in: Proc. Int. Conf. on Extending Database Technology, LNCS 303, Springer, 1988, pp. 336-351

VAR-PAGE-LRU

A Buffer Replacement Algorithm Supporting Different Page Sizes

Andrea Sikeler

University of Kaiserslautern
Erwin-Schrödinger-Straße, D-6750 Kaiserslautern, West-Germany

Abstract

Non-standard applications (such as CAD/CAM etc.) require new concepts and implementation techniques at each layer of an appropriate database management system. The buffer manager, for example, should support either different page sizes, set-oriented operations on pages, or both in order to deal with large objects in an efficient way. However, implementing different page sizes causes some new buffer management problems concerning search within the buffer, buffer allocation, and page replacement. Assuming a global buffer allocation strategy, we introduce a page replacement algorithm which determines several pages stored in subsequent buffer frames to be replaced for a requested page. First investigations prove the algorithm to be a promising solution for buffer management with different page sizes.

1. Introduction

Although several modern operating systems provide a main storage file cache, most database management systems (DBMS) maintain their own database buffer for purposes of interfacing main memory and external storage devices (disks) /St81, CHMS87/. In order to facilitate the exchange of data between main memory and disk storage, the database is commonly divided into pages ("containers") of equal size (generally 512 to 4096 bytes). Typically, the database buffer consists of buffer frames of uniform size /EH84/. Therefore, the unit of data exchange is usually one page requested on demand (FIX operator). For each request (or logical reference /EH84/) the buffer manager has to perform several actions:

- The buffer is searched for the page and the page is located.
- If the page is not in the buffer, a replacement algorithm together with a buffer allocation strategy determines a buffer frame for the page. The "old" page in that buffer frame has to be replaced. Whenever this page has been modified, it has to be written back to disk before the requested page can be located in that buffer frame.
- The requested page is fixed in the buffer, and the address of the buffer frame is returned to the requesting DBMS component, i.e. to the access system.

While the page is fixed in the buffer, the access system may manipulate the contents of that page with simple machine instructions (such as COMPARE, MOVE, etc.). Therefore, replacement for that page is prevented until an explicit UNFIX operation makes the page eligible for replacement.

As a consequence of introducing pages as the unit of data exchange, in most existing DBMS (e.g. SYSTEM R /As76/) the size of objects manipulated by the access system (records or tuples) is limited by the page size of the buffer manager. Thus, modeling of application objects is affected by the page size yielding an unnecessary system restriction. In commercial applications objects are simple, being able to be described by a single record of limited size (less than approximately 2000 bytes). The objects of so-called non-standard applications (such as office automation, geographical data processing or CAD/CAM), however, are generally more complex, and often composed of other simple or complex objects (e.g. the boundary representation of a solid /BB84/). Even a simple object can be described by only a few bytes (e.g. a point in CAD/CAM) or by some Mbytes (e.g. an image or map in geographical data processing). Hence, all new data models for so-called non-standard DBMS (NDBMS), such as NF2 /SS86/, extended relational model /Lo84/, or MAD (molecule-atom data model) /Mi86/, include a data type LONG FIELD, BYTE VAR, etc. for very long attributes. Furthermore, all these models offer the possibility to build complex objects, either in a static or dynamic manner. As a consequence, such concepts have to be supported by all NDBMS components, especially by the access system and by the underlying buffer manager. The access system should be able to handle records spanning two or even more pages. It should cluster all records describing one complex object into one or more pages /DPS86, HR85/. Therefore, it should be possible to define a set of pages. These pages should be treated by the buffer manager as a "single page", i.e. the page set as a whole is transferred to and from the database buffer (e.g. by chained IO). In addition, it seems useful to support different page sizes, as the page size may then be defined in order to approximate the record size, and assuming page level locking in a multi-level concurrency control environment many fictitious synchronization conflicts can be avoided.

In the next section, we describe a possible interface, i.e. the objects and operations, of an NDBMS buffer manager which among other things supports different page sizes. Implementing different page sizes, however, causes some problems discussed in section 3. As a solution to these problems a new page replacement algorithm supporting different page sizes is introduced and investigated (section 4). First quantitative results were gained using slightly modified real page reference strings of CODASYL DBMS applications (section 5). A short summary concludes the paper.

2. Interface of an NDBMS Buffer Manager

The buffer manager described in this section is part of PRIMA (**pr**ototype **i**mplementation of the **m**olecule- **a**tom data model), an DBMS-kernel designed and implemented for non-standard applications (for a detailed description see / HMMS87/). PRIMA may be divided into three layers according to the chosen mapping hierarchy (molecules - atoms - pages - disks, Fig. 1). (Atoms are similar to records or tuples, whereas molecules are sets of heterogeneous atoms forming complex objects.) The data system is responsible for the composition and structuring of molecules, i.e. complex objects, which are built dynamically out of atoms. Atoms are mapped by the access system onto the different kinds of "containers" offered by the storage system. Each container may include various atoms which do not belong necessarily to the same molecule. However, molecules may be predefined using atom clusters, which allow for the common storage of several atoms in a single container, perhaps together with other atom clusters. As a consequence, the storage system as the lowest layer has no knowledge about the contents of the containers offered by itself. It solely pursues two major tasks: It manages the database buffer and organizes the external storage devices, thus being responsible for the data exchange between main storage and disk storage. Therefore, the storage system consists of two components: the buffer manager maintaining the database buffer and the file manager of the underlying operating system DISTOS /Ne87/ for data exchange.

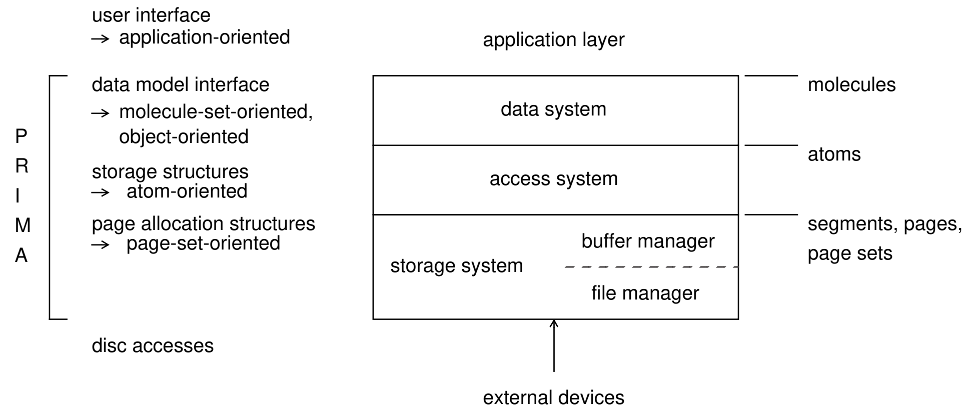


Figure 1: Architecture of a non-standard database system

This file manager offers some nice features regarding the requirements of non-standard applications:

- When creating a file one can choose from among five different block sizes (1/2, 1, 2, 4, and 8 kbytes). The chosen block size is kept fixed during the lifetime of a file.
- Files are dynamic. New blocks may be inserted anywhere and any existing blocks may be deleted.
- Operations on blocks are set-oriented. It is possible to read or write an arbitrary number of single blocks, a sequence of blocks, as well as the whole file.
- Blocks may be clustered. All blocks of a cluster are stored on disk in physical contiguity such that an efficient access to the whole cluster is feasible, e.g. by chained IO. Clusters are dynamic, i.e. they are bearranged whenever they grow or shrink.

Based on the objects and operations of the file manager, the PRIMA buffer manager, by managing the database buffer, provides a number of "infinite" linear address spaces with visible page boundaries. Therefore, the database is divided into various segments consisting of a set of logically ordered pages. Each segment is mapped onto a file and each page to a block. Thus, all pages of a segment are of equal size, which can be 1/2, 1, 2, 4, or 8 kbytes depending on the block sizes of the file manager and which is kept fixed during the lifetime of the segment, i.e. the PRIMA buffer manager supports **different page sizes**. Segments are dynamic. The segment-internal free place administration (FPA), which is integrated into the buffer manager, automatically allocates new pages when space is required which cannot be satisfied by existing pages. Similarly, the FPA automatically deletes empty pages when they reside at the end of the segment. Otherwise, empty pages are reused as soon as possible when space is required.

Hence, two different page types are distinguished within a segment. FPA pages are used only for free place administration. As the FPA is integrated into the buffer manager these pages are not available outside. Data pages, however, are used by the access system to store "user objects", i.e. physical records, address information, access path entries, and so on. They are requested by the FIX-PAGE operator indicating the fix mode, i.e. whether the page is needed for read or write purposes. Moreover, locking is integrated into the FIX-PAGE operator, i.e. the fix mode is used to lock the page in the appropriate mode. Repeated references of the same transaction possibly result in a lock conversion (read to write). Correspondingly, the UNFIX-PAGE operator indicates whether or not the page has actually been modified, i.e. the page has to be written to disk when it is replaced.

The five page sizes, however, do not meet all requirements of non-standard applications. The restriction to a certain page size, even 8 kbytes, is too stringent, especially regarding arbitrary length objects such as complex objects or strings (text, image) which may grow up to some MBytes. Therefore, **page sequences** are introduced as predefined page sets. A page sequence is a set of logical consecutive pages of a segment which contain (from the viewpoint of the access system) one single "user object" spanning these pages /DPS86/. A page sequence consists of a so-called header page and one or more component pages. Page sequences are dynamic since component pages may be added and removed arbitrarily depending on the current length of the "user object". To support physical clustering a page sequence is mapped onto a cluster at the file level. Requesting a page sequence (FIX-PAGE-SEQUENCE operator) causes the buffer manager to fix all pages in the database buffer belonging to that page sequence. These pages, however, are distributed over the buffer, i.e. the page sequence does not build one contiguous linear address space within the buffer, and the access system must be aware of the page boundaries. Correspondingly, the UNFIX-PAGE-SEQUENCE operator releases the whole page sequence, indicating for each page whether or not it has actually been modified.

Additionally, the PRIMA buffer manager offers three operators to handle not only such predefined page sets but also arbitrary page sets. The (UN)FIX-PAGE-SET operator is just a shorthand for a number of subsequent (UN)FIX-PAGE/(UN)FIX-PAGE-SEQUENCE operators. In other words, all specified pages and page sequences are fixed (unfixed) in the buffer. On the other hand, the FIX-ONE-PAGE operator supports access to replicated data fixing only one of the specified pages or page sequences, selecting that page or page sequence with minimum cost, i.e. the buffer manager performs a "page contest" along certain selection criteria (e.g. "already in buffer", page size, lock conflicts).

Regarding the objects and operations of the PRIMA buffer manager the question arises as to whether both *different page sizes* and *predefined page sets* should be supported. In order to answer this question we have decided to implement both concepts thus allowing for a thoroughgoing investigation of the pros and cons of both, e.g.:

- contiguous address space within the buffer versus distribution over the buffer,
- different limited page sizes versus the unlimited size of a predefined page set, and
- support of different block sizes versus support of a flexible cluster mechanism and chained IO by the file manager.

However, the decision as to which concept is best suited strongly depends on the structuring and mapping mechanisms in the application, in the data system, and in the access system (fixed or variable atom sizes, differences in the atom sizes, etc.) Therefore, in this paper we will concentrate on the problem of how to manage the database buffer when implementing different page sizes, whereas first results regarding predefined page sets supported by chained IO are summarized in /WNP87/.

3. Buffer Management Considering Different Page Sizes

Database buffer management includes three major actions which may be influenced using different page sizes: search within the buffer, buffer allocation, and page replacement.

There are different search strategies to locate a page in the buffer (either for a fix or an unfix operator) /EH84/. One of the most efficient is searching indirectly via a hash table. The hash algorithm transforms a segment number and a page number into a displacement in a page table, where an entry describing the page and its current position in the buffer can be found. In conventional DBMS, the current position of the page corresponds to the number of the buffer frame where the page is located. Using different page sizes a page generally occupies a number of subsequent frames depending on how the buffer is divided into frames. Thus, the table entry has to contain the actual number of frames occupied by the corresponding page.

In the case of a fix operation both the buffer allocation strategy and the page replacement algorithm are invoked in order to determine a buffer page to be replaced for the requested page. While the buffer allocation strategy establishes the set of candidate buffer pages from which a "victim" has to be taken, the page replacement algorithm decides which is the "victim". In conventional DBMS, buffer allocation strategies may be classified according to Fig. 2 /EH84/. Using different page sizes, a fourth class of allocation strategies should be considered. Instead of distributing the available buffer frames among the current database transactions or among different page types they are distributed among different page sizes, i.e. the database buffer is divided into different partitions, each containing pages of a single size only /Pa84/. Since once again partition sizes could be either static or dynamic, the same algorithms as in local or page-type oriented strategies may be used (e.g. Denning's working-set algorithm /De68/). Therefore, the same problems and disadvantages arise /EH84/. A static allocation, for example, is very inflexible in situations where the current workload changes frequently, whereas dynamic strategies are sensible regarding the chosen working-set parameters. Moreover, in the case of a dynamic strategy a still unsolved problem is how to reorganize the partitions in an appropriate way. Using a page-size oriented buffer allocation strategy, however, has one major advantage: The same replacement algorithms (and perhaps the same transaction oriented or page-type oriented allocation strategies) as in conventional DBMS can be used within each partition.

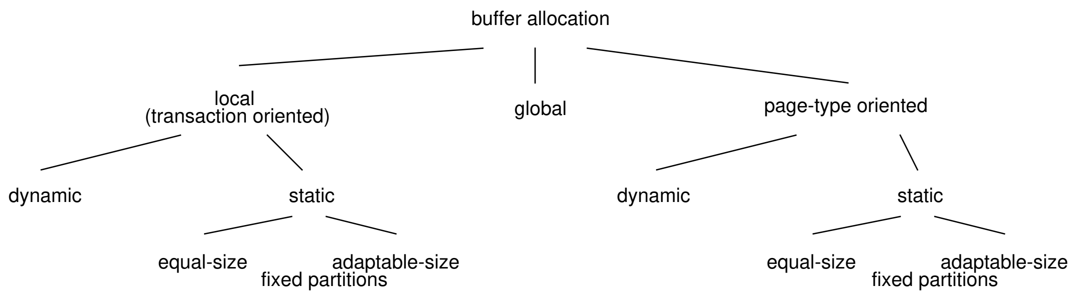


Figure 2: Classification of "conventional" buffer allocation strategies /EH84/

Nevertheless, we have decided to choose a global buffer allocation strategy, i.e. all pages which are not fixed at replacement time can be taken into account by the replacement algorithm, since such a strategy is simple and shows good results in conventional systems /EH84/. As a consequence, the replacement algorithm has to consider different page sizes. Since existing replacement algorithms (such as LRU etc. /EH84/) are only tailored to one page size, we either had to design a new one or to redesign an old one. In PRIMA, we have chosen the second alternative, i.e. the replacement algorithm described in the following section is based on the well-known LRU (least recently used) algorithm. The LRU algorithm was chosen since it has proven well suited in conventional DBMS /EH84/.

4. The VAR-PAGE-LRU Algorithm

VAR-PAGE-LRU was designed for a (database) buffer divided into frames of equal size and for pages of different sizes which are a multiple of the frame size. That is, each page is stored in a certain number of consecutive frames, in order to guarantee relative addressing within the page. As a consequence, VAR-PAGE-LRU has to cope with two problems.

VAR-PAGE-LRU, such as the classical LRU algorithm, initially tries to choose the oldest page (i.e. earliest unfix time) for replacement. However, using different page sizes, two facts have to be considered. On the one hand, pages may be replaced by smaller ones. Thus, free buffer frames arise which results in a **buffer fragmentation** and a perhaps low buffer occupancy. Therefore, VAR-PAGE-LRU has to reuse free buffer frames which are managed in a so-called free-list, in order to minimize buffer fragmentation and to maximize buffer occupancy. On the other hand, in some cases it does not suffice to replace a single page. Rather **multiple buffer pages** have to be replaced in order to achieve enough consecutive free frames for the requested page. These pages, however, should be as old as possible with respect to the LRU property. Therefore, VAR-PAGE-LRU proceeds the LRU-chain backwards starting with the oldest element and marks for each list element those buffer frames as free, which are occupied by the corresponding page until sufficient consecutive buffer frames are marked. Hence, all corresponding pages have to be replaced. However, it may happen that the number of marked buffer frames is too large and that perhaps too many pages are replaced (e.g. when marking combines two regions of already marked buffer frames). Thus, in a second step VAR-PAGE-LRU determines the minimum number of the pages to be replaced. Again the oldest pages should be selected. So, VAR-PAGE-LRU pursues two goals:

(1) The pages being replaced have to be as old as possible (LRU property).

(2) The number of pages to be replaced has to be as small as possible (minimizing IO).

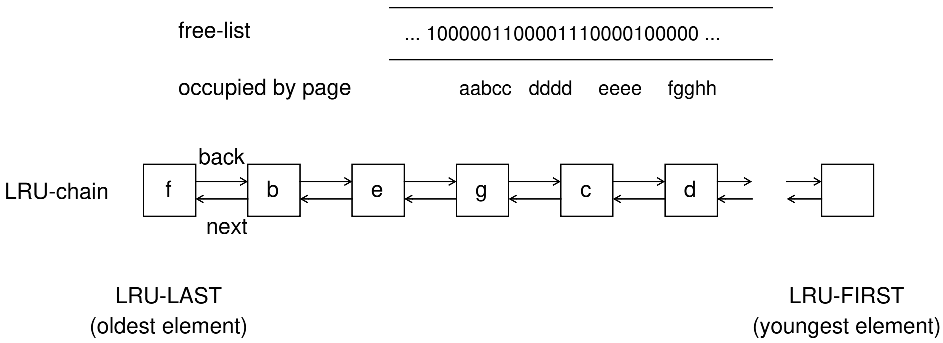


Figure 3: Data structures of VAR-PAGE-LRU

The **data structures** used by VAR-PAGE-LRU are, as already mentioned, an LRU-chain and a so-called free-list. The *LRU-chain* contains all entries of the page table where the corresponding page is unfixed. The entries are sorted according to the time of the last unfix operation on the page. So, "least recently used" is interpreted as "least recently unfixed" and not as "least recently referenced" /EH84/. The beginning and the end of the LRU-chain are indicated by LRU-FIRST and LRU-LAST, respectively. The *free-list* is a bitlist containing one bit for each buffer frame. This bit indicates whether the corresponding frame is free ("1") or occupied ("0"). Occupied means that the frame itself or a consecutive number of frames it belongs to represents a page with a corresponding entry in the page table, whereas free means that the frame may contain a part of a previously replaced page. Fig. 3 shows a corresponding scenario. Additional data structures for buffer management are transaction oriented FIX-chains and a changed-chain. The first contains all entries of the page table in which the corresponding page is fixed by the appropriate transaction, whereas the second contains those entries in which the page has to be written to disk in the case of replacement.

Thus, the first action of VAR-PAGE-LRU (Fig. 4) is to search the free-list for a number of consecutive free frames sufficient for the requested page. If such a sequence exists, no page has to be replaced. Otherwise, the replacement algorithm starts to determine one or more consecutive pages within the buffer which then have to be replaced to get enough free place (perhaps in combination with free buffer frames) for the requested page. Therefore, VAR-PAGE-LRU proceeds in two steps:

```
algorithm:   position := search free-list
             IF   position = 0
                  THEN   copy free-list into work-list
                         actual-page := lru-last
                         range-found := FALSE
                         WHILE NOT (range-found) AND NOT (end of LRU-chain)
DO step 1
                         IF   range-found
                              THEN  IF   position = 0
                                         THEN   step 2
                              ELSE error

step 1:      IF   actual-page + surrounding free frames of free-list >= #-of-frames
                  THENadd actual-page to replace-list
                         position := first free frame of range
                         range-found := TRUE
                  ELSE   IF   actual-page + surrounding free frames of work-list >= #-of-frames
                              THEN   free actual-page in work-list
                                     range := first to last free frame of range
                                     range-found := TRUE
                              ELSEfree actual-page in work-list
                                     actual-page := back (actual-page)

step 2:      actual-page := next (actual-page)
             add back (actual-page) to replace-list
             WHILE actual-page <> NIL DO
                  WHILE NOT (actual-page within range) DO actual-page := next
(actual-page)
                  IF   left side of range without actual-page >= #-of-frames
                       THENrange := left side of range without actual-page
                              actual-page := next (actual-page)
                       ELSE   IF   right side of range without actual-page >= #-of-frames
                                   THEN   range := right side of range without actual-page
                                          actual-page := next (actual-page)
                                   ELSEactual-page := next (actual-page)
                                          add back (actual-page) to replace-list
```

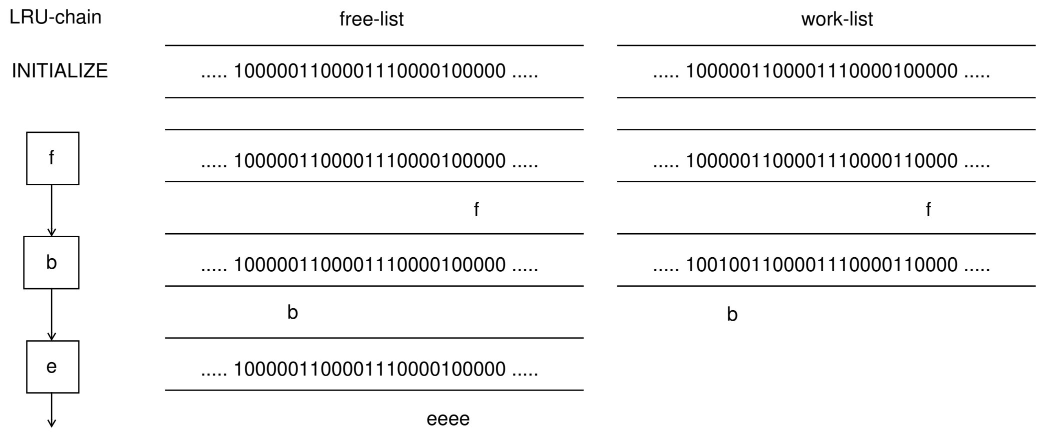
Figure 4: Algorithm VAR-PAGE-LRU

Step 1: Searching for a sufficient range of free frames (Fig. 5)

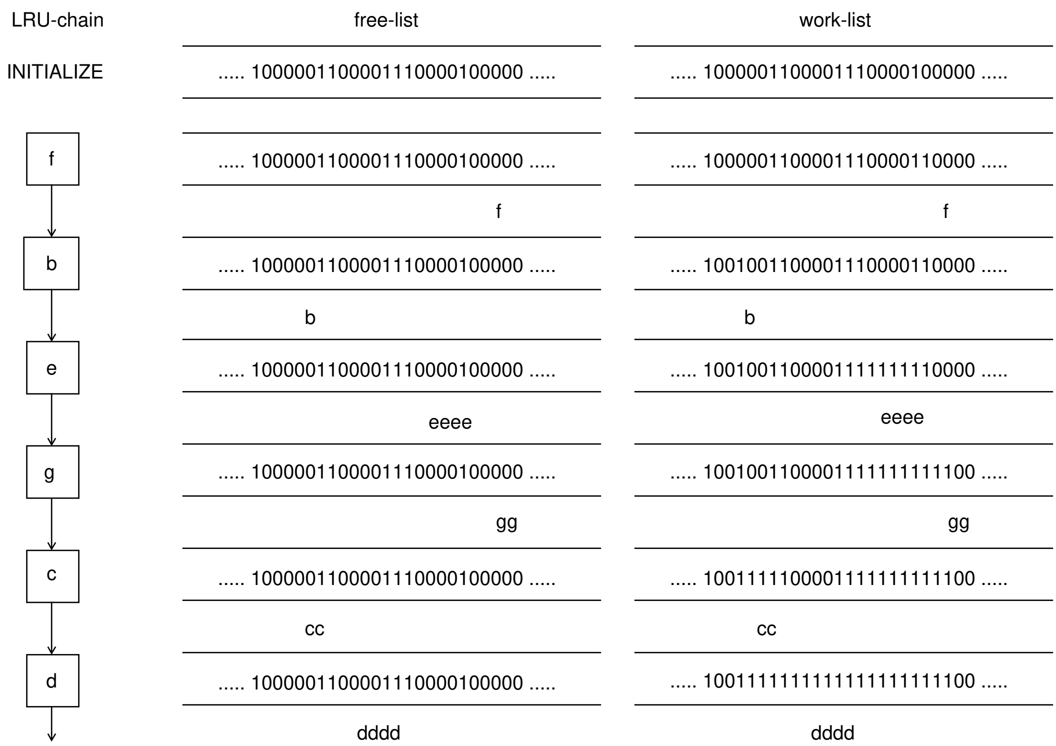
In a first step the LRU-chain is searched, starting with the oldest page, until enough consecutive pages are found to satisfy the request. For that purpose, a second bitlist, the so-called work-list, is used to simulate the behaviour of the free-list by testing the consequences of freeing more than one page before actually doing so. Hence, the work-list is initialized by the current free-list. The LRU-chain is then searched. For each page, it has to be examined to see if the frames occupied by the page, perhaps in combination with neighbouring free frames from the free-list, are sufficient for the requested page. In this case, this is the only page to be replaced, and the algorithm stops (Fig. 5a). The same check is done with the work-list, and the algorithm turns over to the second step if there is enough space (Fig. 5b). Otherwise, the frames occupied by that page are marked as free in the work-list, and the next page of the LRU-chain is examined.

Step. 2: Searching for a set of pages to be replaced (Fig. 6)

The second step is only necessary when more than one page must be replaced. The space accumulated in the work-list can be larger than the space needed, but replacement should select only a minimum number of pages and should preserve the LRU property. Thus, actual replacement proceeds the other direction through the LRU-chain starting with the last page of the first step. Again each page is examined to see if its frames correspond to the free frames detected in the first step. In this case, it is checked as to whether the page is really needed to build the free frames sufficient for the requested page. If not, the page is deleted from the work-list by marking the corresponding buffer frames as occupied, and the algorithm continues with the remaining number of free frames and the next page of the LRU-chain. Otherwise, the page is put on a list containing all pages to be replaced and the algorithm goes on with the next page.

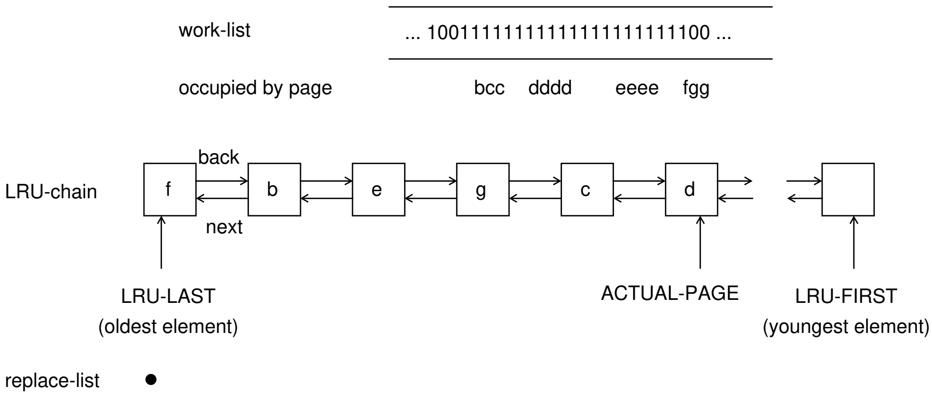


a) searching for 8 subsequent free buffer frames: replace page e

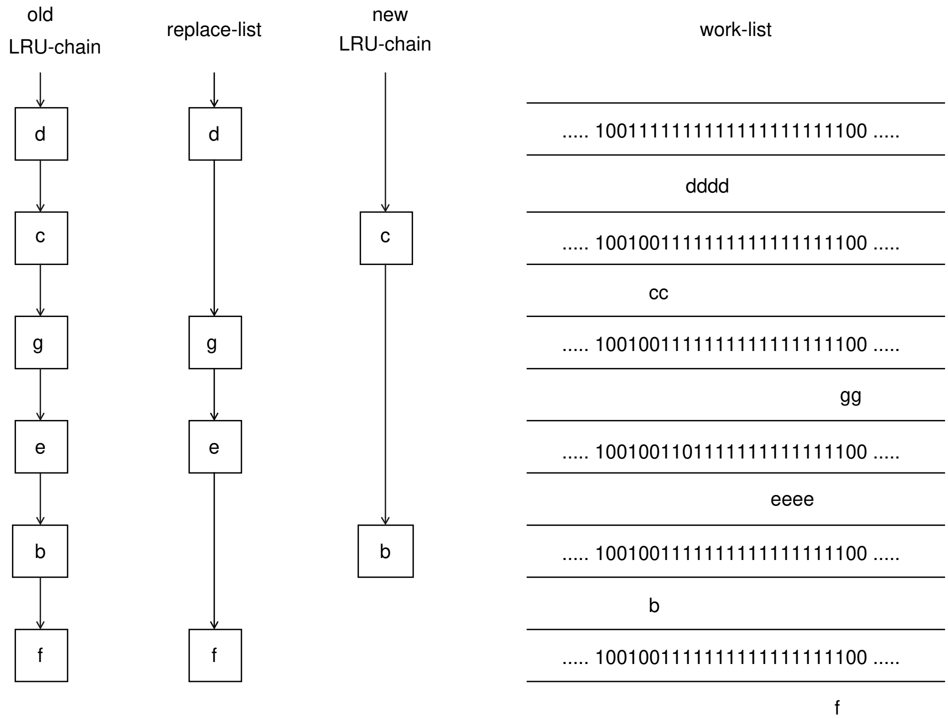


b) searching for 16 subsequent free buffer frames: turn over to step 2

Figure 5: Step 1 of VAR-PAGE-LRU using Fig. 4 as starting scenario



a) starting scenario (see Fig. 5b)



b) VAR-PAGELRU stops with pages f, e, g, d to be replaced

Figure 6: Step 2 of VAR-PAGE-LRU

Fig. 5b and 6 show a complete example of VAR-PAGE-LRU processing. The requested page needs 16 buffer frames. During the first step of the algorithm the pages f, b, e, g, c, and d are added to the work-list in the given order (Fig. 5b). The first step stops with page d, since including this page results in 20 subsequent free buffer frames. During the second step the pages are examined in the opposite direction (d, c, g, e, b, f). Page d is always put on the list containing all pages to be replaced. Page c is removed from the work-list since the remaining number of 17 free frames is still sufficient. As a side effect, page b falls out of the remaining number of examined free frames. Pages g, e, and f are also put on the list. Hence, the number of pages to be replaced is as small as possible and the pages being replaced are as old as possible.

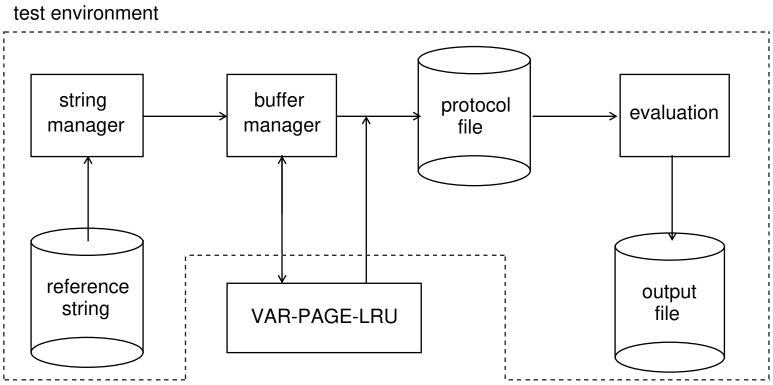


Figure 7: Test environment for VAR-PAGE-LRU

5. Evaluation and Optimization

Before including VAR-PAGE-LRU into PRIMA, the algorithm was implemented in PL/1 and embedded into a test environment (Fig. 7) in order to gain some information about its over-all behaviour. The test environment allows for the execution of a logical reference string consisting of multiple fix and unfix operations. The test buffer manager implements only those tasks which are necessary in order to estimate VAR-PAGE-LRU. During the execution of a string the buffer manager, as well as VAR-PAGE-LRU itself, write different measurement data on a protocol file which may be prepared in different ways. The most interesting information, however, are data concerning

- buffer occupancy (minimum, maximum, average),
- buffer fault rate,
- the number of LRU pages processed during a page fault, and
- the number of pages to be replaced.

For our investigations we used different real reference strings from CODASYL DBMS applications. Although these reference strings come from conventional applications in a CODASYL environment and therefore might be less realistic for non-standard applications, it was the simplest way to gain large reference strings in order to obtain first results about the behaviour of VAR-PAGE-LRU. For additional investigations, however, it will be necessary to generate synthetic reference strings to obtain more detailed results by varying the reference strings in an appropriate way. Nevertheless, the results illustrated in this section are restricted to the CODASYL reference strings and thus have to be reviewed under these circumstances.

For each of the strings the buffer size was varied from 0.5 Mbytes up to 6 Mbytes (step-width 0.5 Mbytes) using a fixed frame size of 0.5 kbytes. Since the original strings only refer to a single page size, the number of different page sizes, as well as the page sizes themselves, could be chosen randomly. Therefore, we executed measurements for page sizes

(1) which are a multiple of 1, 2, ..., 10 of the frame size and

(2) which are a multiple of 1, 2, 4, 8, 16 of the frame size (according to the page sizes of PRIMA).

VAR-PAGE-LRU demonstrates a very similar behaviour for each of the CODASYL reference strings. Hence, the most important results will be illustrated only for one of the strings. This string consists of 55 560 logical references on 7 different segments and about 7 000 different pages. To generate the page size of a segment the corresponding segment number was used:

(1) page size = (segment number mod 10) + 1 * frame size

(2) page size = $2^{\text{segment number mod 5}}$ * frame size

Fig. 8 shows the resulting distributions of the logical references on page sizes. For example, about 40 % of the references refer to a page size of 5 frames for (1) and to a page size of 16 frames for (2). As a consequence, the average page size of the alternatives differ from each other (5.4 frames for (1) and 9.4 frames for (2)). However, these distributions might not be realistic at all, but what are characteristic reference strings in a non-standard environment.

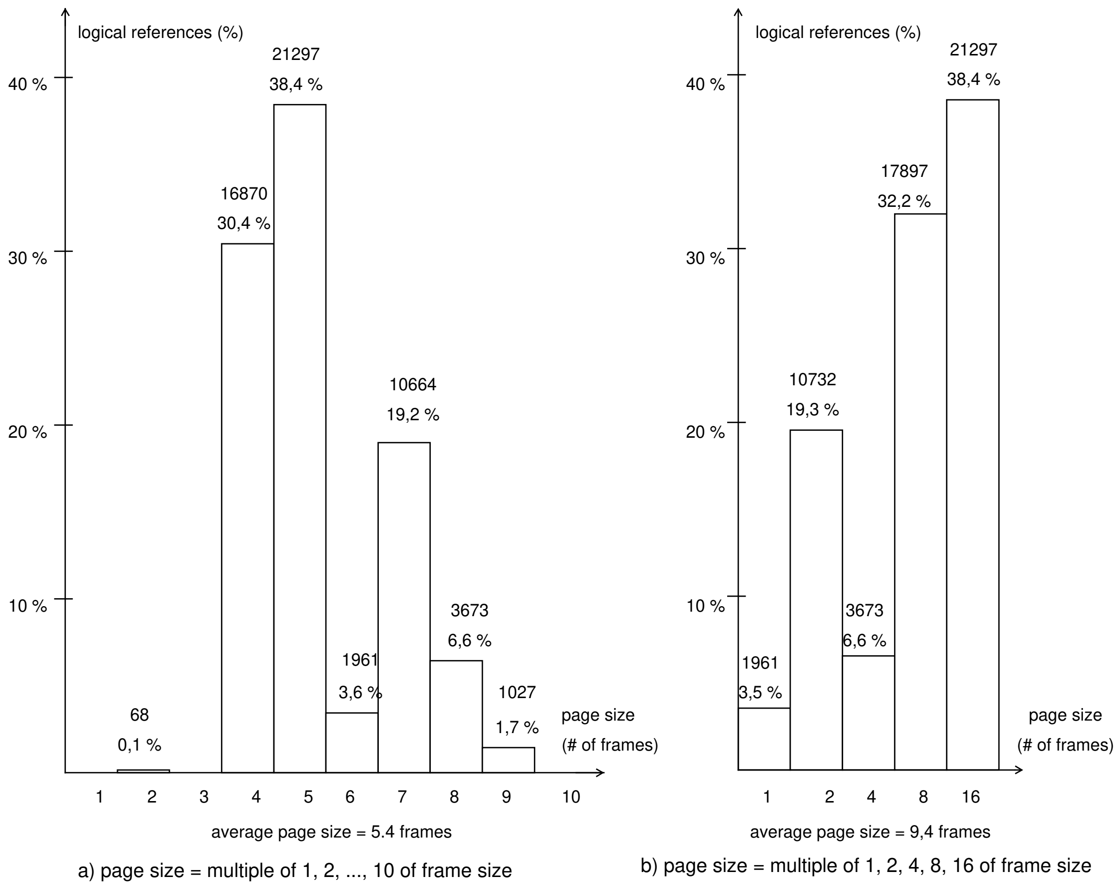


Figure 8: Number of logical references per page size (length of string: 55 560 references)

The results of VAR-PAGE-LRU concerning **buffer occupancy** proved to be very promising (Fig. 9). On the average about 99 % for (1) and 100 % for (2) of the buffer frames are occupied (independent of the buffer size). The minimum buffer occupancy varies between 87.2 % (0.5 Mbytes) and 92.2 % (6 Mbytes) for (1) and between 97.6 % and 99.4 % for (2), whereas the maximum buffer occupancy is always 100 %. The difference between (1) and (2) probably results from the different page sizes. Buffer fragmentation arises when a page larger than the requested one has to be replaced. However, for (2) the page being replaced is at least twice as large as the requested one, i.e. a relatively high number of frames is freed. Therefore, the probability to reuse free frames grows.

For comparison we implemented a straight-forward solution of a static page-size oriented buffer allocation strategy: The buffer is divided into partitions according to the number of different page sizes (10 partitions for (1) and 5 partitions for (2)). Each partition consists of the same number of frames of the appropriate size, i.e. each partition may include the same number of pages. In contrast to VAR-PAGE-LRU, the corresponding results regarding buffer occupancy are substantially worse (Fig. 9). Moreover, buffer occupancy becomes worse as buffer size grows. For instance, using a 6 Mbyte buffer, the minimum buffer occupancy of VAR-PAGE-LRU is about 92 % for (1), whereas partitioning results in a buffer occupancy of 31 %. The reason is that partition sizes are defined independently of the current workload. Hence, some partitions are never or only partially used. This arises in particular for (1) (see Fig. 8).

buffer size			1000	2000	3000	4000	5000	6000	7000	8000	9000	10000	11000	12000
VAR-PAGE-LRU	min	# of frames	872	1806	3633	2724	4515	5439	6367	7339	8204	9160	10088	11067
		%	87,2	90,3	90,8	90,8	90,3	90,7	91,0	91,7	91,2	91,6	91,7	92,2
	avg	# of frames	987	1978	2969	3960	4948	5942	6933	7928	8923	9924	10916	11907
		%	98,7	98,9	99,0	99,0	99,0	99,0	99,0	99,1	99,1	99,2	99,2	99,2
Partitions		# of frames	519	807	1095	1390	1687	1980	2268	2551	2846	3139	3431	3719
		%	51,9	40,4	36,5	34,8	33,7	33,0	32,4	31,9	31,6	31,4	31,2	31,0

a) page size = multiple of 1, 2, ..., 10 of frame size

buffer size			1000	2000	3000	4000	5000	6000	7000	8000	9000	10000	11000	12000
VAR-PAGE-LRU	min	# of frames	976	1969	2970	3970	4969	5969	6970	7968	8953	9948	10959	11926
		%	97,6	98,5	99,0	99,3	99,4	99,5	99,6	99,6	99,5	99,5	99,6	99,4
	avg	# of frames	1000	2000	3000	4000	5000	6000	7000	8000	8999	9999	10999	11999
		%	100	100	100	100	100	100	100	100	100	100	100	100
Partitions		# of frames	891	1731	2571	3405	4245	5085	5925	6761	7601	8441	9281	10115
		%	89,1	86,6	85,7	85,1	84,9	84,9	84,6	84,5	84,5	84,4	84,4	84,3

b) page size = multiple of 1, 2, 4, 8, 16 of frame size

Figure 9: Buffer occupancy

The results concerning the **buffer fault rate** (Fig. 10) also differ for (1) and (2) according to the different average page size. Reference measurements using constant page sizes point out that VAR-PAGE-LRU behaves similarly to the original LRU algorithm (whose behaviour regarding an optimal replacement is known /EH84/). As an example, the buffer fault rate for (1) (average page size = 5.4 frames) is somewhat higher than that for a constant page size of 2.5 kbytes (= 5 frames), whereas the buffer fault rate for (2) (average page size = 9.4 frames) is somewhat lower than that for a constant page size of 5 kbytes (= 10 frames). However, in any case the buffer fault rates of VAR-PAGE-LRU are always smaller than those using different partitions, again for the given CODASYL reference strings.

An important aspect with respect to the time for executing a page fault is the **number of LRU pages processed** during a single page fault. Fig. 11 summarizes the corresponding results which strongly differ for (1) and (2). Whereas for (1) most of the time (86.0 % to 53.4 %) only up to ten pages have to be examined, for (2) in up to 39.9 % of the cases more than 100 pages are touched. This results from the different distributions of the logical references on page sizes as well as from the sequence of the logical references, i.e. the access pattern at the buffer manager interface may strongly influence the number of LRU pages processed. However, synthetic reference strings should be used in order to obtain more detailed results.

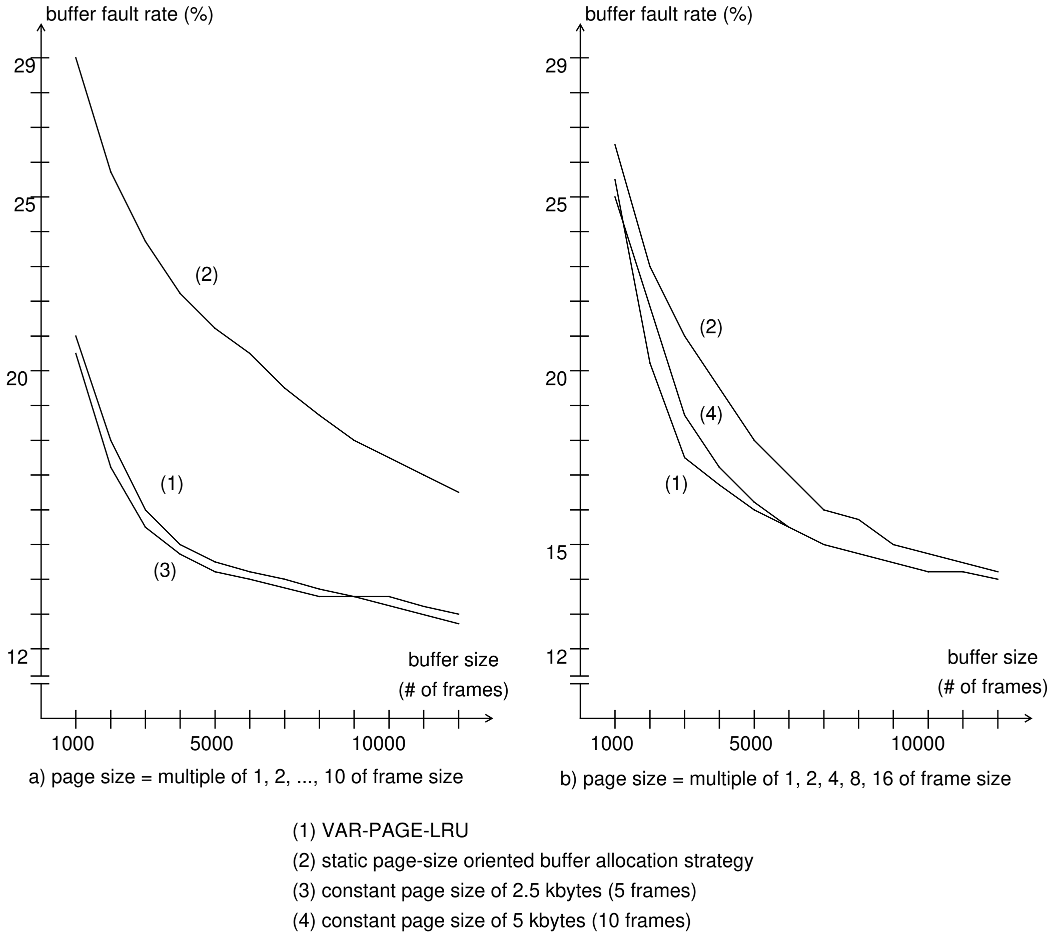


Figure 10: Buffer fault rate

Consequently, VAR-PAGE-LRU had to be improved with respect to the number of LRU pages processed during a page fault. Therefore, so-called page-size oriented pointers were introduced. Until then step 1 of VAR-PAGE-LRU always started with the last page of the LRU-chain (Fig. 4), i.e. during the execution of a number of subsequent page faults, for the same page size, the same LRU pages are examined again and again although it is obvious that these pages are not sufficient. Thus, the page-size oriented pointers indicate the first page not yet examined for the corresponding page size. Those pointers are the new starting-point of step 1. However, before step 1 all LRU pages from LRU-LAST up to the appropriate pointer are entered into the work-list. As a result, a substantial improvement for (2) as well as for (1) is achieved (Fig. 11), i.e. even for (2) most of the time (88.8 % to 68.2 %) only up to 10 pages are touched.

Moreover, the **number of pages to be replaced** during a page fault influences the time for executing a page request, since IO is the most critical point. However, VAR-PAGE-LRU shows a satisfactory behaviour with regard to this (Fig. 12). Most of the time only one page has to be replaced (92.9 % to 82.8 % for (1) and 82.8 % to 60.7 % for (2)). Additionally, some page faults can be satisfied without replacing any page (3.4 % to 6.0 % for (1) and 10.8 % to 30.7 % for (2)). Therefore, in only a few of the cases has more than one page to be replaced. Furthermore, for (1) a maximum of 3 pages is affected (3 times), whereas for (2) up to 8 pages have to be replaced (13 times). This difference between (1) and (2) also results from the different page sizes.

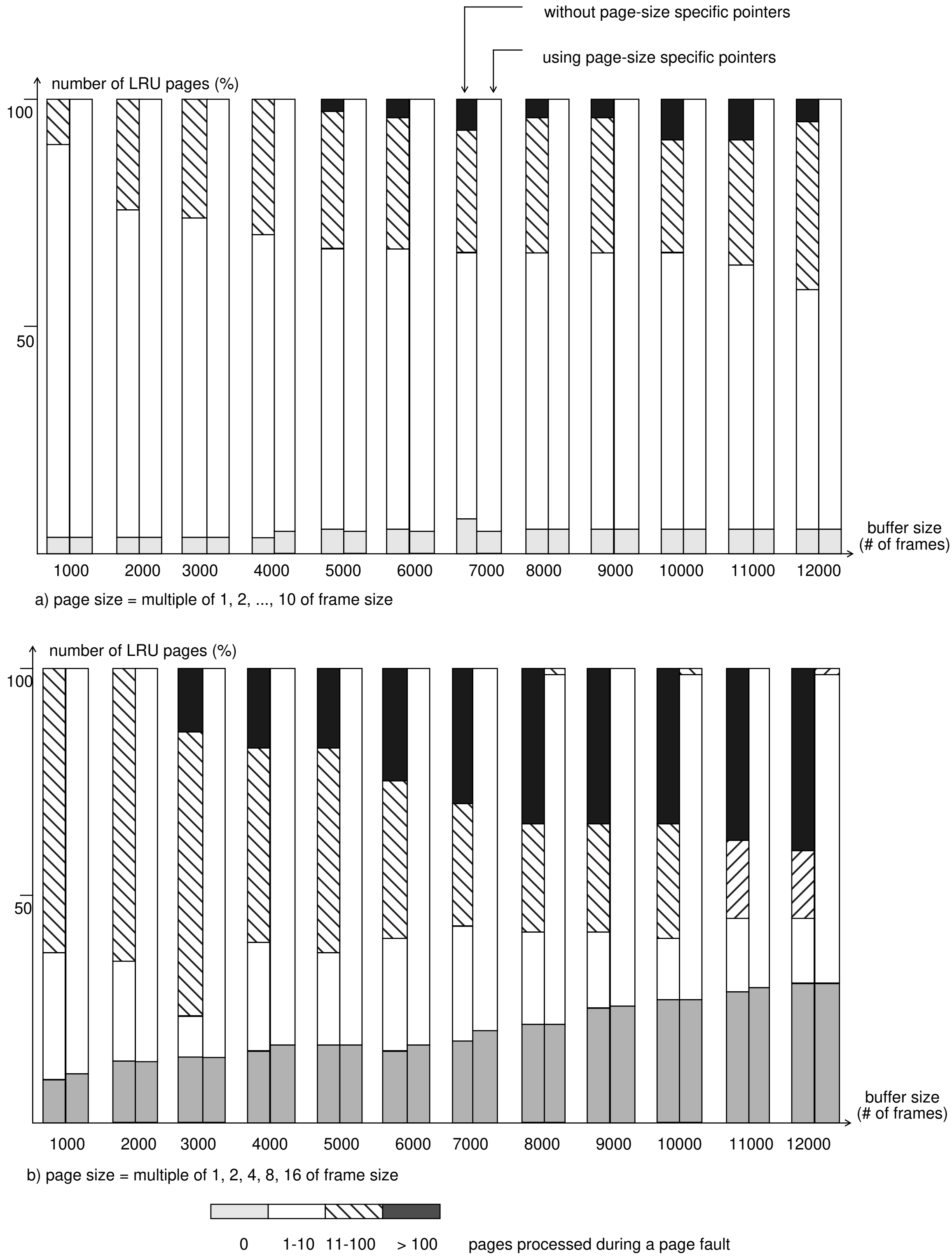
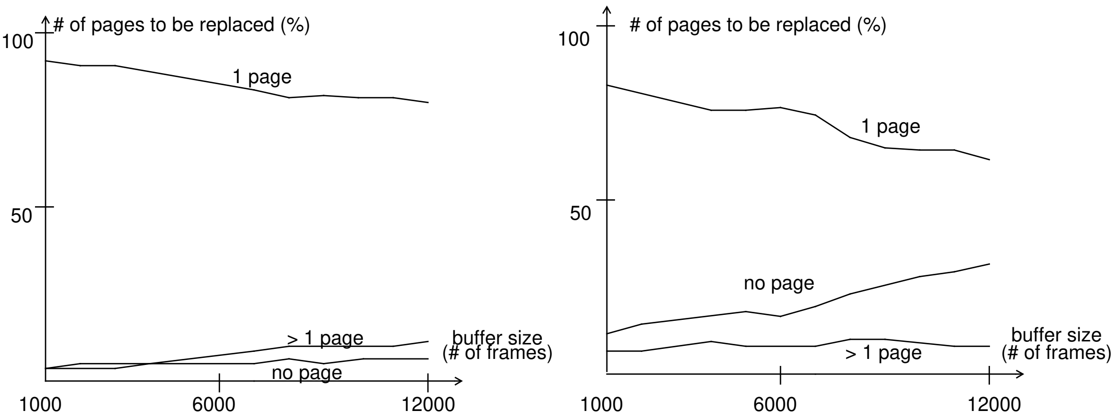


Figure 11: number of LRU pages processed during a page fault



a) page size = multiple of 1, 2, ..., 10 of frame size

b) page size = multiple of 1, 2, 4, 8, 16 of frame size

(without page-size specific pointers)

Figure 12: Number of pages to be replaced

6. Conclusions

Non-standard applications require more flexibility with respect to the objects and operations offered at the different layers of an appropriate NDBMS. The buffer manager, for instance, has to support set-oriented operations on pages as well as different page sizes in order to support large objects of variable length. However, implementing different page sizes causes some problems concerning buffer management, i.e. search within the buffer, buffer allocation, and page replacement. Assuming a global buffer allocation strategy we have introduced a page replacement algorithm called VAR-PAGE-LRU. VAR-PAGE-LRU determines one or more consecutive pages within the buffer which have to be replaced in order to create enough free place for a requested page. First investigations using reference strings from CODASYL DBMS applications have proved that VAR-PAGE-LRU is an appropriate and efficient solution. The buffer occupancy as well as the buffer fault rate are, at least for these reference strings, much better compared to those for a static page-size oriented buffer allocation strategy. Moreover, the buffer fault rate corresponds to that of the original LRU algorithm for the appropriate average page size. Hence, the overhead for searching a set of pages to be replaced is accidental. Consequently, we have integrated VAR-PAGE-LRU into the DBMS kernel PRIMA in order to allow for additional investigations with respect to non-standard applications.

However, the question remains whether both different page sizes and predefined page sets should be supported. And if so, how VAR-PAGE-LRU will behave in the case of handling page sets. Certainly, indepth investigations concerning the area of handling page sets /WNP87/ in combination with VAR-PAGE-LRU are necessary. Furthermore, VAR-PAGE-LRU itself has to be analysed in more detail using synthetic reference strings and new ideas such as moving unfixed pages within the buffer have to be reflected and perhaps integrated into the algorithm.

Acknowledgements

I would like to thank M. Michels who implemented VAR-PAGE-LRU and the test environment. Thanks are also due to T. Härder, E. Rahm, B. Mitschang, and I. Littler for helpful commments and editorial suggestions which helped to improve the presentation of this material. Furthermore, I would like to thank the referees for their fruitful comments and the inspiration of new ideas (e.g. moving pages within the buffer).

References

As76 Astrahan, M.M., et al.: SYSTEM R: A Relational Approach to Database Management, in: ACM Transactions on Database Systems, Vol. 1, No. 2, June 1976, pp. 97-137.

BB84 Batory, D.S., Buchman, A.P.: Molecular Objects, Abstract Data Types and Data Models: A Framework, in: Proceedings of the 10th International Conference on Very Large Databases, Singapore, 1984, pp. 172-184.

CHMS87 Christmann, H.-P., Härder, T., Meyer-Wegener, K., Sikeler, A.: Operating System Support for Database Management Systems, to appear in: Proceedings of the Workshop on "Experiences with Distributed Systems", Kaiserslautern, 1987.

De68 Denning, P.J.: The Working Set Model for Program Behaviour, in: Communications of the ACM, Vol. 11, No.5, 1968, pp. 323-333.

DPS86 Deppisch, U., Paul, H.-B., Schek, H.-J.: A Storage System for Complex Objects, in: Proceedings of the International Workshop on Object Oriented Database Systems, Asilomar, ed.: K. Dittrich, U. Dayal, 1986, pp. 183-195.

EH84 Effelsberg, W., Härder, T.: Principles of Database Buffer Management, in: ACM Transactions on Database Systems, Vol. 9, No. 4, 1984, pp. 560-595.

HMMS87 Härder, T., Meyer-Wegener, K., Mitschang, B. Sikeler, A.: PRIMA - a DBMS Prototype Supporting Engineering Applications, to appear in: Proceedings of the 13th International Conference on Very Large Data Bases, Brighton, 1987.

HR85 Härder, T., Reuter, A.: Architektur von Datenbanksystemen für Non-Standard-Anwendungen (Architecture of Database Systems for Non-Standard Applications), in: Proceedings of the GI Conference on Database Systems for Office, Engineering, and Science Environments, Karlsruhe, ed.: A. Blaser, P. Pistor, Informatik-Fachberichte No. 94, Springer, Berlin Heidelberg New York Tokyo, 1985, pp. 253-286.

Lo84 Lorie, R., et al.: Supporting Complex Objects in a Relational System for Engineering Databases, in: Query Processing in Database Systems, ed.: Kim, W., Reiner, D.S., Batory, D.S., Springer, Berlin Heidelberg New York Tokyo, 1984, S. 145-155.

Mi86 Mitschang, B.: MAD - Ein Datenmodell zur Verwaltung von komplexen Objekten (MAD: A Data Model for Complex Object Management), SFB 124 Research Report, No. 20/85, University of Kaiserslautern, revised in 1986.

Ne87 Nehmer, J., et al.: Key Concepts of the INCAS Multicomputer Project, in: IEEE Transactions on Software Engineering, Vol. SE-13, No. 8, 1987, pp. 913-923.

Pa84 Paul, H.-B., at al.: Überlegungen zur Architektur eines "Non-Standard"-Datenbankkernsystems (Considerations on the Architecture of a "Non-Standard" Database Kernel System), Research Report DVSI-1984-A2, Technical University Darmstadt, 1984.

SS86 Schek, H.-J., Scholl, M.H.: The Relational Model with Relation-Valued Attributes, in: Information Systems, Vol. 2, No. 2, 1986, pp. 340-355.

St81 Stonebraker, M.: Operating System Support for Database Management, in: CACM, Vol. 24, No. 7, 1981.

WNP87 Weikum, G., Neumann, B., Paul, H.-B.: Konzeption und Realisierung einer mengenorientierten Seitenschnittstelle zum effizienten Zugriff auf komplexe Objekte (Concept and Implementation of a Set-Oriented Page-Interface with Efficient Access to Complex Objects), in: Proceedings of the GI Conference on Database Systems for Office, Engineering, and Science Environments, Darmstadt, ed.: H.-J. Schek, G. Schlageter, Informatik Fachberichte No. 136, Springer, Berlin, Heidelberg New York Tokyo, 1987, pp. 212-230.