# The page fault frequency replacement algorithm*

*by* WESLEY W. CHU and HOLGER OPDERBECK

*University of California*
Los Angeles, California

## INTRODUCTION

Dynamic memory management is an important advance in memory allocation especially in virtual memory and multiprogramming systems. In this paper we consider the case of paged memory systems: that is, the physical and logical address space of these systems is partitioned into equal size blocks of contiguous addresses. The paged memory system has been used by many computer systems. However, the basic memory management problem of deciding which pages should be kept in the main memory to allow efficient operation without wasting space is still not sufficiently understood and has been of considerable interest. Obviously, pages should only be removed from the main memory if there is a very low probability that they will be used in the near future. The difficulty lies in trying to determine which pages to remove, without incurring difficult implementation problems at the same time.

Many replacement algorithms have been proposed and studied in the past, such as Random, First-in First out, and Stack Replacement Algorithms[1] [for example, Least Recently Used (LRU)]. These replacement algorithms are usually operated with a fixed size memory allocation. For such a fixed size memory replacement algorithm we need to have prior knowledge about program behavior. For example, in the LRU case, we need to have an estimate for the number of page frames which have to be allocated for each individual program. Further, program behavior is usually data dependent and changes during execution. An efficient replacement algorithm should therefore automatically adapt to the dynamically changing memory requirements. A recent study by Coffman and Ryan[2] shows that such dynamic storage partitioning provides substantial increases in storage utilization over fixed partitioning. The working set model of program behavior[3] takes into account the varying memory requirements during execution. With respect to this model, we call the replacement algorithm, which keeps exactly those pages in main memory that have been accessed during the last $\tau$ references, the Working Set Replacement Algorithm. The performance of the Working Set Algorithm still depends on the choice of the working set parameter $\tau$ and program characteristics (e.g., locality).[4] Further, the Working Set Algorithm appears expensive to implement. Therefore we were motivated to develop an adaptive replacement algorithm which is largely independent of program behavior and input data and is simple to implement. We shall use the page fault frequency (the frequency of those instances at which an executing program requires a page that is not in main memory) as an adaptive parameter to control the decision process of the replacement algorithm. Since the page fault frequency reflects the actual memory requirements of a program at execution time, the Page Fault Frequency (PFF) Algorithm can be applied to arbitrary programs without prior knowledge about program behavior.

The performance of replacement algorithms is usually compared in terms of efficiency and space-time product. Because of the complex nature of program behavior, we used simulation techniques to measure the efficiency and the space-time product for various programs. From these simulations we were able to compare the performance of the LRU and Working Set Replacement Algorithms. Next we describe the PFF Algorithm and compare its performance with the LRU and Working Set Replacement Algorithms. Finally, we discuss the advantages of this new replacement algorithm when employed in a multiprogramming environment and the implementation of the PFF Replacement Algorithm.

## PERFORMANCE OF LRU AND WORKING SET REPLACEMENT ALGORITHMS

Because of the complex nature of program behavior, analytical estimation of such parameters as page fault

* This research was supported by the U.S. Office of Naval Research, Mathematical and Information Sciences Division, Contract Number N00014-69-A-0200-4027, NR 048-129.

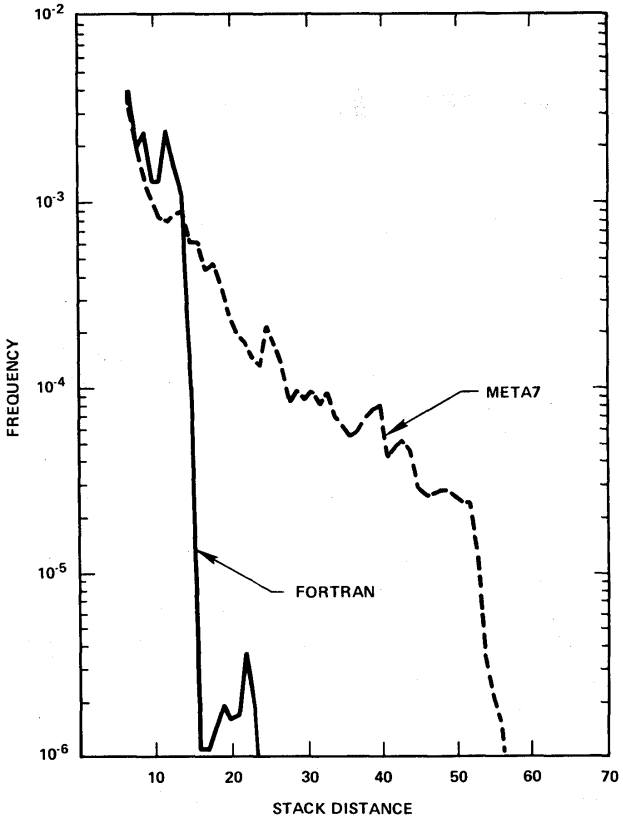
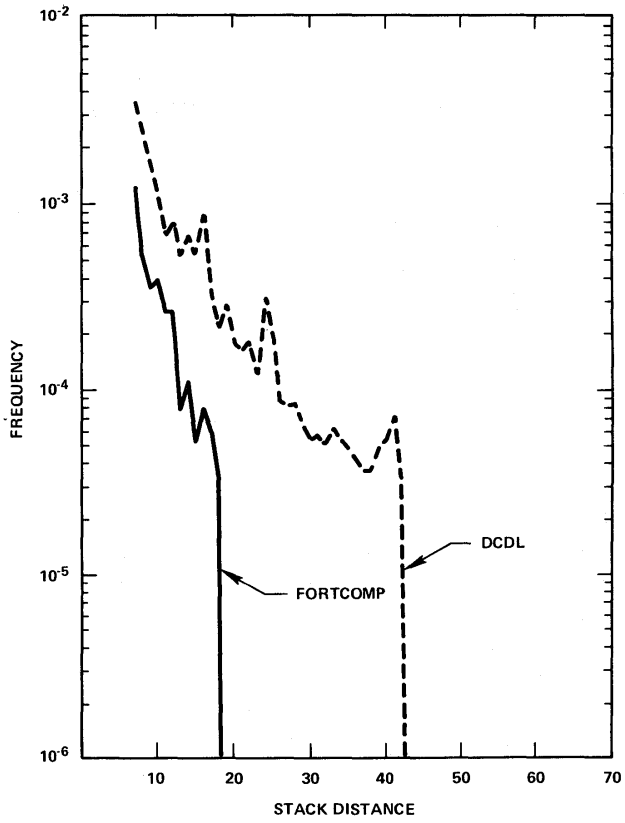



Figure 1—Stack distance frequency for the four measured programs
a) FORTRAN and META7




b) FORTCOMP and DCDL

frequency and the average inter-page-fault-time (average process running time between page faults) becomes very difficult. Yet this information is important in the planning of an efficient replacement algorithm that optimizes system performance. Therefore we employ measurement techniques for such estimations. This technique has been used previously to measure dynamic program behavior[5] and also to measure the performance of the Belady Optimal Replacement Algorithm,[6] the LRU Replacement Algorithm,[7,8] and the Working Set Replacement Algorithm.[4]

For this purpose an interpreter for the UCLA SIGMA-7 time-sharing system has been developed. This interpreter is capable of executing SIGMA-7 object programs by handling the latter as data and reproducing a program's reference string. This sequence, in turn, 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 a thousand page references correspond to one millisecond (msec).

Four different programs of various characteristics were interpretively executed. A FORTRAN Program (FORTRAN) 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

TABLE I—Characteristics of Measured Programs

| | SIZE | | NUMBER OF PAGE REFERENCES |
|---|---|---|---|
| | STATIC $s_o$ | DYNAMIC $r_o$ | |
| FORTRAN | 24 | 38 | 4,870,000 |
| FORTCOMP | 24 | 39 | 3,810,000 |
| DCDL | 44 | 71 | 3,010,000 |
| META7 | 84 | 165 | 2,590,000 |

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 in Reference 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 I 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 pages which make up the program may be referenced while processing a particular set of input data; second, a number of data pages is 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.

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

$$f(\omega) = \frac{r}{t}$$

where

$r$ = total number of page faults

$t$ = total number of page references

For a finite $t$, since $r \geq r_0 > 0$, $f(\omega)$ is always greater than 0. The average inter-page-fault-time is $t/r$ page references.

If no pages are replaced, the smallest page fault frequency is $f_0(\omega) = r_0/t$ which depends on $r_0$ and $t$. In general, $f_0(\omega)$ is different for different programs and could be different even for the same program when processed with different sets of input data. For this reason, it is awkward to use $f(\omega)$ as a measure to compare the performance of a replacement algorithm when applied to different reference strings. We therefore define a normalized page fault frequency, $f_n(\omega)$, as

$$f_n(\omega) = \frac{r - r_0}{t}.$$

The normalized page fault frequency considers only those page faults which are caused by references to pages which have been accessed before but which were replaced later. Clearly, if no pages are replaced, $f_n(\omega)$ is 0.

The efficiency $E$ for a program execution is defined as the ratio of total virtual processing time (processing time without page-wait times) to the total real processing time (total virtual processing time plus total page-wait times); that is,

$$E = \frac{\text{total virtual processing time}}{\text{total real processing time}}$$

$$= \frac{1}{1 + f(\omega) \cdot R}, \tag{1}$$

where

$R = T_s / T_m$

$T_m$ = access time of main memory

$T_s$ = access time of secondary memory

$R$ is called the speed ratio of a particular combination of secondary and main memory. We assume $T_m$ to be the time of one page reference ($10^{-3}$ msec). The maximum efficiency which can be achieved if no pages are replaced is

$$E_0 = \left(1 + \frac{r_0}{t} \cdot R\right)^{-1}$$

$E_0$ again depends on $r_0$ and $t$ and is therefore in general different for each reference string. For this reason, we define the normalized efficiency $E_n$ which corresponds to the normalized page fault frequency $f_n(\omega)$, as

$$E_n = [1 + f_n(\omega) \cdot R]^{-1}$$

and use this as a measure to compare the performance of various replacement algorithms. Note that $E_n$ always reaches its maximum of 100 percent if no page is replaced that will be referenced again, and that $E_n$ is independent of $r_0$.

Figures 2 and 3 display the normalized efficiency and the average inter-page-fault-time as a function of memory space allocation for the LRU Algorithm with $R = 10{,}000$.* For a given memory space we notice that

* These curves can be derived directly from the stack distance frequencies in Figure 1 (see Reference 1).

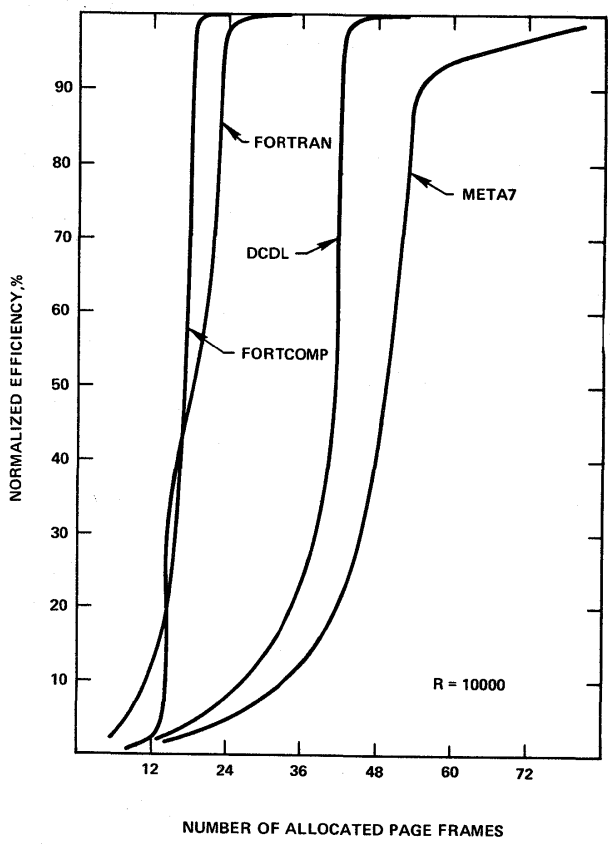



Figure 2—Normalized efficiency of the LRU algorithm

different programs have different normalized efficiencies and different average inter-page-fault-times. Further, programs with small localities tend to yield better performance than programs with large localities. The average inter-page-fault-time increases as the assigned memory space increases and reaches its maximum, $t/r_0$, as the memory space reaches a certain size. At this memory size the normalized efficiency reaches 100 percent. Further increase in memory space does not increase the average inter-page-fault-time and efficiency. The fact that all four curves in Figures 2 and 3 have their steepest slope occurring at different memory sizes reflects the different memory needs for each program. Thus, for a given process that uses the LRU replacement algorithm, one of the most difficult tasks is to determine the size of the memory which is to be allocated for each process. Assigning too large a number of page frames for a process results in inefficient utilization of memory space, while assigning too small a number of page frames yields too many page faults, resulting in inefficient operation. A procedure which gives the same amount of main memory to every process will almost surely result in either inefficiencies or waste of storage. In addition, the estimate of the memory needs of a process should be fairly accurate because only a few pages less than actually necessary means, in many cases, a large decrease in the average inter-page-fault-time. The determination of the amount of required memory is further complicated by the fact that it is usually data dependent and may vary during execution.

In contrast with the LRU Algorithm, the Working Set Replacement Algorithm requires a variable sized storage space for each process. This variable storage space provides the capability to adapt to dynamic changes in program behavior. The working set $W(t, \tau)$ at a given time $t$ is the set of distinct pages referenced in the process (or virtual) time interval $(t-\tau+1, t)$, that is, the set of pages accessed during the last $\tau$ references where $\tau$ is called the working set parameter. The working set size $w(t, \tau)$ is the number of pages in $W(t, \tau)$. The basic replacement policy is to keep in the main memory those pages which have been referenced

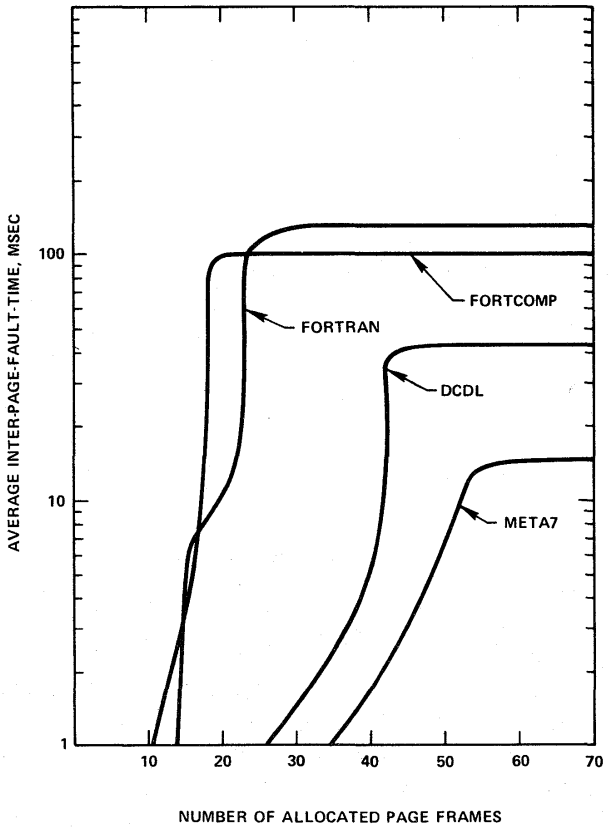



Figure 3—Average inter-page-fault-time of the LRU algorithm

during the last $\tau$ msec. $\tau$ is an important parameter which affects the performance of the Working Set Algorithm.

In general, the Working Set Algorithm can be considered as an LRU Algorithm with variable size memory allocation. There is, however, a crucial difference. Using an LRU Algorithm, pages are always replaced when a page fault occurs. This does not apply to the Working Set Algorithm. Here, page frames are freed whenever they have not been accessed during the last $\tau$ msec. A strict implementation would require the setting of a flag at this point; that is, an indication that the corresponding page frame can be used for a different page of any process. Another problem is to detect the exact time when a page has not been referenced during the last $\tau$ msec. Hence, it appears to be rather expensive to implement the Working Set Algorithm.

In the simulation of the Working Set Algorithm we assume that exactly the working set is kept in main memory. This means that a page fault is recorded whenever a page is accessed which is not included in the current working set. However, in systems using the working set strategy it may happen that a page leaves the working set and returns later without leaving main memory in the meantime. Thus, the actual page fault frequency during execution can be slightly lower than the page fault frequency we observed during simulation. This, of course, is true for every replacement algorithm which frees page frames without allocating them immediately to another page.

Figures 4 and 5 display the normalized efficiency and the average inter-page-fault-time as a function of $\tau$ for the four programs when the Working Set Replacement Algorithm is employed.* Both the average inter-

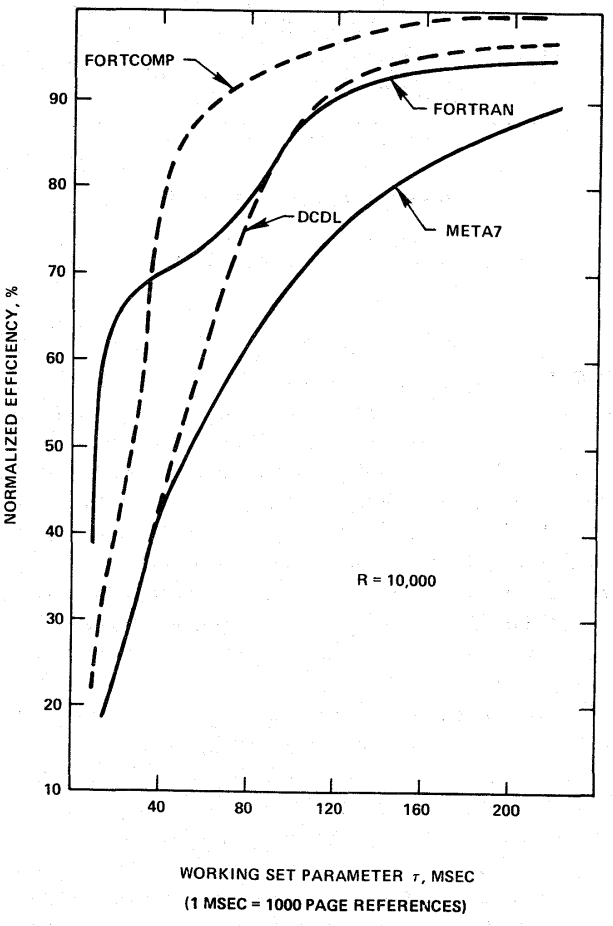



Figure 4—Normalized efficiency of the working set algorithm

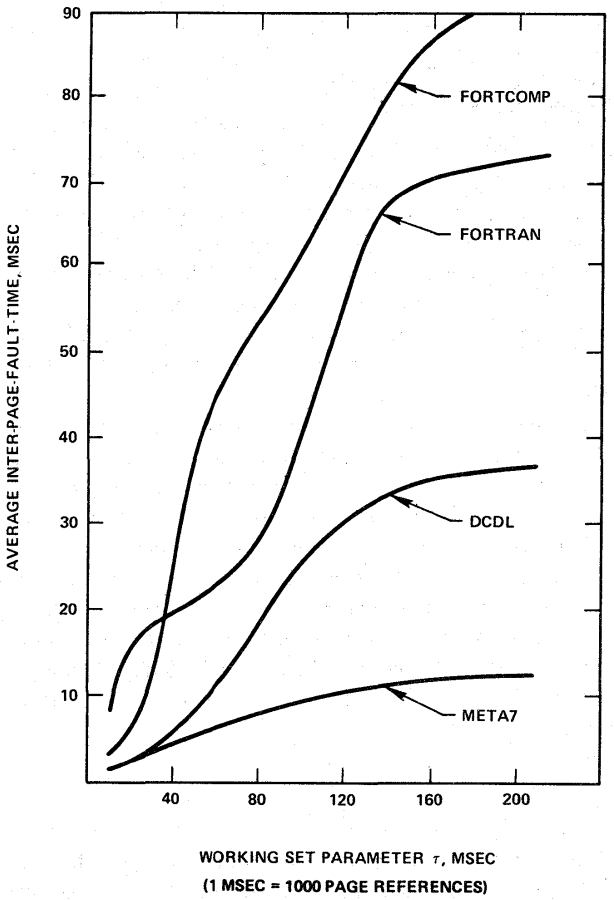



Figure 5—Average inter-page-fault-time of the working set algorithm

* For more measurement data on the Working Set Replacement Algorithm and their applications, interested readers should refer to Reference 4.

page-fault-time and the normalized efficiency increases rapidly as $\tau$ increases for the range of $\tau$ less than 80 msec. This suggests that in implementing a Working Set Replacement Algorithm for these four measured programs, we should choose a parameter $\tau$ of at least 80 msec. For $\tau = 200$ msec, the normalized efficiency for all four programs is greater than 85 percent. We also note that different programs achieve different efficiencies. For a given $\tau$, the programs with small localities usually give better performance than the programs with large localities. This occurs because $\tau$ may not be large enough to keep the favored pages in the main memory to assure a high efficiency and high average inter-page-fault-time.

As defined in (1), program efficiency is the ratio of total virtual processing time to the total real processing time and provides us with information on how fast the process will run. From an economic point of view, another parameter of interest is cost. Therefore, we shall also include cost of storage as a measure to compare various replacement algorithms. One of the criteria is the space-time product which can be considered as being proportional to the cost of storage. Belady and Kuehner[9] define the space-time product $C$ during the real time interval $(0, t)$ as

$$C = \int_0^t S(z)\, dz \qquad (2)$$

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 actual processing time. This occurs because of multiple processes running in parallel and because of page-wait times. Since our study is concerned with the application of replacement algorithms to individual programs, only page-wait times have been considered.

If we consider the execution of a program as a discrete process, the integral in (2) can be replaced by a sum which consists of two parts. The first part is the space-time product, due to the actual processing, while the second part is due to the total page-wait time. Thus, the space-time product $C$ can be re-written as

$$C = \sum_{i=1}^{t} S_i T_m + \sum_{i=1}^{r} S_{t_i+1} \cdot R \cdot T_m \qquad (3)$$

where $t$ is the total number of references; $r$ is the total number of page faults; $S_i$ is the number of allocated page frames prior to the $i$th reference ($i$ is called the number of the reference); $t_i$ is the number of the reference which causes the $i$th page fault (since we do not preload any pages, $t_1 = 1$); and $S_{t_i+1}$ is therefore the number of page frames which are allocated during the $i$th page-wait time.

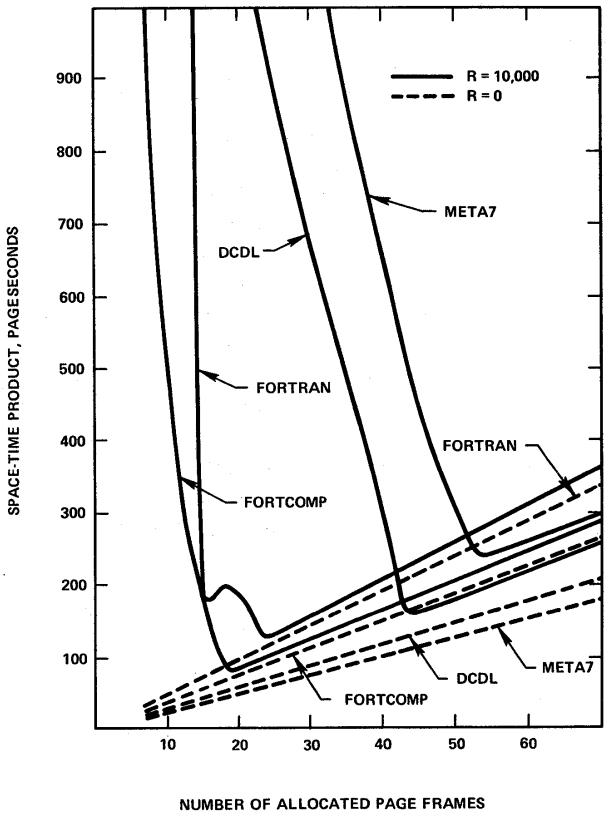



Figure 6—Space-time product of the LRU algorithm

Since $\sum_{i=1}^{r} S_{t_i+1} \cdot T_m$ and $\sum_{i=1}^{t} S_i T_m$ 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 $\sum_{i=1}^{r} S_{t_i+1} \cdot T_m$ from (3), and thus $C$ for any $R$ for $0 < R < \infty$. For example, the space-time products of $R = 0$ (the dashed lines) and $R = 10{,}000$ for META7 with the LRU Algorithm are shown in Figure 6. The space-time product for $R = 5{,}000$ can be obtained easily from a linear interpretation of the curve $R = 0$ and $R = 10{,}000$. Hence the space-time product presented in this paper can also be extrapolated to other values of $R (0 < R < \infty)$.

Figure 6 displays the space-time product as a function of the allocated memory space for the LRU Replacement Algorithm with $R = 10{,}000$. Figure 7 displays the space-time product as a function of $\tau$ for the Working Set Replacement Algorithm with $R = 10{,}000$.

Comparing Figure 6 with Figure 2, we observe two interesting properties. First, the minimum space-time product is below the number of page frames necessary to achieve maximal efficiency. This means that minimal space-time product and maximal efficiency can be

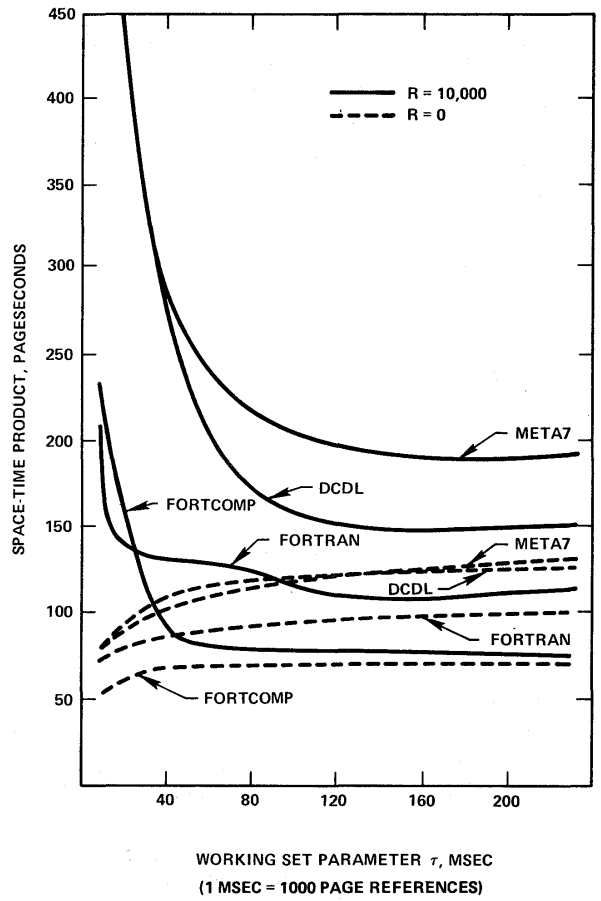



Figure 7—Space-time product of the working set algorithm

contradictory. This is because, when the number of allocated page frames reaches the point that yields the minimal space-time product, an additional page frame may decrease the total number of page faults (that is, increase the efficiency) and thereby possibly decrease the space-time product due to page-wait times [second term in (3)]. However, this decrease is not off-set by the increase in the space-time product due to actual processing [first term in (3)].

Second, in terms of the space-time product it is much more disastrous not to allocate enough pages to a process than to allocate too many. For instance, in the META7 case, if 70 pages of memory space are allocated instead of the optimal number of 55 pages, the space-time product increases from 247 to 306 pageseconds. But if only 40 pages are assigned to the same process, the space-time product increases to 700 pageseconds. Comparing Figure 7 with Figure 4, we observe similar characteristics except that the Working Set Replacement Algorithm has a much better and less sensitive (with respect to $\tau$) space-time product than does LRU.

From the above study we know that the major disadvantage of the LRU Replacement Algorithm is that it is not at all clear how many pages have to be allocated for different programs in order to assure efficient running without wasting space. In addition, this number is usually data dependent and may vary during execution. The Working Set Replacement Algorithm constitutes a possible solution to this problem. In this case, the amount of main memory which has to be allocated is automatically determined by the number of pages referenced during the last $\tau$ msec.

Since all of the measurements were done with the SIGMA-7 time-sharing system, the results presented here are somehow characterized by this very system. However, the conclusions are likely to be applicable to other paging systems. The reason for this is that program behavior is mainly determined by programming characteristics as for example, looping, modularity of code, and data layout. These characteristics are relatively independent of, for instance, word length or instruction set, and are common to all major computer systems. This assumption is supported by the fact that the results of the LRU Algorithm simulations for the SIGMA-7 are very similar to those for other computer systems (see Joseph's results for the ATLAS-computer,[8] and Coffman and Varian's results for the IBM 360/50[7]).

From the above measurement results, we know that a replacement algorithm that is to achieve a small space-time product, has to adapt itself to the memory requirements at execution time. Therefore, the number of page frames assigned to a process must not be a constant, but must vary according to program demand during execution. Lastly, the implementation of the replacement algorithm should be simple and of low cost. With these as our objectives, in the next section we introduce a new replacement algorithm that has these properties.

## THE PAGE FAULT FREQUENCY REPLACEMENT ALGORITHM

An "ideal" replacement algorithm should be independent of prior knowledge about program behavior; instead, all of the information needed to assure efficient memory allocation should be gathered during program execution. Conceptually, the Working Set Algorithm accomplishes this by keeping a table with an entry for each page which indicates when the corresponding page has been referenced last. This information enables the memory scheduler to decide when a page frame can be freed.

The Page Fault Frequency Algorithm uses the measured page fault frequency as the basic parameter for the memory allocation decision process. We assume

that 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., 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 usually reduces the page fault frequency. On the other hand, once the page fault frequency falls below $P$, page frames may be freed. The same operation will be repeated whenever the page fault frequency rises above $P$ again. We shall designate $P$, measured in number of page faults per msec (1 msec = 1,000 page references) as the PFF-parameter. $P$ may be expressed as $P = 1/T$, where $T$ is the critical inter-page-fault-time. We shall use the inversion of the inter-page-fault-time as a running estimate of the page fault frequency. That is, at the time of a page fault, if the inter-page-fault-time is less than or equal to $T$, then we increase the number of page frames by one and no pages in the main memory are replaced. For example, $P = 1/50$ means that if one or more page faults (excluding the current one) occurred during the last 50 msec, the memory scheduler will add one page frame to the allocated memory at the time of a page fault. As an initial condition it is assumed that a page fault occurred at time 0. This allows a process to start collecting its pages at the beginning of processing. Note that this "increase decision" is based only on the number of page faults which occurred during the last $T$ msec (virtual time) and is not based on the total number of page faults which occurred since the processing began. Thus the PFF Replacement Algorithm provides very fast response to a sudden increase in memory requirements.

Let us now describe the "decrease decision" rules of the PFF Algorithm: (1) page frames in the main memory are only freed at the time of a page fault, (2) only those page frames are freed which have not been referenced since the last page fault occurred, and (3) page frames are freed only if the page fault frequency lies below the critical level $P$. More precisely, let $t_k$ denote the time at which the $k$th page fault occurred. The page fault frequency lies below the critical level $P$ at time $t_{k+1}$ if and only if $t_k < t_{k+1} - T$, where $T = 1/P$. In this case, all page frames which have not been accessed in the time interval $(t_k, t_{k+1})$ are

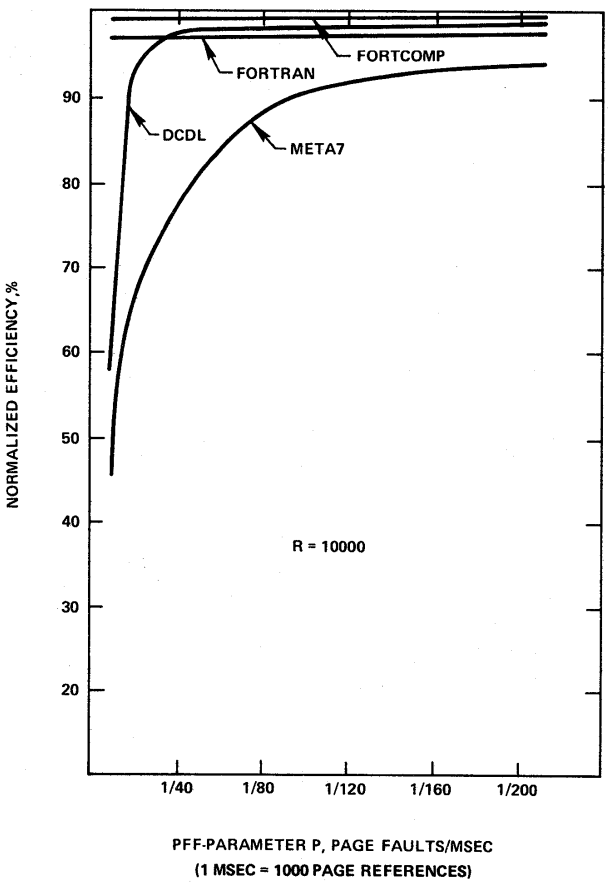



Figure 8—Normalized efficiency of the PFF algorithm

freed. If $t_k \geq t_{k+1} - T$, then no page frame is freed since the page fault frequency lies above $P$. We note that $T$ is somewhat similar to the working set parameter $\tau$. However there is an important difference. While $\tau$ indicates when a page should be freed, $T$ represents only a lower limit. Furthermore, in contrast to the Working Set Algorithm,* 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. This policy provides the PFF Replacement Algorithm with an extra adaptive capability which makes its performance less dependent on program characteristics than is the case with the Working Set Algorithm. The PFF Replacement Algorithm may therefore be considered as a Working Set Algorithm with variable $\tau$, where the value of $\tau$ is determined by

* This is, of course, only true for the strict implementation of the Working Set Algorithm which has been simulated.

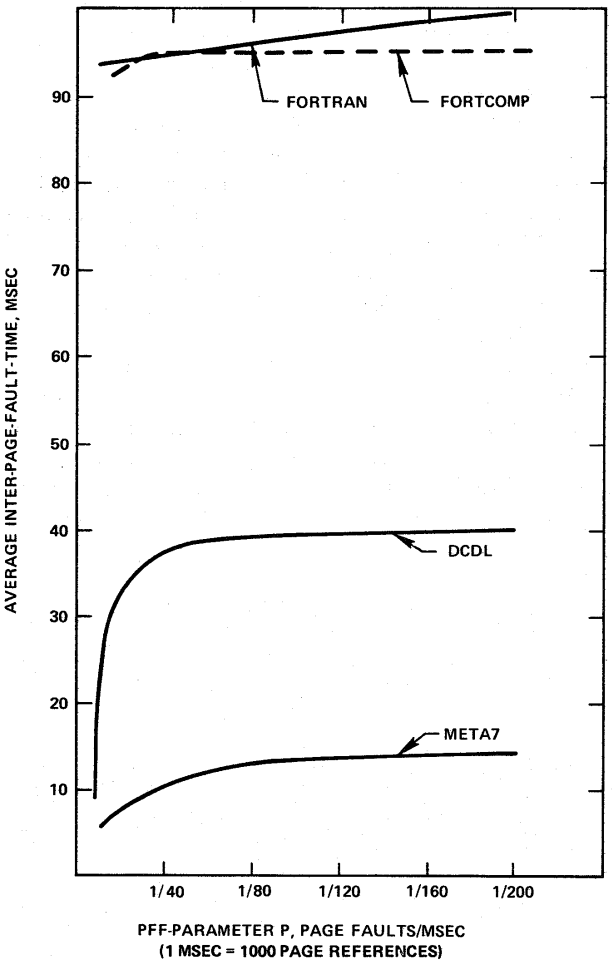



Figure 9—Average inter-page-fault-time of the PFF algorithm

the page fault frequency and the lower limit of $\tau$ 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 inter-page-fault-times.

Figures 8 and 9 show the normalized efficiency and the average inter-page-fault-time for the PFF Replacement Algorithm for which $P$ ranges from 1/10 to 1/200 page faults/msec. Figure 8 reveals two interesting properties of the PFF Algorithm: (1) For $P<1/100$, the normalized efficiency is larger than 90 percent for the four measured programs. This implies that high efficiency can be achieved by the PFF Replacement Algorithm by using the same page fault frequency parameter for all four programs regardless of their size and characteristics. (2) for FORTCOMP and FORTRAN the normalized efficiency is virtually independent of $P$. The same is true for DCDL if $P$ is less than 1/40 and for META7 if $P$ is less than 1/120 page faults/msec.

Figure 10 displays the space-time product for the PFF Replacement Algorithm. Again we can observe that the performance of the PFF Algorithm is almost independent of the choice of $P$ for $P<1/50$. For the four measured programs, the space-time products are almost constant over a wide range of values of $P$. The performance of the PFF Replacement Algorithm is therefore relatively insensitive to $P$. This is a very appealing feature of the PFF Algorithm since it alleviates the task of selecting a parameter $P$ for implementation.

In Figure 11, the number of allocated page frames is displayed as a function of virtual processing time. As can be seen, the memory requirements 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

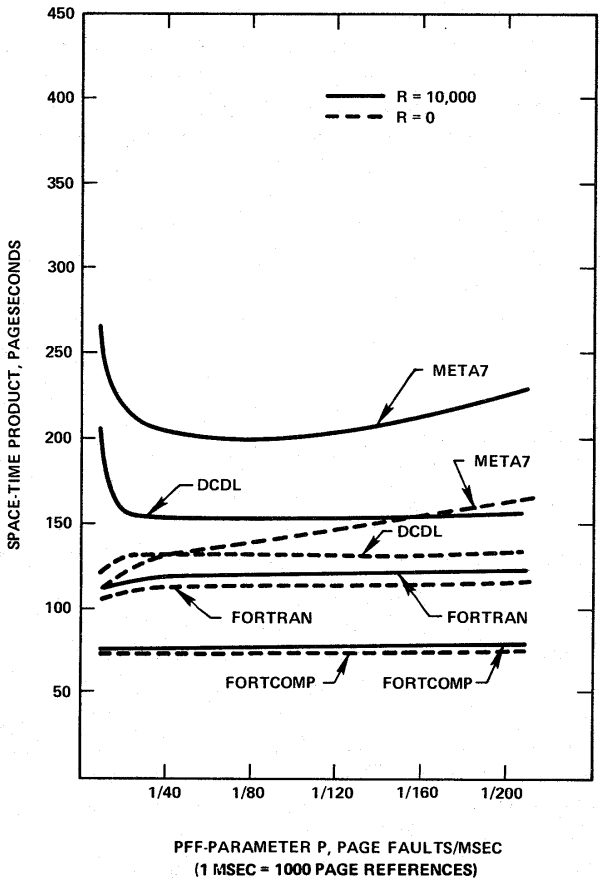



Figure 10—Space-time product of the PFF algorithm

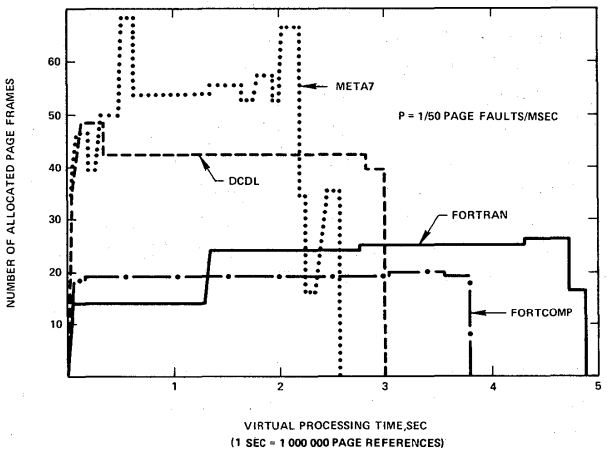



Figure 11—Dynamic changes in memory allocation of the PFF algorithm

below the four curves corresponds to the space-time product due to actual processing time. For simplicity in representation, only major changes in the number of allocated page frames are indicated in Figure 11. Nevertheless, the figure shows clearly that the majority of page faults which resulted from changes in program locality occurred during relatively short time intervals.

## COMPARISON WITH THE LRU AND WORKING SET REPLACEMENT ALGORITHMS

In order to compare different replacement algorithms, it is not enough to compare their measured efficiency, since every replacement algorithm will yield a high efficiency if the number of replacements is very small. This can be achieved by providing a large amount of memory. A better criterion for evaluating the performance of a replacement algorithm is the amount of main memory required to achieve a high efficiency level. For this reason the space-time product is a more valid measure for comparison.

Let us use the performance of a specific PFF Algorithm with $P=1/100$ page faults/msec to compare it with the LRU Replacement Algorithm for various numbers of allocated page frames. Figure 12 shows that the space-time products of the PFF Algorithm for all four programs are lower than the minimum space-time products achievable by the LRU Replacement Algorithm. This implies that the performance of the PFF Algorithm is better than that of the best LRU Algorithm.

The above results reveal the inefficiency involved in the fixed-size memory allocation. In such systems good performance can only be achieved if we allocate a number of page frames which is close to the optimal number that minimizes the space-time product. But even if we knew this optimal number, the performance of the PFF Algorithm would still be superior. This is especially true in those cases where the memory requirements vary drastically during execution (e.g., META7). If the memory requirements are relatively constant (e.g., FORTCOMP), then the performance of the LRU Algorithm is similar to that of the PFF Algorithm, provided that the optimal number of page frames is known a priori.

Figure 13 shows a comparison of the performance of the Working Set and PFF Algorithms. The space-time product is plotted for the working-set parameter $\tau$ and PFF-parameter $P$. For large values of $\tau$, the space-time product of the Working Set Algorithm is usually lower than the space-time product of the PFF Algorithm for

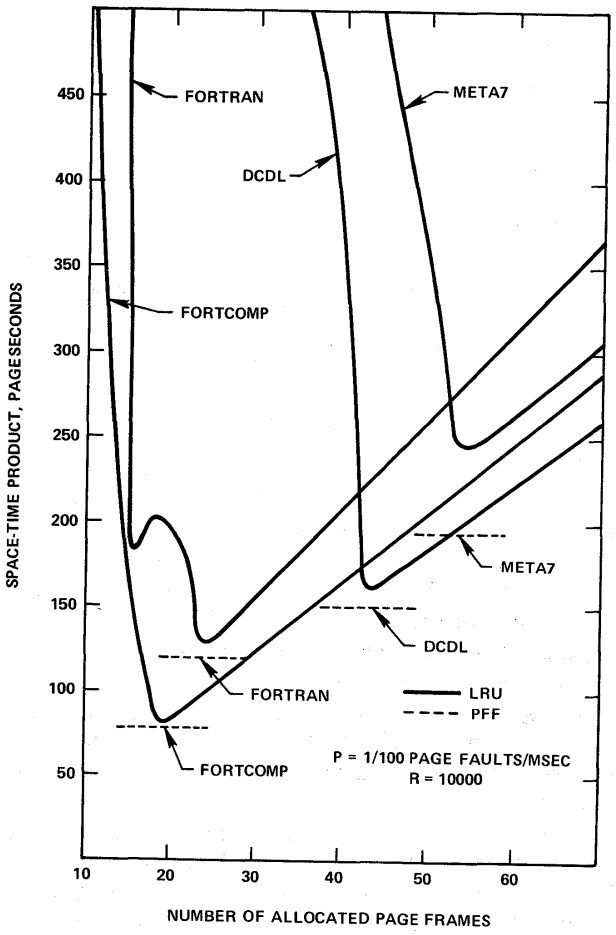



Figure 12—Performance comparison between the LRU and PFF algorithms

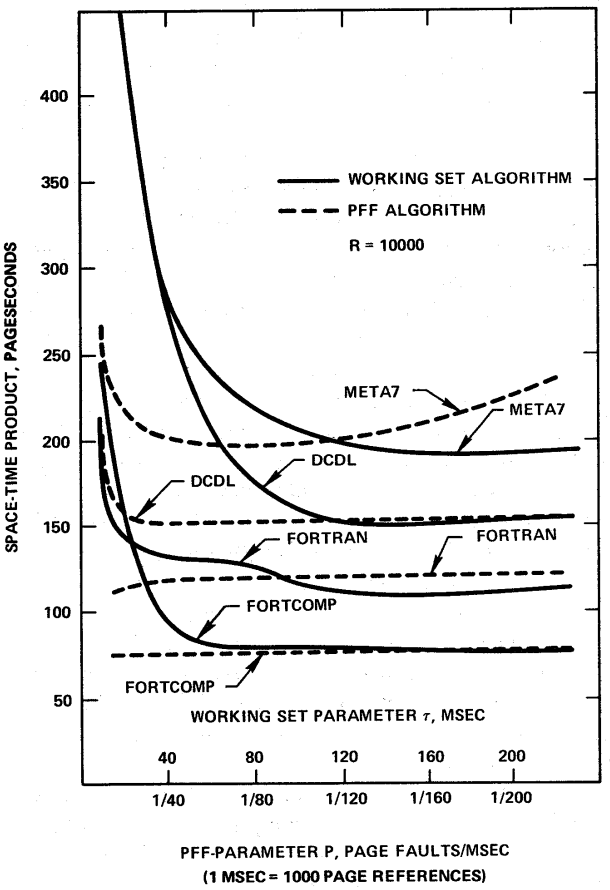



Figure 13—Performance comparison between the working set and PFF algorithms

corresponding values of $P$. Within the range of $P$ and $\tau$ of interest, the space-time product of the PFF Algorithm is less sensitive to $P$ than the space-time product of the Working Set Algorithm is to $\tau$. A similar statement can be made with respect to the normalized efficiency and the average inter-page-fault-time (see Figures 4, 8 and 5, 9). Since the space-time product of the PFF Algorithm is relatively insensitive to the PFF-parameter $P$, we do not have to know an "optimal PFF-parameter" to come close to optimal performance. Further, the minimum space-time product of the Working Set Algorithm is comparable to that of the PFF Algorithm. The normalized efficiency and the average inter-page-fault-time of the PFF Algorithm are greater than the normalized efficiency and average inter-page-fault-time of the Working Set Algorithm for all corresponding values of $P$ and $\tau$. This shows that the performance of the PFF Algorithm is comparable to the performance of the Working Set Algorithm.

## APPLICATION OF THE PFF ALGORITHM IN A MULTIPROGRAMMING ENVIRONMENT

Variable-sized memory allocation algorithms such as the Working Set Algorithm and the PFF Algorithm are useful in a multiprogramming environment where the main memory is shared by several programs. In this case the total amount of main memory not occupied by the resident supervisor can be considered as a pool of available page frames. These page frames are allocated to processes and returned to the pool according to the dynamically changing memory requirements of each individual process.

In the previous sections, we have used simulation techniques to study the performance of different replacement algorithms. It has always been assumed that there is enough main memory available so that a process can extend its memory space whenever needed. Any implementation of these variable-sized replacement algorithms, of course, must consider the possibility that the pool of available page frames is empty. The probability of this event is determined by the degree of multiprogramming and the type and size of the programs currently in the main memory. However, these variables can, to a large extent, be controlled by the process scheduling mechanism. Therefore, memory management is closely related to process scheduling in multiprogramming and time-sharing systems.

In order to reduce CPU idle time due to excessive page swapping, each process must be provided with enough main memory to keep the page fault frequency low. In a multiprogramming environment, it is crucial that this is accomplished without waste of memory space. The PFF Replacement Algorithm provides the supervisor with a means of achieving this effect which assures the same high efficiency level for programs of completely different types and sizes. In addition, the decrease policy of the PFF Algorithm continually tries to free those page frames which are no longer used by the process which enable processes to be run efficiently without wasting memory space.

The PFF Algorithm can also be very helpful for process scheduling since it gives the supervisor information about the required number of page frames for each process during execution. Once a process is removed from the main memory this information can be used to schedule this process for the next time quantum. In general, a process will be put on the processor queue only if there are enough available page frames in the pool. The information about the memory space of each process can also be used to decide which process has to be removed from the main memory if the page frame pool becomes empty.

There are many ways for the supervisor to make use

of the information about program behavior provided by the PFF Algorithm. Further investigations might yield other interesting applications.

## IMPLEMENTATION OF THE PFF REPLACEMENT ALGORITHM

The implementation of the PFF Algorithm is very simple. We need only a clock in the CPU to measure the time between page faults of every process. This clock measures the process (or virtual) time of each process. The current process time is recorded in the process' stateword. The page table entry can be used to determine which pages are residing 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. Whenever a page fault occurs, the USE-BITs are reset and the supervisor determines whether the process 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 USE-BITs are used to determine which pages have to be removed from the main memory.

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

1. The overhead is proportional to the number of page faults. Since the PFF Algorithm assures a low page fault frequency, the overhead is very low.
2. Due to sudden changes of program localities the virtual processing time between page faults is very short in many cases (see Figure 11). 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.

From the above implementation discussion we know that the PFF Algorithm is much easier to implement, and requires less overhead to operate than both the LRU and Working Set Algorithms.

## SUMMARY

A new type of replacement algorithm based on page fault frequency (PFF) is developed in this paper. This PFF Replacement Algorithm allocates memory according to the dynamically changing memory requirements of each process. It does not require prior knowledge of program behavior and can be applied to programs of different types and sizes. The PFF Algorithm uses the measured page fault frequency as the basic parameter for the memory allocation decision process. A high page fault frequency is considered to be an indication that a process needs more memory space to run efficiently. Thus whenever a page fault occurs the amount of allocated memory is increased if the page fault frequency lies above a given critical level $P$. $P$ is called the PFF-parameter. The number of allocated page frames may be decreased if the page of allocated page frames may be decreased if the page fault frequency falls below this level $P$. In this case only those page frames are freed which have not been accessed between successive page faults. The PFF Replacement Algorithm adapts to dynamic changes in program behavior during execution. As a result, this algorithm is largely independent of individual program behavior and input data.

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 (for which the optimal memory space is known a priori), and is comparable to the Working Set Replacement Algorithm. Further, the performance is relatively insensitive to changes in the PFF-parameter $P$.

The implementation of the PFF Replacement Algorithm is simple and less complicated than that of LRU and far less complicated than that of the Working Set Replacement Algorithm. It does not require any additional hardware. Using the PFF Algorithm in a multiprogramming environment, the supervisor has control over the efficiency and memory requirements of all processes. Based on this information, the supervisor can perform efficient process scheduling and memory allocation. From this study, we conclude that the PFF Replacement Algorithm should have high potential for use in future virtual memory and multiprogramming systems.

## REFERENCES

1 R L MATTSON J GECSEI D R SLUTZ L TRAIGER
*Evaluation techniques for storage hierarchies*
IBM Systems Journal 9 2 pp 78-117 1970

2 E G COFFMAN T A RYAN
*A study of storage partitioning using a mathematical model of locality*
Communications of the ACM 15 3 pp 185-190 March 1972

3 P J DENNING
*The working-set model for program behavior*
Communications of the ACM 11 5 pp 323-333 May 1968

4 W W CHU N OLIVER H OPDERBECK
*Measurement data on the working set replacement algorithm*

*and their applications*
Proceedings of the MRI International Symposium XXII Polytechnic Institute of Brooklyn April 1972

5 G H FINE C W JACKSON P V McISAAC
*Dynamic program behavior under paging*
Proceedings of the 21st National Conference of the ACM pp 223-228 1966

6 L A BELADY
*A study of replacement algorithms for virtual storage computers*
IBM Systems Journal 5 2 pp 78-101 1966

7 E G COFFMAN L C VARIAN
*Further experimental data on the behavior of programs in a paging environment*
Communications of the ACM 11 7 pp 471-474 July 1968

8 M JOSEPH
*An analysis of paging and program behavior*
Computer Journal 13 1 pp 48-54 February 1970

9 L A BELADY C J KUEHNER
*Dynamic space sharing in computer systems*
Communications of the ACM 12 5 pp 282-288 May 1969