D. J. Hatfield

Experiments on Page Size, Program Access Patterns, and Virtual Memory Performance

Abstract: The assumption about virtual memory systems that as overhead (time for access and software page management) decreases page size should be reduced is not always a good one. Recent experiments indicate that larger page sizes can provide better performance for programs that make highly localized use of memory space.

Introduction

One parameter that affects the performance of memory hierarchies is the size of the block of data (e.g., the page) transferred between the memories. We are specifically interested in determining the best page size for a virtual memory system.

Key factors influencing the choice of page size are the time required to transfer a page between memories and the patterns of use of memories made by programs. There is a penalty involved in using large pages, since much of the data transferred may never be used. There is also a penalty for using small pages, which results from the time involved in separately transferring many small pages that may be used together during program execution. Consideration of these penalties leads to one working hypothesis about page size: as overhead (times for hardware access and software page management) decreases, page size should also be decreased; as overhead approaches zero, a smaller page is always better than a larger page[1-5]. Investigations of the access patterns of programs indicate that this hypothesis is not always good and that the problem of determining the best page size may not have a simple solution.

The work described in this paper is an extension of some recent experiments to investigate the effectiveness of automatic repacking of programs and program pieces into pages of virtual memory so as to reduce the page exceptions generated by instruction or data references across page boundaries[6]. In this paper, we first describe relevant portions of the procedures used in the earlier study. We then consider more thoroughly the usual basis for choosing page sizes and compare that with our experimental results.

Paging simulators

In order to compare the effectiveness of different placements of the same set of programs (or subroutines, control sections, common arrays, data areas, etc.), paging simulators were developed that accurately mimicked[7] the software and, whenever necessary, the hardware associated with actual page replacement algorithms. The paging simulators were accurate for a single user paging against himself in either a fixed amount of space or space that varied dynamically in response to his varying requirements. Simulator input was a sequence of page requests generated by processing a full instruction trace of a real program running under a real operating system. More precisely, since all the studies were performed on the IBM System/360 model 67, the sequence was one of page sets. For an instruction to be processed on the model 67, all addresses involved must be in physical memory at once. If the instruction and data are both in the same page, only one page is needed; but if the instruction crosses one page boundary and the data cross another, three or four pages are required. The instruction images from the trace were examined sequentially, and when the set of pages required for an instruction changed, the current set was recorded and replaced by the new set. Instructions that used a subset of the pages in the current set did not cause the set to be recorded and replaced by the subset, since for a single user no page exception could result from the requirements of the subset[8].

The sequence of page sets was processed by a page management simulator, which was given either static or dynamic constraints on the number of page frames available to the program during its execution. Since only a sequence of sets of virtual page numbers is supplied to the

simulator, it needs no information about page size and in fact is independent of page size.

The page size must be specified in the process of generating page-set sequences from the full instruction trace. Each instruction or data address from the trace is mapped into the corresponding address for the specific placement into pages of programs, subroutines, and data areas. This address is then divided by the page size to give the page number. Therefore, a page-set sequence can be generated for any desired page size.

Our initial reason for looking at different page sizes for different placements was to determine whether improvements in performance (fewer page exceptions for a fixed memory constraint) gained by packing for one page size would also apply to double- and half-size pages without further repacking. Since the packing algorithm uses the page size as well as the size of the program and data areas to be packed, it was not obvious that the overall placement of programs and data derived from packing for a particular page size would prove effective for other page sizes.

Because of the great likelihood that page sizes are some power of 2, the half-page and the double-page were first examined. Results were gratifying in terms of our original objectives, in that placements made on the assumption of 4096-byte pages proved to be good placements for 2048- and 8192-byte pages as well[9]. But more interesting was the comparison of page exceptions for page sizes of $n/2$ and n bytes, where n in the cases examined took on the values of 2048, 4096, 8192, and 16,384.

Expected results

Before examining the actual results, here is a summary of the results expected based both on our intuition and on the literature[1–5]. If a program is divided into pages of size n and then into pages of size $n/2$, execution of the program with the smaller size pages would seem to imply less data transfer resulting from page exceptions. The reason for this is that we would not expect the program to always use both halves of the larger size pages. If only half of the larger size pages were always used, the length of the sequence of pages for both page sizes would be the same. Hence a request for large page i would correspond to a request for either small page $2i$ or small page $2i-1$. If the correspondence from large to small pages always involved the same half of the large page i, the request sequence would be exactly the same except for renumbering, and the same number of exceptions would result for the same number of page frames. But the same number of page frames implies half the space, so that for the same amount of space the number of exceptions should be appreciably less. And the time for data transfer would also be less, since each page is half as long. This situation clearly favors the smaller page size.

At the other extreme (i.e., both halves of a larger page are always used) every time large page i is requested, the corresponding sequence of small pages needed would be $2i-1$, $2i$. Then the overall sequence of small pages would be exactly twice as long as the sequence of large pages; for the same amount of real memory space, there would be exactly twice the number of small page exceptions as large page exceptions. But again, since the small pages are only half as long, the total number of bits transferred would be the same. Any extra overhead for small pages would be due to access time plus software time associated with updating and searching page tables.

In addition, we had assumed that an effective upper bound on small page exceptions is twice the number of large page exceptions, and that, as the density of use within large pages decreases, the ratio of small page exceptions to large page exceptions decreases also. Of course one would expect the advantage of the small page size to increase as the amount of space available decreased.

Experimental results

Some measurements on a program that we characterized as having low-density memory use (program A in Fig. 1) generally confirmed these expectations. The vertical axis in the figure represents real memory space and the vertical lines represent memory usage. The horizontal axis represents execution time. Figure 2 shows the curves of page exceptions versus available space for 2K-byte pages (solid line) and 4K-byte pages (dotted line). The page replacement algorithm was first-in, first-out (FIFO). The dashed line gives the ratio of exceptions for 2K-byte pages to exceptions for 4K-byte pages. The horizontal line at 1000 exceptions indicates a ratio of one-to-one, and the line at 2000 exceptions indicates a ratio of two-to-one. Over most real memory sizes, the 2K-byte pages give fewer exceptions than the 4K-byte pages. Only when the program has all the space it needs does the ratio approach two-to-one. The height at which the dashed line stops indicates the ratio of the total number of 2K-byte pages needed to the total number of 4K-byte pages needed.

When this program was repackaged to increase the localization of memory use by placing together in memory space program parts used close together in time (see Fig. 3), the number of exceptions for both 2K- and 4K-byte pages was fewer than in the previous case. The increased localization was indicated by: fewer page references (for both 2K and 4K) required for completion of the program; a smaller working set size, measured over 2500-, 5000-, 7500-, and 10,000-instruction intervals; and an increase in the ratio of program instruction and data transfers within pages to those transfers from one page to another. However, the ratio was much less favor-

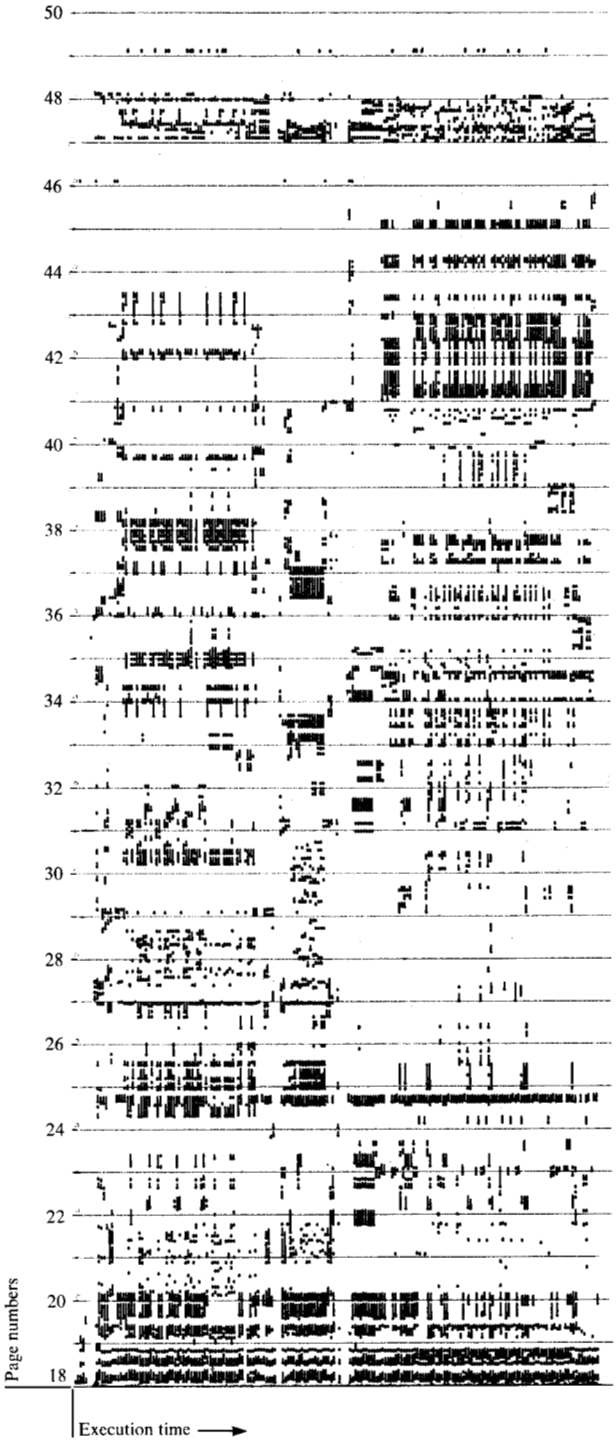

Figure 1 Memory usage by program A.

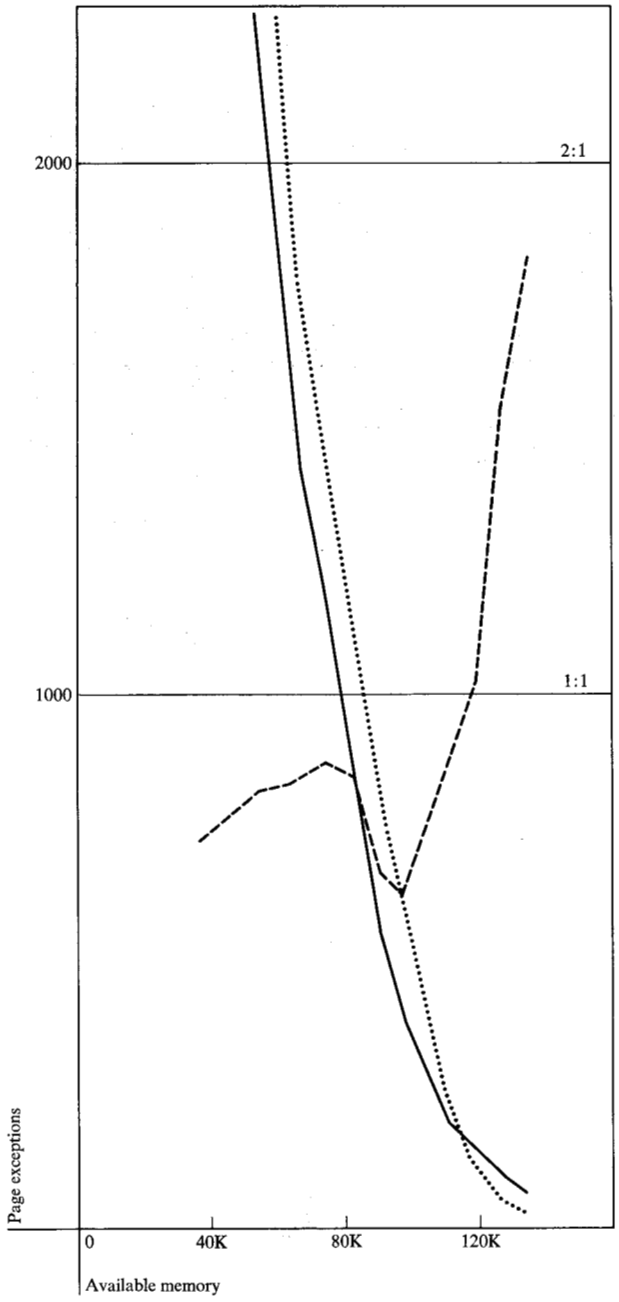

Figure 2 Page exceptions for program A.

able to the smaller page size, especially in the right half of the curve, which is the region of moderate to low paging activity and therefore clearly the most desirable region (Fig. 4). In fact, throughout most of this region, the exception ratio was above what we refer to as the break-even level for the hardware-software environment within which the experiments were run.

The break-even level is determined as follows: the time to process a page exception is composed of the time for data transfer, the access time for the device where the page resides, and the software time involved in handling the page exception interruption and finding a page to replace. Under the assumptions that the device speed and the overall paging load are the same for both page sizes, the access time α will be the same for both the large

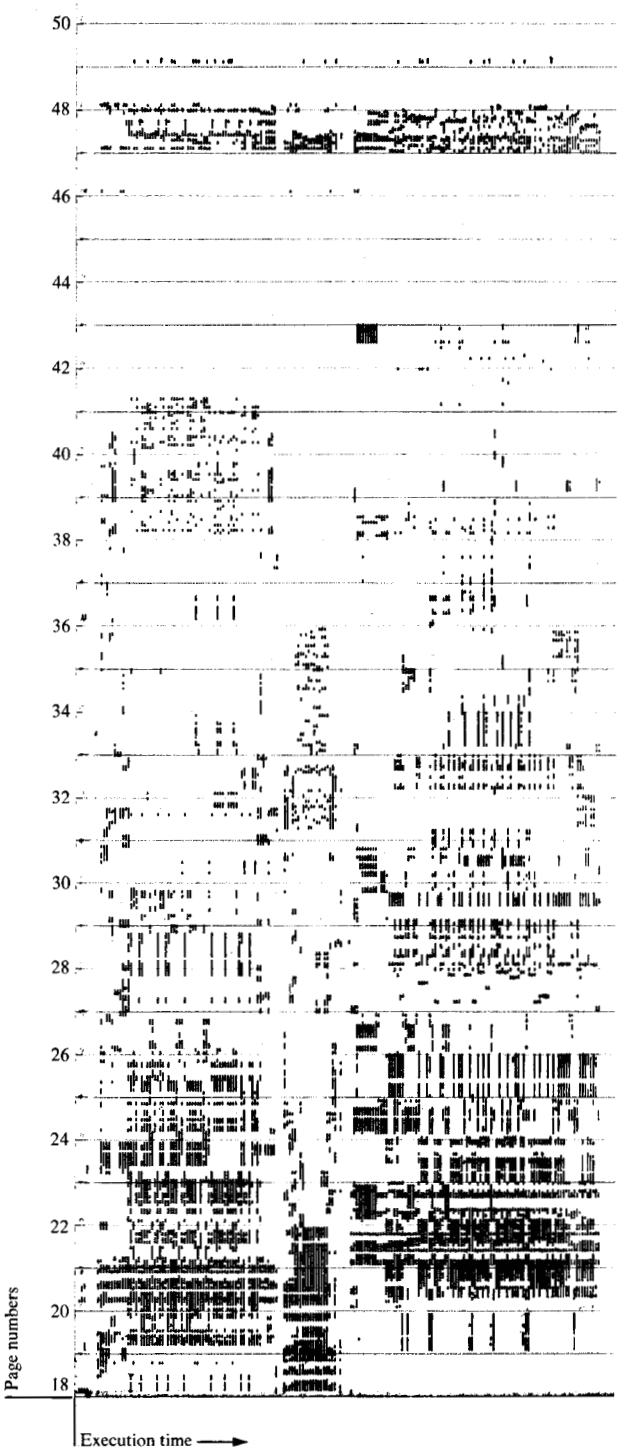

Figure 3 Memory usage by program A after repackaging.

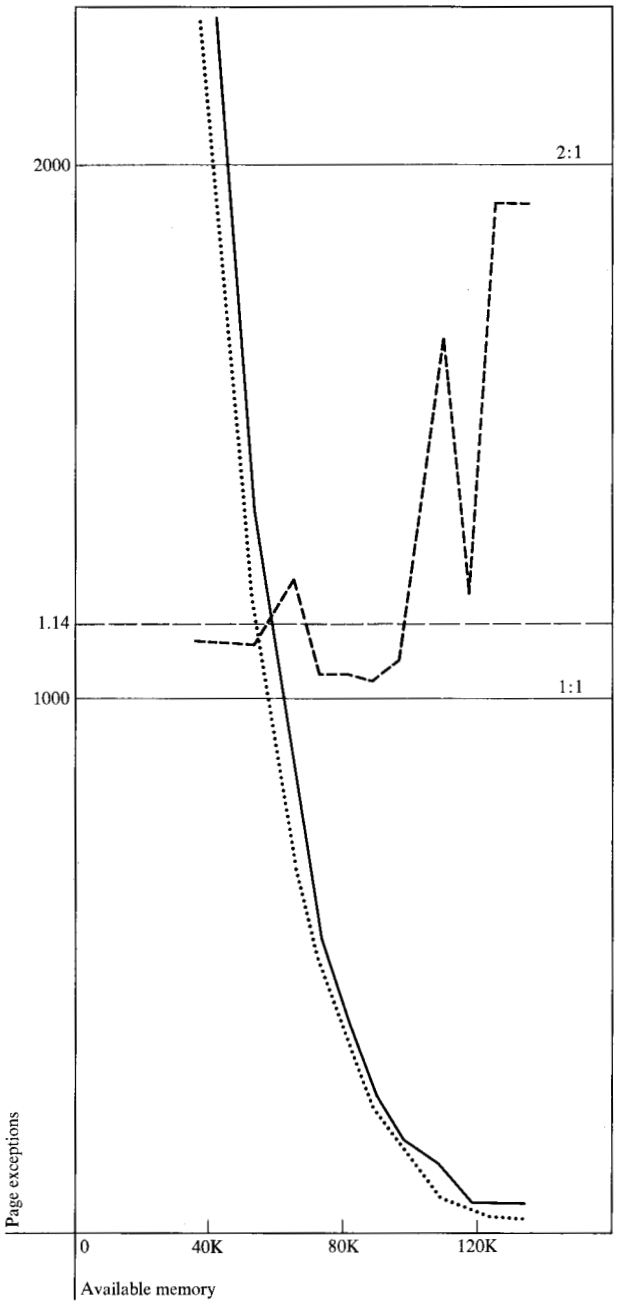

Figure 4 Page exceptions for program A after repackaging.

and the small page size. The transfer time will be β for the large page size and $\beta/2$ for the small. The software overhead to replace a page is difficult to determine as a function of page size. There are potentially more small pages to look through, but the condition for which the search is made may be distributed in the same manner through small and large pages, so that the depth of search would be the same. Since it is easy to stipulate conditions that would favor either size, the software overhead δ was assumed to be the same for both cases.

The number of small page exceptions that can be processed in the time it takes to process one large page exception is

$$\frac{\alpha + \beta + \delta}{\alpha + \frac{1}{2}\beta + \delta}$$

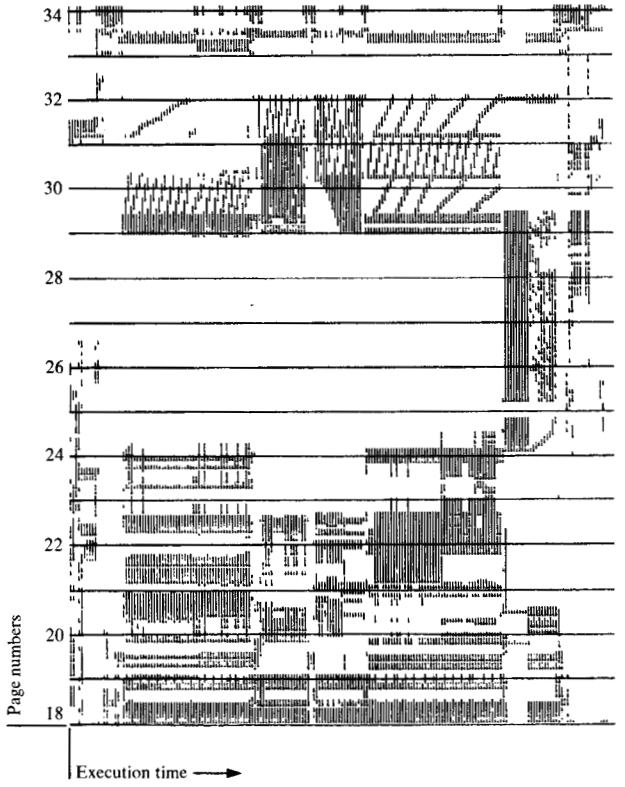

Figure 5 Memory usage by program B.

For CP-67 version 3 on the model 67 with an IBM 2301 paging drum, this number is 1.14, largely because α is large compared to β and δ.

For devices with access time reduced with respect to transfer time, this number is closer to 2.0. On the other hand, if the I/O operation time is totally overlapped with program execution, the ratio reduces to $\delta_n/\delta_{n/2}$. The value of this ratio is dependent on the program, the page size, and the page replacement algorithm, and will not be discussed further in this paper.

Given a break-even level of 1.14, it is clear that the repacked program favored the larger page size throughout most of the desirable performance region. For programs with greater localization of heavily used memory, the bias to the large page size is also greater (Figs. 5 and 7). Both of these programs have a more stable working set and show a sharper bend in the page exception curve than do either the original or the repackaged version of the first program. And in the page exception graphs (Figs. 6 and 8) the smaller page size often resulted in more than twice the exceptions, contrary to our intuitive expectations. After checking our page replacement simulators and finding no logical errors, we tried to find models that would predict ratios in excess of two:one[5].

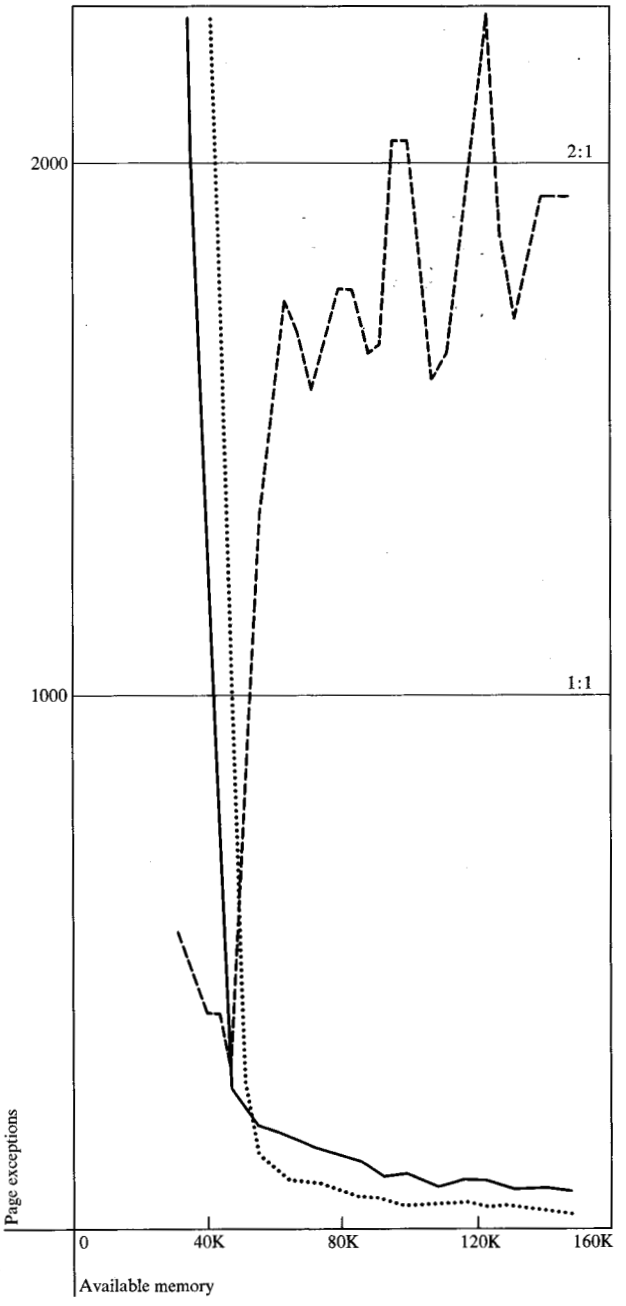

Figure 6 Page exceptions for program B.

It was not difficult to find examples of page request sequences giving two, three, and four times the number of exceptions for the smaller page size. The examples in Fig. 9 are simple sequences that parallel the activity of real programs in an environment involving more real pages and longer strings of requests between page exceptions. The vertical boxes at the left represent an initial stack of pages ordered for removal when a page exception occurs. The bottommost member of the stack is to be

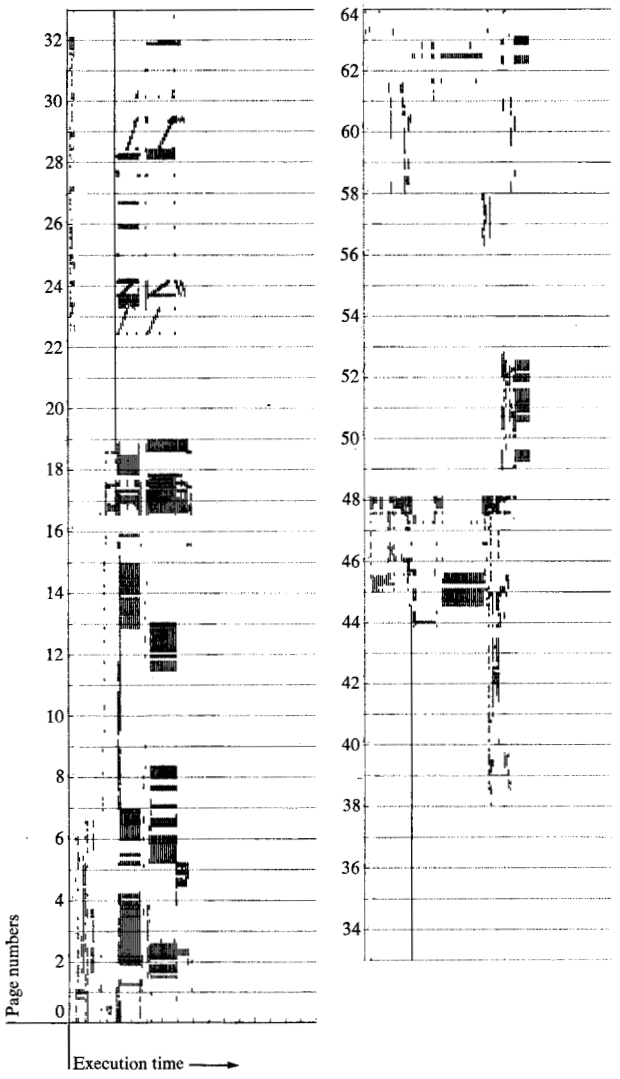

Figure 7 Memory usage by program C.

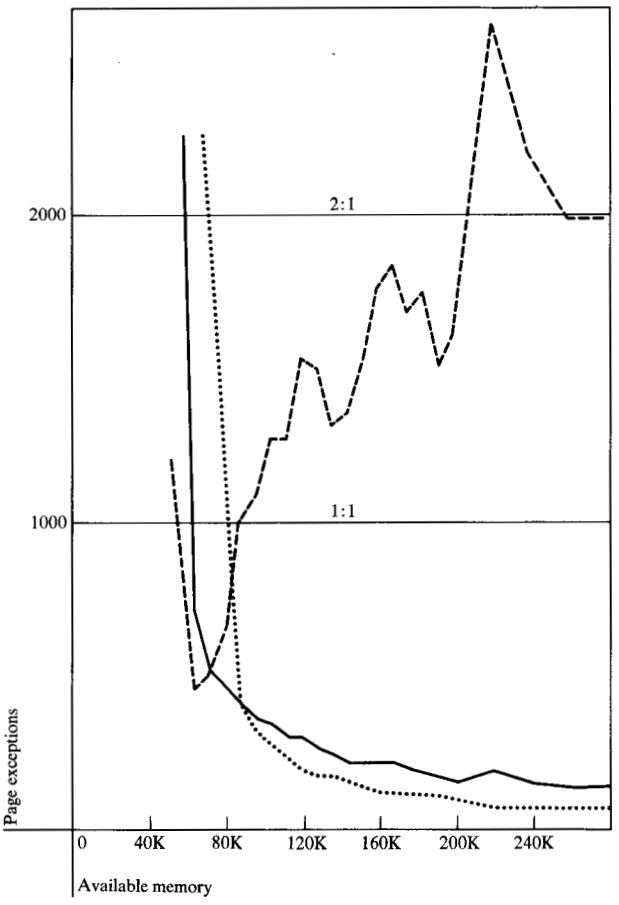

Figure 8 Page exceptions for program C.

removed first. Corresponding to two requests for large page i are two requests for either one or both halves of the page as some combination of the numbers $2i$ and $2i - 1$. Page requests that cause page exceptions are underlined in both cases. In the first and second and the fourth and fifth examples, the request sequence and exception pattern can cycle indefinitely. This is because the removal stack at an earlier position in the sequence is reestablished later in the sequence, defining a cycle that will regenerate itself as many times as desired. In these cases, the number of exceptions given is that for one cycle (all examples are for single page sequences, but can be broken into sets without destroying the phenomenon).

It should be emphasized that these sequences are the result of the accessing patterns of programs and not an artifact resulting from an instruction or a word of data spanning a half-page boundary. The occurrence of an instruction or data reference spanning a half-page boundary was less than one percent in the real programs examined. In other words, if we consider a string of program instructions translated into page references (one or more page references per instruction) first for pages of size n and then for pages of size $n/2$, the page reference strings for n and $n/2$ are essentially the same length. Note that such a reference string is not the same length as the compressed string of page sets given to the page management simulator. For the single-user case, however, both the full and the compressed string contain the same information.

These examples all have some things in common. Most of the time, both halves of the large page are used. Page exceptions for the large page are far enough apart so that between the corresponding small page exceptions there are more than enough changes of state to significantly reorder the stack. On the other hand, between the times of the large page exceptions, the stack is not significantly reordered. The large page that causes a page exception usually corresponds to two small page requests, and both usually cause a page exception. These conditions would

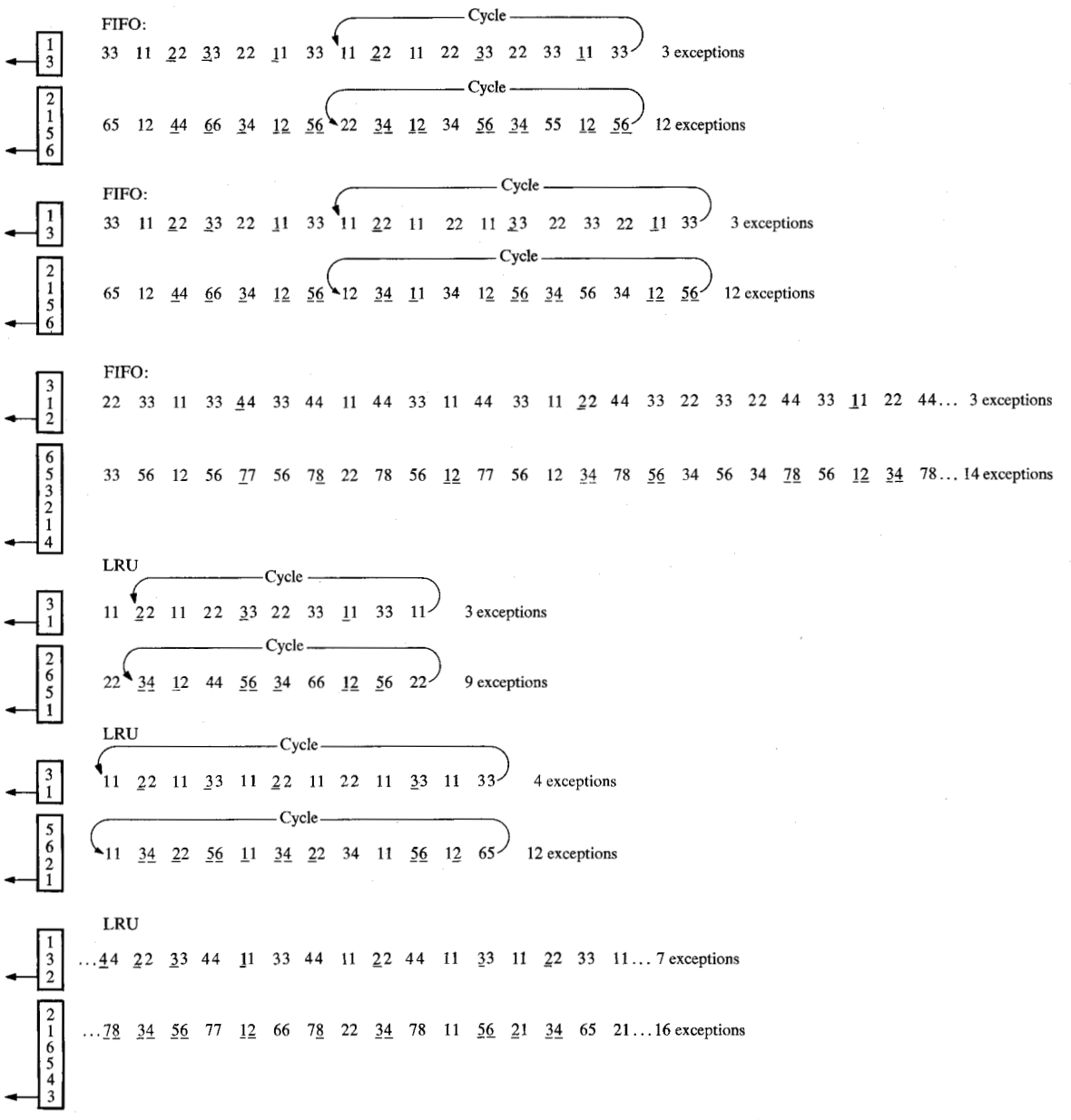

Figure 9 Examples of page request sequences.

be expected from programs characterized as high-density users of memory, and for those programs only when the paging rate for the large page size is relatively low (i.e., the righthand side of the graphs). In addition, the stipulation that the removal stack for the large page size not be reordered by the time a page exception occurs would imply that the phenomenon is more likely to occur with FIFO than with least recently used (LRU) replacement. This follows because the FIFO stack can change only at page exception time, while the LRU stack can change with every instruction. This has been observed experimentally, with the LRU algorithm seldom giving an exception ratio greater than 2:1 for real programs. Figure 10 shows LRU replacement applied to the program shown in Fig. 7 for FIFO. As yet we have been unable to prove that there is a replacement algorithm using only the past history of page requests that cannot generate more than twice the exceptions with half size pages. Figure 11 shows the program displayed in Fig. 5 passed by a replacement algorithm that selects a page for removal by examining a single "used" bit for the page. It can easily be shown that the MIN algorithm, which gives the minimum number of page exceptions for any request string, cannot produce more than twice the exceptions for the half page size[11]. But for other algorithms, especially those in use today, there seem to be no guarantees.

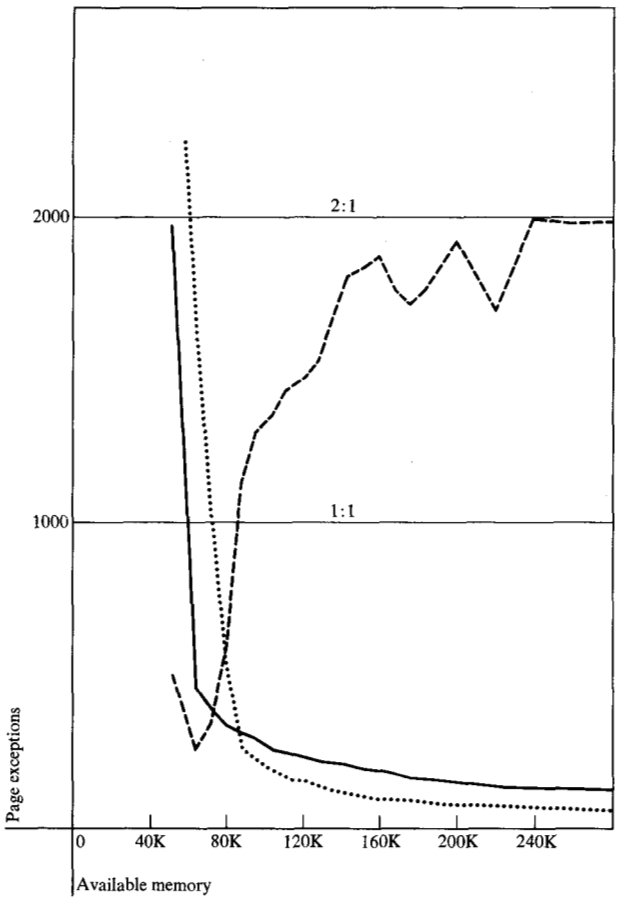

Figure 10 LRU replacement for program C.

One other not obvious characteristic of the page ratio curves should be noted. Instead of progressively favoring the smaller page size as available space is reduced, the curves (for as far as we have measured them) reach a minimum near the middle of the available space range, with the minimum growing sharper and shifting to the left as usage density increases. Several modes of page request behavior could account for this, but we have not yet been able to specify analytically the degree to which real programs resemble these models.

For instance, the amount of memory space involved in a memory cycle can be the determining factor. If a program is cycling through r pages and has only $r - 1$ page frames available, both LRU and FIFO replacement algorithms will generate a page exception for every page request. If the same program is run on pages half as large, the cycle may involve far fewer than $2r$ small pages. Any cycle using $2(r - 1)$ or fewer small pages will generate no page exceptions, compared with one exception per page request with large pages. But if the available space is further constricted so that the cycle does not fit for either large or small pages, both cases will generate one

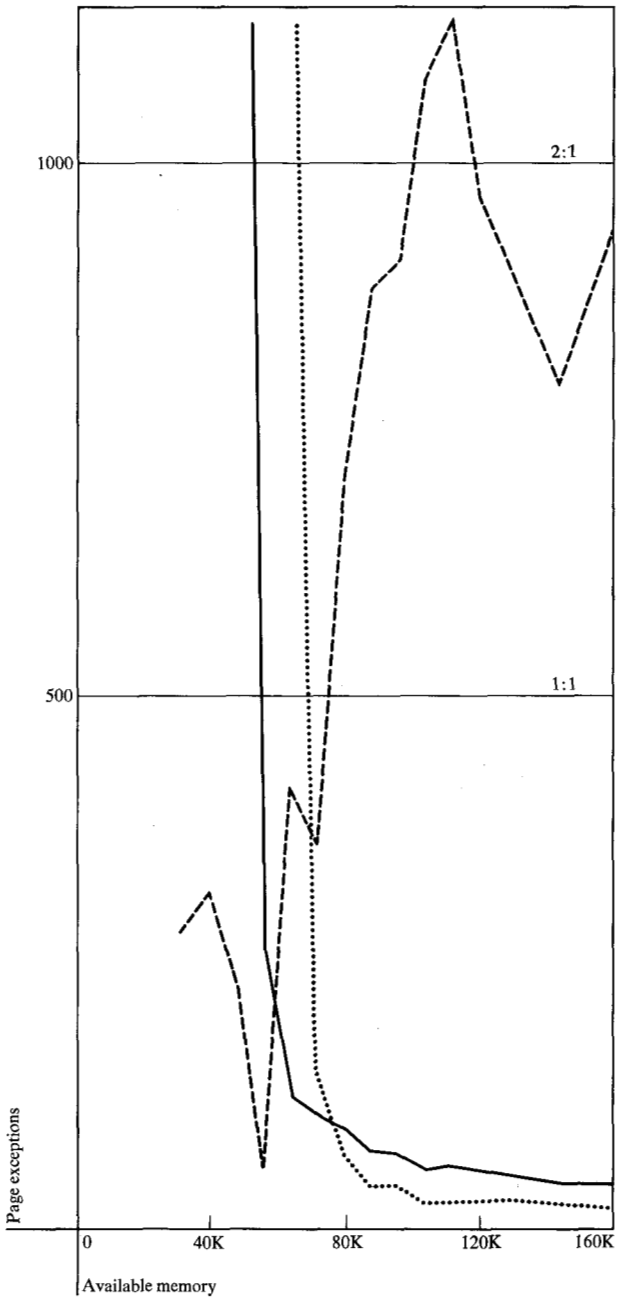

Figure 11 "Used bit" replacement for program B.

exception per request, and the large size page will become competitive again with respect to the small. The curve representing the ratio of small to large page exceptions will climb as the available space is reduced.

The effect of more than doubling the exceptions for the half-size page has been noted for a large size page of 16,384, 8192, 4096 and 2048 bytes. It is a characteristic of programs that make highly localized use of memory and that therefore perform well on systems using relocation hardware for address translation and is also a char-

acteristic of those programs in the region of low paging activity. It seems to affect all implementable replacement algorithms, especially those that seldom permanently alter the removal stack between exceptions. Because it is related to the page request string and not the page size, it can apply to small pages (of 64 and 32 bytes) as well as large ones, and so can be encountered in machines with caches as well as those using relocation hardware.

The question of page size (usually termed slot size) for a cache is usually more complex than that of page size for main memory. Typically the replacement algorithms used for caches are only locally LRU. The ratio of retrieval time for slots of n and $n/2$ bytes depends on more than the addressing and data transfer time from memory. In addition, the degree of interleaving of memory modules determines how many bytes can be sent to the cache at once. For the System/360 model 85, a 16-byte draw and a 4-way interleaving implies that 64 bytes can be brought into the cache just as quickly as 32 bytes. For each module of an interleaved memory, the memory design can determine whether successive accesses can be attempted as soon as the addressing and transfer hardware are ready, or whether a memory cycle must elapse between one access and another. Individual system designs are complex enough so that the ratio of retrieval times for n and $n/2$ bytes can range anywhere from one to a little more than two.

It is not really surprising that more localized use of memory would favor larger pages, since if a program stayed uniformly within one large block, the optimal page size would be the size of the block. The possible degree of mismatch between page size and memory use pattern, however, implies that careful study is required in order to decide the best performing page size for a program or a programming system, and that the assumption, "the smaller the page size the fewer wasted I/O transfers," is not always correct. The relation between page reference patterns and page replacement algorithms gives rise to behavior that is not yet well understood and may require stricter definitions of program locality.

References and notes

1. M. Joseph, "Analysis of Paging and Program Behavior," *Computer Journal* **13,** No. 1, 49 (February 1970).
2. L. Belady, "A study of replacement algorithms for a virtual-storage computer," *IBM Systems Journal* **5,** No. 2, 93–94 (1966).
3. B. Randell, "A note on storage fragmentation and program segmentation," *Communications of the ACM* **12,** No. 7, 365–366 (July 1969).
4. M. H. J. Baylis, D. G. Fletcher, and D. J. Howarth, "Paging Studies Made on the ICT Atlas Computer," *Proceedings IFIP Congress 1968* **2,** 835–836 (1968).
5. P. J. Denning, "Virtual Memory," *Computing Surveys* **2,** No. 3, 169 (September 1970).
6. D. Hatfield and J. Gerald, "Program Restructuring for Virtual Memory" *IBM Systems Journal* **10,** 168 (1971).
7. The limit of this accuracy is the set of all events triggered by changes in virtual memory or execution cycle requirements. Timing considerations introduced by the speed of different memory or data transfer devices were ignored, and to this extent the simulation was not complete.
8. The average size of a page set for the programs we have examined varies between two and three pages. Naturally the difference in performance between a sequence of single pages and a sequence of sets becomes more critical as the available paging space is reduced.
9. The improvements in the double page case were greater than with the full page for which the repackaging was performed. The repackaging programs strung together pages that communicated with one another and so created very well packed double pages. Splitting the full pages in half destroyed some of the effect of page packing and the improvements for the half page case were not as great as with the full.
10. I am grateful for discussion of the problem and for some examples of sequences of page requests that give more than twice the exceptions at the half page size to John Pomerantz, Department of Computer Science, University of Chicago.
11. An algorithm can be devised that gives exactly twice the number of exceptions for the half page sequence by looking ahead in the request string, as does the MIN algorithm. The MIN algorithm itself must do at least this well, so cannot generate more than twice the page exceptions with half pages. For a discussion of the extention of the MIN algorithm to handle page sets, see L. A. Belady, "Use of the minimum page replacement algorithm to produce specified memory states," IBM T. J. Watson Research Center Report.

Received August 26, 1971

The author is located at the IBM DPD Scientific Center, Cambridge, Massachusetts 02139.