorithm since it alleviates the problem of selecting an "optimal" interpage fault interval for implementation. For the same reason it is important to determine how the throughput of a multiprogramming system depends on the choice of the critical interpage fault interval. Therefore we repeated the previously described simulation experiment for a given memory size and varied the critical interpage fault interval from 10 to 100 msec. Figure 10 shows the throughput as a function of the critical interpage fault interval.

We observe that the throughput is very insensitive to changes of the critical interpage fault interval T for T greater than 30 msec. The throughput decreases for small values of T. This decrease corresponds to the increase in the average page fault frequency and the space-time product we observed for single programs when T is very small.

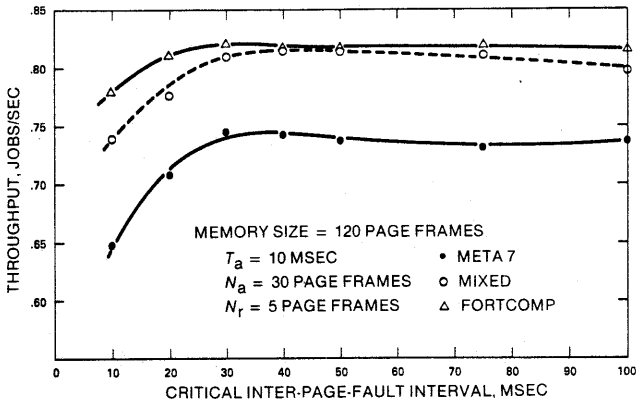


Figure 10. Throughput of the PFF algorithm as a function of the critical interpage-fault interval.

Implementation of the PFF replacement algorithm

In this section, we describe the implementation of the PFF algorithm and the handling of modified (dirty) pages and shared pages. Associated with each page, there is a page status bit which designates whether that page has been modified. Further, each activated process has its own USE-BIT table to record the used page of that process during an interpage fault time.

We need only a clock in the CPU to measure the process (or virtual) time between page faults of every process. The current process time of each process is recorded in the process' statusword. The page table entry can be used to determine which pages are residing and/or have been modified in the main memory. For those paging systems that have a USE-BIT feature, this feature can be used to determine those pages which have been referenced during the time interval since the last page fault occurred for a process. When the PFF algorithm operates in a multiprogramming environment, each process is assumed to have its own USE-BIT table. Whenever a page fault occurs in the ith process, the supervisor determines whether the process i is operating below the critical page-fault-frequency level P. For this purpose, the time of the last page fault has to be stored. If the last page fault occurred more than $T = 1/P$ msec ago, the supervisor examines the USE-BITS of all the activated processes as well as the page status bits to determine those pages of the ith process that have been neither shared nor modified since the last page fault. The supervisor then releases these pages from the main memory and resets their USE-BITS for the ith process. If the process is operating above the critical page-fault-frequency level P, no page frames are freed, and the reference page is added to the main memory. However, if there are no page frames available, then the supervisor according to the deactivation rule determines which process to deactivate. Again, by examining the USE-BITS of all the active processes as well as the page status list, the supervisor determines the set of pages of the deactivated process that can be released from the main memory and then resets the USE-BITS of that process. Whenever the pool of page frames reaches a level at which a new process can be activated, a new process is activated, its pages are bought into the main memory, and the USE-BITS of that process are reset.

Let us now consider the overhead of the above mentioned operations. We know that:

(1) The total overhead is proportional to the number of page faults. Since the PFF algorithm assures a low page

fault frequency, the overhead for handling page fault is simple, therefore, the overhead is low.

(2) Due to sudden changes of program localities, the virtual processing time between page faults is very short in many cases (see Figure 4). Whenever the time between page faults is less than $T = 1/P$, no page frames are freed and therefore there is no overhead involved in the "decrease decision" in these cases.

(3) The activation and deactivation rules do not require prior knowledge of program behavior and their overheads are very low. Comparing their implementation costs with the corresponding costs for the LRU and working set algorithm, we notice that the PFF algorithm is simple and is easier to implement, and requires less overhead to operate than the LRU and the working set algorithm.

In the PFF algorithm, increased or decreased page frames only occur when page fault occurs. Should a process operate in a loop many times, and all the referenced instructions and data be within the allocated pages, the interpage fault time could be exceedingly long. Therefore some of the nonreferenced pages of that process might not be released from the main memory. This problem can be dealt with in the following way: When the pool of available page frames becomes empty, the supervisor interrupts the active processes whose interpage fault interval exceeds the maximum interpage fault time, then releases these page frames that have not been referenced since the last page fault, and rests the USE-BITS as if a page fault had occurred.

Conclusion

The major purpose of this study was to gain insight into the behavior of programs in a paged memory system, to develop models of program behavior, and to use these models for an overall system evaluation. It was shown that the interaction between program behavior and memory management is very important for the efficient operation of a paged memory hierarchy. Therefore memory management policies which automatically adapt to program behavior should have great potential for the design of future computer systems. This is particularly true of the page-fault-frequency replacement algorithm, since it is relatively easy to implement.

We have shown that the PFF algorithm allows the number of allocated pages frames for a process to grow and shrink according to the dynamic changing memory requirements of the running program. The degree of multiprogramming varies by its activation and deactivation policies and adapts to the current system load. Therefore the PFF algorithm is less sensitive to changes of program behavior and in many cases gives a better performance (throughput) than the LRU replacement algorithm. Of course, all these comparisons were based on simulations and analytic studies with simulated system load. We still have to test the PFF algorithm in a more realistic environment. However, our results suggest that the PFF algorithm should be implemented.* An implementation would then clearly show what the benefits of such a dynamic memory policy are. ■

Acknowledgement

This research was supported by the U.S. Office of Naval Research, Contract No. N00014-75-C-0650.

*During the writing of this paper, it has been brought to the authors' attention that the Hewlett-Packard Company is currently implementing the PFF algorithm in one of their new computer products.

References

1. R. L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger, "Evaluation Techniques for Storage Hierarchies," *IBM Systems Journal*, Vol. 9 (2), 1970, pp. 78-117.
2. W. W. Chu and H. Opderbeck, "The Page Fault Frequency Replacement Algorithm," *AFIPS Conf. Proc., 1972 FJCC*, Vol. 41, pp. 597-609.
3. P. J. Denning, "The Working Set Model for Program Behavior," *CACM*, Vol. 11 (5), May 1968, pp. 323-333.
4. L. A. Belady and C. J. Kuehner, "Dynamic Space Sharing in Computer Systems," *CACM*, Vol. 12 (5), pp. 282-288.
5. W. W. Chu and H. Opderbeck, "Performance of Replacement Algorithms with Different Page Sizes," *Computer*, December 1974, pp. 14-21.
6. R. Fagin and M. C. Easton, "The Independence of Miss Ratio on Page Size," *JACM*, Vol. 23 (1), January 1976, pp. 128-146.
7. J. R. Spirn and P. J. Denning, "Experiments with Program Locality," *AFIPS Conf. Proc., 1972 FJCC*, Vol. 41 (1), pp. 611-621.
8. W. W. Chu and H. Opderbeck, "Analysis of the PFF Algorithm via a Semi-Markov Model," *CACM*, Vol. 19 (5), May 1976, pp. 298-304.
9. E. G. Coffman and T. A. Ryan, "A Study of Storage Partitioning Using a Mathematical Model of Locality," *CACM*, Vol. 15 (3), March 1972, pp. 185-190.
10. H. Opderbeck and W. W. Chu, "Performance of the Page Fault Frequency Replacement Algorithm in a Multiprogramming Environment," *IFIP Congress '74*, Stockholm, Sweden, August 1974, pp. 235-241.

Wesley W. Chu is a professor in the Computer Science Department at UCLA. He has worked on switching circuit design in GE's Computer Department, on the design of high speed computers at IBM, and on research in computer communication studies at Bell Laboratories. A consultant to several computer industries, he has authored and co-authored more than 40 articles on information processing systems and computer communications. Among his current research interests are computer communications, memory management, distributed data bases, and fault tolerant design.

Dr. Chu has been an ACM National Lecturer, an IEEE Computer Society Distinguished Visitor, chairman of the ACM's SIGCOMM, program chairman of the 4th Data Communication Symposium, Quebec, Canada, 1975, and is a senior member of the IEEE. He is a member of Tau Beta Pi, Eta Kappa Nu, Phi Tau Phi, and Sigma Xi, as well as editor of the book *Advances in Computer Communications*, Artech House, 1974 (1st ed.) and 1976 (2nd ed.). He received his BS and MS from the University of Michigan, and his PhD from Stanford.

Holger Opderbeck is director of network design with Telenet Communications Corp., Washington, D.C. From 1973 to 1975 he was head of the ARPA Network Measurement Center at UCLA. He had been at UCLA since late 1970 when he was awarded a Fulbright scholarship for the study of computer science. Before that he worked as a systems analyst at Siemens, AG, Munich, where he did research in the area of information system design.

Dr. Opderbeck received his BS and MS degrees in physics from the University of Munich in 1967 and 1969, and his MS and PhD degrees in Computer Science in 1971 and 1973, respectively. He is a member of the IEEE Computer Society and ACM.