1981

Improving Virtual Memory Performance by Off-Line Page Clustering

Jehan-Francois Paris

Report Number:
81-399

http://docs.lib.purdue.edu/cstech/325

This document has been made available through Purdue e-Pubs, a service of the Purdue University Libraries. Please contact epubs@purdue.edu for additional information.
ABSTRACT

A new approach to the improvement of paging systems performance is presented. The method is especially suited to those systems which have a relatively small page size. It consists of defining for each program clusters of pages that will always be fetched into memory and returned to the secondary store as a single entity. The algorithm building these clusters takes into account the memory policy under which programs are to run and operates upon data extracted from a trace of the program being reorganized and attempts to minimize its space-time product.

We prove that our algorithm simultaneously minimizes linear combinations of upper and lower bounds for page fault frequency and mean memory occupancy of all programs to be run under a working set policy, provided that the paging behavior of the program can be described by a stochastic model having a steady-state solution. These claims are confirmed by empirical evidence obtained from trace-driven simulations, which show that the method can substantially improve the performance of some programs running under a working set policy.

Index Terms: Virtual memory, paging, prefetching, program behavior, program restructuring, strategy-oriented restructuring, space-time product.
1. Introduction

One of the most difficult decisions facing the designer of paged virtual memory systems is the choice of the proper page size [14] [12] [11] [15]. On one hand, small page sizes are known to reduce internal fragmentation and to minimize the memory occupancy of programs. Larger page sizes, on the other hand, allow more information to be brought into memory in a single I/O operation. This makes larger page sizes especially attractive for systems using a disk unit as secondary store since these devices have a rather slow positional addressing mechanism and a high transfer rate. The effectiveness of a large page size depends however on the likelihood that the additional information brought into memory will be referenced in the near future. This depends on the referencing behavior of each individual program and, more specifically, on their degree of spatial locality. Since this spatial locality does not normally extend beyond the limits of each individual procedure, there is no point of increasing the page sizes much above this limit. (From this viewpoint, segmentation, where the number and the sizes of segments directly reflect the organization of programs, is clearly superior to paging.)

One could thus safely say that there is no such thing as an ideal page size.
A tempting solution is then to settle for a relatively small page size but to allow more than one page to be brought into memory at fault time. This is known as prefetching.

2. Prefetching and its Limitations

Prefetching attempts to reduce the frequency of page faults by fetching pages into memory before they are referenced and cause a fault. The most general prefetch policies would be able to anticipate future page faults and decide when it would be time to bring a given page into memory. Such policies would require collecting a lot of information and would be difficult to implement. A more feasible approach would then be to prefetch pages only at fault times. This technique—known as demand prefetching—has been studied by several authors [14] [1] [18] [22] [20] [21] [16] and has been successfully implemented on some systems, among which VAX VMS [4].

The main interest of demand prefetching vs. increasing the page size is to allow more control on the amount of information transferred into memory at each page fault. To keep comparable transfer costs, one must however ensure that all pages fetched together can be brought into memory in a single I/O operation; this will be the case if these pages are stored sequentially on the secondary store.

Depending on the way the prefetched pages are selected, one can distinguish two broad classes of prefetching policies. "Naive" policies only attempt to prefetch pages that have contiguous virtual addresses. These policies only require that pages with contiguous addresses should be stored sequentially on
the secondary store. They are relatively easy to implement and have been found to perform very successfully for hierarchical data bases, which are known to exhibit a very strong sequential behavior [20]. When applied to programs, these policies tend however to be not much more effective than simply increasing the page size [14] [21].

More sophisticated prefetching policies allow to fetch pages that do not have consecutive virtual addresses [1]. Such schemes obviously require much more information gathering on program behavior and referencing patterns. Since each cluster of pages fetched together should ideally reside in contiguous locations on the secondary store, these schemes could also require an extensive reorganization of the program lay-out in the secondary store. Besides, the task of managing the secondary store is also made more complex if these clusters can have any arbitrary size.

An important limitation to the efficiency of pure prefetching schemes lies in the fact that prefetching policies do not affect the ultimate fate of pages after they are brought into memory. So, once a page is fetched, one cannot do anything to ensure that the page will stay in memory until it is referenced.

A better solution would thus be to make both fetch and replacement policies aware of cluster existence. Then, all memory allocation decisions would be made at the cluster level and the information gathered by the replacement policy on individual page behaviors could be used to modify dynamically the composition of clusters. Bennett and Franaiezek, for instance, have proposed a scheme, known as Permutation Clustering [2], where all transfers between the main memory and the secondary store involve fixed-size clusters of pages.
The composition of these clusters can be dynamically altered during program execution by permuting pages between blocks residing in memory. In another scheme, advanced by Pooch [18], dynamic clusters of "time and reference" related pages are built at run time. When a page fault occurs, all pages belonging to the cluster containing the faulting page are brought into memory while page frames holding pages not in that cluster are returned to the memory management system. Unlike Bennett and Franaczek's method, Pooch's algorithm does not attempt to modify the location of pages in the secondary store.

The main problem with such schemes lies in the time and space overhead required for the continuous monitoring of page references and the dynamic updating of page clusters. (On that account, Bennett and Franaczek's Permutation Clustering fares better than Pooch's algorithm, more sophisticated and therefore more cumbersome.)

These problems, inherent to all on-line page clustering algorithms prompted us to investigate the feasibility of off-line procedures. In this respect, a great deal of insight can be acquired by looking at program restructuring techniques.

Program restructuring [3] [12] [7] [9] [17] attempts to rearrange the various blocks of code or data constituting a program in such a way that blocks with the strongest mutual "affinities" will be stored into the same page or the same segment. Depending on the way these affinities are defined, several aspects of program performance, like its fault frequency or its memory occupancy, can be improved.
Because of the extremely high overhead required for continuously monitoring the behavior of programs at the block level \cite{10}, practically all program restructuring algorithms are strictly off-line procedures. The validity of this approach has been demonstrated in several experiments \cite{12} \cite{7} \cite{8} showing that the benefits of the restructuring process are rather insensitive to change in input data, making thus continuous monitoring of program behavior unnecessary.

3. An Alternative: Off-Line Clustering

3.1. General Philosophy

Off-line clustering attempts to improve the behavior of programs in paging environments by constructing off-line optimal—or near-optimal—clusters of pages that will be later handled as indivisible entities by the system's memory management policy. As page clusters are constructed off-line, our method requires very little modifications to the system's memory policies, which must only be aware of the page-to-cluster mapping for performing its fetch and replacement tasks. These modifications would also allow the implementation of lesser sophisticated schemes, like clustering only pages with contiguous virtual addresses, which may be indicated for programs a priori known to have a sequential behavior.

In order to transfer efficiently clusters of pages within the memory hierarchy, one has to ensure that all pages of each cluster are stored in contiguous locations on the secondary store. Although this will complicate the task of managing the secondary store, one should point that sequential block
allocation is also required by "warm start" fetch policies, which should be a part of any efficient virtual memory management scheme.

Anyways, the crucial part of the technique consists of the procedure used to identify clusters and this procedure will be therefore the focus of the remainder of the paper.

3.2. The Cluster Identification Procedure

The purpose of the cluster identification procedure is to find the best possible grouping of pages into clusters with respect to some index measuring the program's performance in a paged environment. This problem is essentially equivalent to the problem of restructuring a program to be run in a segmented virtual memory system [18], the only difference being that those elementary blocks considered in the segmented case can be of any arbitrary size.

Our approach will thus closely follow the one we sketched in [18] for constructing restructuring algorithms tailored to segmentation environments. First, we will gather information on the dynamic behavior of the program to be processed—ideally by collecting one or several page reference strings. Then we will use this information to build an affinity matrix, each element of which will represent the cost of keeping two pages in separate clusters. (A negative affinity between two pages thus signifies that the two pages should not belong to the same cluster.) Finally, a clustering algorithm is applied to the matrix and the resulting clusters identified.

As it has been consistently observed with program restructuring algorithms, the overall performance of such a scheme mostly depends on the algo-
ritthm used to construct the affinity matrix. Moreover, the grouping of two unrelated pages in the same cluster will normally increase both page traffic and memory occupancy and thus worsen the program's performance. It is therefore very important to base our definition of inter-page affinities on a comprehensive indicator of the program's performance. Among all popular indicators of program performance, the space-time product criterion is the only one to satisfy this condition. Since it had also been used very successfully in the context of program restructuring tailored to segmentation environments [18], it was only natural to make the same choice here.

3.3. Constructing the Affinity Matrix

The space-time product criterion measures the performance of a program in a virtual memory environment by the integral of its main memory storage costs over all time intervals during which the program was either running or waiting for a transfer from secondary store to main memory.

Unlike what is done for pure paging environments, we will have to take into account the fact that the time required to service a fault will depend here on the size $m$ of the cluster brought into memory at that time. More precisely, if $s_p$ is the page size, the average time $T_w$ required to service the fault will be given by

$$T_w = T_1 + m.T_1.s_p$$

where $T_1$ is the mean access time of the secondary store and $T_1$ the mean time to transfer one data unit.

Let now $S(u)$ denote the memory occupancy of a program at a given time $u$. The space-time product characterizing the behavior of the program being
executed in a paging environment with clustering during a virtual time interval
\((0, t)\) is given by

\[
C = \int_0^t S(u) du + \sum_{j=1}^r S(t_j) \cdot (T_{j+1} - T_j)
\]

where \(r\) is the total number of page faults occurring during \((0, t)\), \(t_j\) the time of the \(j\)-th page fault and \(m_j\) the size of cluster brought into memory at the \(j\)-th fault time.

Consider for a moment the behavior of a program before any attempt has been made to cluster pages into larger transfer units. For a given memory policy, it is normally possible to predict which pages of a program will reside in memory at any time of a given run of a program. Define the resident set of pages of a program at time \(t\) as the set of pages that are guaranteed to be present in memory while the \(t\)-th instruction is processed.

As we pointed out earlier, the decision to store two pages in the same cluster can have both beneficial and detrimental effects on the performance of the program. Since these effects will be directly reflected by corresponding variations of its space-time product, we can evaluate the resultant of these variations for each pair of pages \(i\) and \(j\) by examining the program's reference patterns. That value will be, by definition, the element \(a_{ij}\) of the clustering matrix.

Suppose, for instance, that page \(j\) is referenced while not being member of the current resident set of pages. In the absence of any clustering, this reference would necessarily result in a page fault. Suppose now that page \(j\) belonged to a cluster containing only pages that were in the same situation. Then, the cluster containing page \(j\) would not be present in memory and the
page fault become a *cluster fault*. On the other hand, should page \( j \) have belonged to a cluster containing at least one page that was currently in the resident set of pages, the cluster would have been present in memory and the page fault avoided. This would be reflected in the space-time product of the restructured program as a saving of

\[
\alpha = S(t)(T + T_p \cdot s_p)
\]

space-time units, where \( S(t) \) is the current memory occupancy of the program and \( s_p \) the page size.

Suppose now that page \( i \) belonged to a cluster \( k \) containing other pages and that some of them were active during a time interval \( \Delta t \) during which page \( i \) was inactive. Then page \( i \) would be residing in memory during that time interval although its presence in memory was not necessary. This would be reflected in the space-time product of the program as a waste of

\[
\beta = s_p \cdot \Delta t
\]

space-time units, where \( s_p \) is the page size.

Similarly, each time the cluster would be brought into memory because some page of that cluster, different from page \( i \), was referenced after having been inactive for a while, page \( i \) would be fetched into memory although the presence of page \( i \) in memory was not requested. This will result in an unwanted increase of the program's space-time product by

\[
\gamma = S(t_f)(T + T_p \cdot s_p)
\]

additional space-time units.

Thus,

(1) each time a page fault occurs, we should
increment by $S(t)T_i$ all the entries of $A$ that correspond to the pairs of pages containing a page already in memory and the page causing the page fault.

- decrement by $S(t_p)T_i s_p$ all the entries of $A$ that correspond to the pairs of pages containing a page not residing in memory and the page causing the page fault.

(2) at each reference, we should decrement by $s_p T_m$ all the entries of $A$ that correspond to the pairs where one page resides in memory and the other does not.

4. Formal Definition of the Algorithm

Let

$$(r_1,r_2,...,r_m)$$

be a page reference string collected during one run of the program to be restructured,

$s_p$ the page size,

$S(t)$ the memory space occupied by the program while processing the $t$-th reference (this size obviously depend on the page-to-cluster mapping).

$R_p(t)$ the resident set of pages at time $t$, i.e. while processing the $t$-th reference (we assume $R_p(1) = \{r_1\}$).

$T_m$ the mean inter-reference time,

$T_i$ the mean access time of the secondary store.

$T_t$ the mean time to transfer one data unit.

The affinity matrix $A = (a_{ij})$ has all zero entries initially and is constructed
in the following way:

(a) For all $t$ from 1 to $n$ do
   if $r_i \notin R_p(t-1)$ then (* page fault *)
   increment by $\alpha = S(t) \cdot (T_1 + T_r \cdot s_p)$ all $a_{ij}$'s such that $i \in R_p(t)$ and $j = r_i$;
   decrement by $\gamma = S(t) \cdot T_r \cdot s_p$ all $a_{ij}$'s such that $i \in R_p(t)$ and $j = r_i$;

(b) For all $i$ and all $j < i$ do
   $\alpha_{ij} := \alpha_{ji} := \alpha_{ij} + \alpha_{ji}$
   od.

Note that the algorithm we have described can be applied to all memory policies for which it is possible to construct the resident set of pages $R_p(t)$ and the memory space $S(t)$ occupied by the program at time $t$. To obtain the balanced clustering algorithm tailored to a specific memory policy, like the Balanced Working Set for the working set policy or the Balanced PSI for the global LRU policy, one has only to specify the proper expressions for $R_p(t)$ and $S(t)$.

5. Implementation Considerations

A few problems will arise with the above scheme when one attempts to implement a given balanced algorithm. First, let us point out that $S(t)$, which is appears in the expressions for $\alpha$ and $\beta$, depends on the page-to-cluster mapping and will thus be impossible to evaluate at restructuring time; the simplest solution will then be to replace $S(t)$ by a constant value $\bar{S}$ that will be some estimate of the program mean memory occupancy $\bar{S}$.

Another modification will involve the cost of running the algorithm. The most time-consuming part of the algorithm here will be the one where all ele-
ments \(a_{ij}\) of \(A\) corresponding to an \(i \not\in R_p(t)\) and \(j \not\in R_p(t)\) are decremented by \(s_p \cdot T_m\) after each reference. A sampling technique will then be used and the aforementioned routine will thus be performed once every \(K\) memory references.

A last modification and can be made whenever the secondary store is a disk-like device. These devices are essentially characterized by a significant access time \(T_i\) and a high transfer rate \(1/T_i\). One can thus neglect, as a first approximation, the contributions of the cluster sizes to the costs of cluster faults.

Keeping the same notations as before, the final version of our algorithm will then become:

(a) For all \(t\) from 1 to \(n\) do
   
   if \(i \not\in R_p(t-1)\) then (*page fault *)
   
   increment by \(\alpha = s_p \cdot T_i\) all \(a_{ij}\)'s such that \(i \in R_p(t-1)\) and \(j = r_i\); fi;

   if \(t \mod K = 0\) then (* sampling time *)
   
   decrement by \(\beta = s_p \cdot K \cdot T_m\) all \(a_{ij}\)'s such that \(i \not\in R_p(t)\) and \(j \in R_p(t)\)
   
   fi
   
   od:

(b) For all \(i\) and all \(j < i\) do
   
   \(a_{ij} := a_{ji} := a_{ij} + a_{ji}\)
   
   od.

Tailoring our algorithm to a given paging environment will thus essentially consists in inserting in the above algorithm the correct definition of the resident set of pages \(R_p(t)\). The problem is essentially equivalent to the one of defining a resident set of blocks for a given memory policy in a strategy-oriented memory policy. A complete coverage of that problem can be found in references [17] and [18].
6. Analytical Study of the Algorithm

We will only consider here the case of our algorithm applied to programs to be run under a working set policy. The resident set of pages of these programs is identical to their working set and contains all pages that have been referenced at least once during the last \( \tau \) time units, where \( \tau \), the \textit{window size}, is the policy's control parameter.

In many other memory policies, including FIFO, LRU, Global LRU and Page Fault Frequency, replacement decisions are (or may be) triggered by the occurrences of page faults. Since the most expected result of the clustering procedure is a reduction in the number of page faults, the clustering procedure will thus indirectly influence the fate of all pages, even if they do not belong to any cluster. For all of these policies but FIFO, it will remain possible to define a subset of the resident sets of pages \( R_p(t) \) such that all pages belonging to this subset will necessarily reside in memory, while some single pages residing in memory may not belong to that set \([17, 18]\). While being quite satisfactory in practice, this solution makes the analytical study of the algorithm very difficult \([17]\).

A more fundamental problem results from the fact that the restructuring graph only takes into account interactions between two pages, which leaves unanswered the question of defining affinities among more than two pages.

We defined earlier the affinity \( a_{ij} \) between two pages \( i \) and \( j \) as the total space-time product savings that could be achieved by storing the two pages \( i \) and \( j \) into the same cluster. It is then reasonable to assume that the affinity \( a_{ijk} \) among the three pages \( i, j \) and \( k \) should represent the total space-time
product savings that could be achieved by storing pages \(i, j\) and \(k\) into the same cluster.

It could happen that none of the expected effects of the clustering process would overlap. The affinity \(a_{ijk}\) is then equal to the sum all affinities between all pairs of pages in \(\{i, j, k\}\). We would have

\[ a_{ijk} = a_{ij} + a_{jk} + a_{ki} \]

and we would then speak of additive affinities.

However, this is not generally true. Suppose for instance that we have observed that storing page \(i\) with either page \(j\) or \(k\) will avoid a given page fault. Storing \(i\) with \(j\) and \(k\) will not save two page faults. In fact, we rather have

\[ a_{ijk} \leq a_{ij} + a_{jk} + a_{ki} \]

and no means to estimate \(a_{ij} + a_{jk} + a_{ki} - a_{ijk}\).

The simplest solution consists then of assuming that affinities will always add up and to define the affinities among \(s\) pages \(i_1, i_2, \ldots, i_s\) as being equal to

\[ a_{i_1, \ldots, i_s} = \sum_{j=1}^{s} \sum_{k=j+1}^{s} a_{i_j i_k} \]

Since the algorithm attempts then to maximize an optimistic estimate of the effects of the new page-to-cluster mapping, it tends to minimize a linear combination of a lower bound of the program's page fault frequency and an upper bound of its memory occupancy. We want to show here that it also minimizes linear combinations of upper and lower bounds of both indices.

We will suppose that the program to be restructured consists of \(m\) pages of sizes \(s_p\). After restructuring, these \(m\) pages will be partitioned into \(n\) clusters \(K_1, K_2, \ldots, K_n\).
Rather than restricting ourselves to a specific class of programs whose behavior can be described by a given stochastic model of program behavior, we will only assume that the program to be restructured behaves in a way that can be accurately described by a stochastic chain having a steady-state solution. From the practitioner's viewpoint, this assumption means that the program exhibits an essentially stable behavior, which should obviously be a prerequisite for any attempt to apply our method.

**Lemma 1:** Consider a program whose behavior can be described by a stochastic chain having a steady-state solution. If this program is running under a Working Set policy with a given page-to-cluster mapping \((K_1, K_2, \ldots, K_n)\), its page fault frequency will be bounded by

\[
\begin{align*}
\text{f} &= \sum_{i=1}^n \sum_{j \in K_i} \Pr[\tau_i = j \cap j \notin R_p(t-1)] \\
&\quad - \sum_{i=1}^n \sum_{j \in K_i} \sum_{K_k \neq j} \Pr[\tau_i = j \cap j \notin R_p(t-1) \cap k \in R_p(t-1)].
\end{align*}
\]

and

\[
\begin{align*}
\text{f} &= \sum_{i=1}^n \sum_{j \in K_i} \Pr[\tau_i = j \cap j \notin R_p(t-1)] \\
&\quad - \sum_{i=1}^n \sum_{j \in K_i} \frac{1}{\tau - 1} \sum_{k \in K_k \neq j} \Pr[\tau_i = j \cap j \notin R_p(t-1) \cap k \in R_p(t-1)].
\end{align*}
\]

where \(\tau\) is equal to the maximum number of pages per cluster.

**Proof:**

The page fault frequency of the program after restructuring will be equal to

\[
\begin{align*}
\text{f} &= \sum_{i=1}^n \sum_{j \in K_i} \Pr[\tau_i = j \cap j \notin R_p(t-1)],
\end{align*}
\]

which can be rewritten as
The first double sum on the right-hand side of the last equation is equal to the program's page fault frequency before clustering. The second double sum then represents the sum of the frequencies of all faults that have been avoided because of the clustering process.

Upper and lower bounds for (1) are respectively given by

\[ \sum_{i=1}^{n} \sum_{j \in K_i} \sum_{k \in K_i, k \neq j} \Pr[r_t = j \cap \not j \in R_p(t-1) \cap k \in R_p(t-1)]. \]

and

\[ \sum_{i=1}^{n} \sum_{j \in K_i} \frac{1}{\tau - 1} \sum_{k \in K_i, k \neq j} \Pr[r_t = j \cap \not j \in R_p(t-1) \cap k \in R_p(t-1)]. \]

where \( \tau \) is equal to the maximum number of pages per cluster.

**Lemma II:** Consider a program whose behavior can be described by a stochastic chain having a steady-state solution. If this program is running under a Working Set policy with a given page-to-cluster mapping \((K_1, K_2, \ldots, K_n)\), its mean memory occupancy will be bounded by

\[ \overline{S}_{\text{min}} = \sum_{i=1}^{n} \sum_{j \in K_i} s_p \Pr[j \in R_p(t)] + \frac{1}{\tau - 1} \sum_{i=1}^{n} \sum_{j \in K_i} \sum_{k \in K_i, k \neq j} s_p \Pr[k \in R_p(t) \cap \not j \in R_p(t)]. \]

and

\[ \overline{S}_{\text{max}} = \sum_{i=1}^{n} \sum_{j \in K_i} s_p \Pr[j \in R_p(t)] + \sum_{i=1}^{n} \sum_{j \in K_i} \sum_{k \in K_i, k \neq j} s_p \Pr[k \in R_p(t) \cap \not j \in R_p(t)]. \]

where \( \tau \) is equal to the maximum number of pages per cluster.
Proof:

After clustering, the program’s mean memory occupancy will be equal to

\[ \bar{S} = \sum_{i=1}^{n} S_i \Pr[ \bigcup_{j \in K_i} j \in R_p(t)] , \]

where \( S_i = s_p \text{card}(K_i) \) is the size of the \( i \)-th cluster. This expression can be rewritten as

\[ \bar{S} = \sum_{i=1}^{n} \sum_{j \in K_i} s_p \Pr[ j \in R_p(t)] + \sum_{i=1}^{n} \sum_{j \in K_i} s_p \Pr[ \bigcup_{k \in K_i, k \neq j} k \in R_p(t) \cap j \in R_p(t)] . \]

The first double sum on the right-hand side of the last equation is equal to the program’s mean memory occupancy before clustering. The second double sum

\[ \sum_{i=1}^{n} \sum_{j \in K_i} s_p \Pr[ \bigcup_{k \in K_i, k \neq j} k \in R_p(t) \cap j \in R_p(t)] . \]  

represents the increased memory occupancy resulting from the clustering process.

Upper and lower bounds for (2) are respectively given by

\[ \sum_{i=1}^{n} \sum_{j \in K_i} \sum_{k \in K_i, k \neq j} s_p \Pr[ k \in R_p(t) \cap j \in R_p(t)] . \]

and

\[ \sum_{i=1}^{n} \sum_{j \in K_i} \frac{1}{\tau - 1} \sum_{k \in K_i, k \neq j} s_p \Pr[ k \in R_p(t) \cap j \in R_p(t)] . \]

where \( \tau \) is equal to the maximum number of pages per cluster.

**THEOREM:** The simplified version of our clustering algorithm with full sampling and additive affinities minimizes linear combinations of lower and upper bounds for the number of faults and of the mean memory occupancy of all programs, running under a Working Set policy, whose behavior can be described by a stochastic chain having a steady-state solution.
Proof:

In the version of the algorithm being considered, all elements $a_{ij}$ of the clustering matrix are proportional to

$$\hat{S} T_i \cdot \Pr[i=r_i \cap i \notin R_p(t-1) \cap j \in R_p(t-1)] + \hat{S} T_i \cdot \Pr[j=r_i \cap j \notin R_p(t-1) \cap i \in R_p(t-1)] - s_p T_m \Pr[i \notin R_p(t) \cap j \in R_p(t)] - s_p T_m \Pr[j \in R_p(t) \cap i \notin R_p(t)]$$

The result of the clustering procedure will be a partition of the $m$ pages into $n$ clusters $K_1, K_2, ..., K_n$ maximizing

$$\sum_{i=1}^{m} \sum_{j \in K_i} \sum_{k \in K_j, k \neq j} a_{jk} =$$

$$\sum_{i=1}^{m} \left( \hat{S} T_i \cdot \sum_{j \in K_i} \sum_{k \in K_j, k \neq j} \Pr[j=r_i \cap j \notin R_p(t-1) \cap k \in R_p(t-1)] - s_p T_m \sum_{j \in K_i} \sum_{k \in K_j, k \neq j} \Pr[j \notin R_p(t) \cap k \in R_p(t)] \right)$$

Observing that

$$\sum_{i=1}^{m} \hat{S} T_i \sum_{j \in K_i} \Pr[j=r_i \cap j \notin R_p(t-1)]$$

and

$$\sum_{i=1}^{m} s_p T_m \sum_{j \in K_i} \Pr[j \in R_p(t)]$$

do not depend on the page-to-cluster mapping, we add them to the objective function and rewrite it either as

$$\min \sum_{i=1}^{m} \left( \hat{S} T_i \sum_{j \in K_i} \Pr[j=r_i \cap j \notin R_p(t-1)] + (r-1) s_p T_m \sum_{j \in K_i} \Pr[j \in R_p(t)] - \hat{S} T_i \sum_{j \in K_i} \sum_{k \in K_j, k \neq j} \Pr[j=r_i \cap j \notin R_p(t-1) \cap k \in R_p(t-1)] + s_p T_m \sum_{j \in K_i} \sum_{k \in K_j, k \neq j} \Pr[j \in R_p(t) \cup k \notin R_p(t)] \right)$$
which is equivalent to

\[ \min(\tilde{S} T_i f_{\min} + (r-1) T_m \tilde{S}_{\max}), \]

where \( r \) is the maximum number of pages per cluster, or as

\[
\min \sum_{i=1}^{m} [(r-1) \tilde{S} T_i \sum_{j \in K_i} Pr[j \neq r_i \cap i \notin R_p(t-1)] \\
+ s_p T_m \sum_{j \in K_i} Pr[j \in R_p(t)] \\
- \tilde{S} T_i \sum_{j \in K_i} \sum_{k \in K_i, k \neq j} Pr[j \neq r_i \cap i \notin R_p(t-1) \cap k \in R_p(t-1)] \\
+ s_p T_m \sum_{j \in K_i} \sum_{k \in K_i, k \neq j} Pr[j \in R_p(t) \cup k \notin R_p(t)]
\]

which is equivalent to

\[ \min((r-1)\tilde{S} T_i f_{\max} + T_m \tilde{S}_{\max}). \]

7. Empirical Results

A series of trace-driven simulations were conducted in order to evaluate the performance of our off-line page clustering algorithm under a working set policy. As we pointed out earlier, the Resident set of Pages \( R_p(t) \) for this policy contains all pages that have been referenced during the interval \( (t-\tau, t] \).

The traces we used for our experiments were full traces (instruction and data references) of a WATFIV compiler and an APL interpreter collected on an IBM 360/91 at the Stanford Linear Accelerator Center. Page sizes were 1,024 bytes for both programs. The WATFIV trace was one million reference long and the APL trace 2,300,000.

In order to limit the cost of our study, we decided to use compressed versions of the traces. The reduction algorithm used to generate the compressed traces replaced each trace by a sequence of "reference sets", each containing
the pages being referenced at least once during a time interval of 1,173 references. The whole process [16], very similar to the "Snapshot Method" described by Smith [19], is especially well suited to the study of program behavior in working set environments.

Since we were primarily interested in the mechanism used to define inter-page affinities, we decided to use for the clustering phase a very simple "greedy" algorithm combining at each iteration the two pages having the highest mutual affinities.

Besides the program, two other factors were considered in our experiments, namely

- the control parameter of the memory policy—here, the window size \( \tau \), and
- the fault cost coefficient \( \alpha \).

The performance indices measured were the number of faults, the total number of bytes brought in memory and the space-time product of the program. Corresponding values of these three indices for the WATFIV trace are presented in Figures I to III. On each graph, the solid line represents the behavior of the program for various window sizes before any attempt was made to cluster pages; points on this curve are thus labelled as "ORIGINAL". Each of the four interrupted lines corresponds to measurements that were made at a given window size—namely, 10, 20, 50 and 100ms—but with different values of \( \alpha \) varying between 0 and 1,000 byte-seconds.

Looking at Figure I, one can see that the clustering process can decrease the number of faults by at least 50\% without causing any corresponding increases of the program's memory occupancy. For three out of four window
sizes, this increase remains negligible as long as $\alpha$ remains inferior to 5 byte-
seconds. Figure II, on the other hand shows clearly that the total number of 
bytes fetched decreases much more slowly than the number of page faults. 
This observation, already made with program restructuring algorithms tailored 
to segmented environments, is easy to understand if one considers that the 
clustering consists essentially of merging the program’s pages into larger 
units. Therefore, one should expect to have, for a given memory occupancy, 
less page faults but a somewhat higher byte traffic.

The global effect of the clustering process on program performance can 
be evaluated by computing the space-time products of all versions of the pro-
gram. Figure III displays these values for a latency time $T_l$ of 10ms and a byte 
transfer time of $T_t=10^{-8}$s/byte. As one can see, the minima of the space-time 
product curves corresponding to the clustered version of the program are 
much below the curve corresponding to the original version of the program. 
All these four minima correspond to values of $\alpha$ in the neighborhood of 75 
byte-seconds.

Results obtained with the APL trace are very similar to—or even somewhat 
better than—these obtained with the WATTV trace. In particular, one can see 
on figure IV that the increases of the program’s memory occupancy remain 
almost negligible for small values of $\alpha$ while the program’s page fault frequen-
cies decrease by almost 50%. Figure V on the other hand, show a somewhat 
more erratic behavior for the number of bytes fetched. Finally, Figure VI 
shows very dramatic improvements of the space-time product of the clustered 
version of the program with the minima occurring for values of $\alpha$ in the vicinity
of 1,000 byte-seconds, which is much higher than the values observed for the WATFIV trace.

Experimental data on the performance of our clustering under other memory policies are still too fragmentary to be discussed here. Nevertheless, they already indicate that our scheme could also significantly improve the performance of programs to be run under a page fault frequency policy.

8. Conclusions

We have presented here a new off-line page clustering algorithm that can be tailored to various page replacement policies. The algorithm attempts to find a page-to-cluster mapping that minimizes the space-time product of the programs. Operating essentially off-line, it is much easier to implement than corresponding on-line schemes.

When applied to programs to be run under a working set policy, the algorithm simultaneously minimizes linear combinations of upper and lower bounds for page fault frequency and mean memory occupancy of all programs whose paging behavior can be described by a stochastic model having a steady-state solution. This was confirmed by empirical evidence obtained from trace-driven simulations, which showed that the method can substantially improve the performance of a WATFIV compiler and an APL interpreter running under a working set policy.

We think therefore that off-line clustering constitutes a very promising technique for improving the performance of often-used programs running in paging environments using a disk as secondary store.
Acknowledgements

The author wants to thank here Professor D. Ferrari from the University of California, Berkeley for his suggestions and encouragements as well as his friends, inside and outside the PROGRES group, for their support. He is also indebted to the COMPSAC '81 referees which helped to make the paper clearer.

References


Figure 1

WATFIV: Number of Faults vs. Memory Occupancy
Figure II
WATFIV: Byte Traffic vs. Memory Occupancy
Figure III
WATFIV: Space-Time Product vs. Memory Occupancy
Figure IV

APL: Number of Faults vs. Memory Occupancy
Figure V

APL Byte Traffic vs. Memory Occupancy
Figure VI
APL: Space-Time Product vs. Memory Occupancy