ELSEVIER

The Journal of Systems and Software 50 (2000) 151–170

The Journal of Systems and Software

www.elsevier.com/locate/jss

The effect of compression on performance in a demand paging operating system

Allen Wynn, Jie Wu *

Department of Computer Science and Engineering, Florida Atlantic University, Boca Raton, FL 33431, USA

Received 21 October 1997; accepted 13 April 1998

Abstract

As engineers increase microprocessor speed, many traditionally computer-bound tasks are being transformed to input/output (I/O) bound tasks. Where processor performance had once been the primary bottleneck, I/O performance is now the primary inhibitor to faster execution. As the performance gap widens between processor and I/O, it is becoming more important to improve the efficiency of I/O in order to improve overall system performance. In a demand paging operating system, secondary memory is accessed during program execution when a page fault occurs. To improve program execution, it is increasingly important to decrease the amount of time required to process a page fault. This paper describes a compression format which is suitable for both pages of code and pages of data. We find that when the OS/2 operating system is modified to use this compression format, the time saved due to reduced I/O offsets the additional time required to decompress the page.

Keywords: Demand paging; Compression; Operating system; Page-replacement policy

1. Introduction

1.1. Motivation

As microprocessors increase in speed many traditional compute-constrained tasks are transformed into Input/Output (I/O) bound task. Access to secondary memory or direct access storage devices (DASD) is quickly becoming the primary bottleneck for many computer Systems. Engineers continue to make great strides in processor speed, the processing power of the Intel X86 family of processors has increased 335 times between 1978 and 1993 (Wirth, 1995). Such amazing increase in performance has not been seen in DASD, however. As the performance gap widens, system performance will tend to become bounded by I/O performance. This is similar to Amdahl's Law: The small DASD-access component of a problem will limit the speed-up attainable by increasing processor speed. Thus, it is becoming much more important to understand the dynamics of DASD performance if we are to improve overall system performance.

1.2. Problem description

Many present-day operating systems use demand paging, which means that a page of code or data is not read into memory until the page is needed. In an overcommitted situation, where the number of pages actively in use exceeds the number of pages of physical memory, it becomes necessary to select pages that will be removed from physical memory in order to make room for new pages.

One key concept to remember is that DASD I/O is the major contributor to the performance difference between two identical systems with different amounts of memory. The point being, in a memory constrained environment a system

* Corresponding author. Tel.: +1-561-297-3491; fax: +1-561-297-2800.
E-mail addresses: awynn@us.ibm.com (A. Wynn); jie@cse.fau.edu (J. Wu).

0164-1212/00/$ - see front matter
PII: S0164-1212(99)00090-4

with less memory must age and swap out or discard pages (remove pages for memory) and swap in pages (bring pages into memory) to account for the missing memory. Although the statement above is not strictly correct (there is some CPU activity associated with the paging and additional scheduling activity) the majority of the difference in performance can be directly attributed to DASD I/O.

Demand paging operating systems access DASD on an as-needed basis. Because DASD access is a performance bottleneck, a key to improving system responsiveness is improving DASD performance. Compression is a popular technique for increasing storage capacity. Due to the wide differential between the CPU speed and DASD speed, compression can sometimes be used to gain a performance improvement.

If the time required to read a compressed page from DASD and decompress it is less than the time required to read an uncompressed page from DASD, then page-based compression may improve the performance of a demand paging operating system. However, more memory is required for compressed page because an additional buffer is needed. In a memory constrained environment, removing a minimal amount of memory could cause performance degradation.

Implementation and analysis of page-based compression in the OS/2 operating system has yielded two results with respect to memory. First, an operating system with page-based compression uses less virtual memory than an operating system without page-based compression. In an unconstrained environment, the system with page-based compression will generate fewer page faults than the system without page-based compression, which translates into faster performance.

Second, a system with page-based compression requires more locked (wired) memory than a system without page-based compression, which effectively reduces the amount of memory available to the rest of the system. This will produce no noticeable effect in an unconstrained environment. In a memory constrained environment, however, even a small reduction in the amount of memory will cause an increase in the number of page faults.

We have found that the time saved due to compression offsets the additional time caused by the increase in number of page faults, even in a memory constrained environment. Hence, page-based compression improves the performance of a demand paging operating system, irrelative to the amount of memory in the computer.

Section 2 gives an overview of compression and describes lossless and lossy algorithms. Dictionary-based compression algorithms, specifically LZ77 and LZ78 and variants, are explored in more detail. Section 2 also describes the OS/2 operating system, focusing primarily on application loading and page-replacement policy, including aging and the page fault process.

Section 3 describes the approach used in this study. The Exepack2 compression format, which is based off LZ77 and variants, is described in detail. Utility and operating system modifications are listed, as well as the hardware setup, software setup and test sequence.

Section 4 lists the results of this study. The effect of page-based compression on the page fault rate is analyzed, as well as the effect of page-based compression on performance. Scalability analysis is also furnished.

Section 5 includes a summary and the contributions of this study. Suggestions for future research are provided.

2. Preliminaries

2.1. Compression algorithms

Compression techniques can be divided into two major categories, lossy and lossless. Lossy compression includes techniques which allow a tolerable loss of accuracy in return for a greater compression ratio. Lossy compression is well suited for data which are digital representations of analog phenomena, such as speech, audio, images and video signals. Such digital data do not perfectly match the measured events due to the analog-to-digital conversion, so additional loss of fidelity may be tolerable. Lossy compression techniques can usually be tuned to trade-off accuracy for increased compression.

Lossless compression, on the other hand, allows perfect reconstruction of the original data. This type of compression is appropriate for database records, financial information, text files and executable modules. Lossless compression is also known as noiseless compression because no noise, or loss of quality, results (Gersho and Gray, 1992).

We are exclusively interested in lossless compression because our research deals with compression of code and data within an executable module. Loss of code and or data could have disastrous results.

The compression methods can also be divided into three categories. The first category recognizes and removes sequential repeating characters, while the second category uses statistics in an attempt to encode characters into bit

strings which are smaller than the characters. Compression methods in the first category work well on some types of data, but do not work on other types of data or on executable code. Compression methods in the second category work well if the range of characters is not evenly distributed.

The third category of compression methods is dictionary-based compression. Dictionary-based compression algorithms do not operate on a character-by-character basis, like the previous compression methods. Dictionary-based compression algorithms encode variable length strings as single tokens, which are indexes into a phrase dictionary (Nelson, 1992).

Dictionary-based compression algorithms can have either a static dictionary or an adaptive dictionary. Most static dictionary-based compression schemes are built for a specific application, and are not general purpose. An example would be an inventory system with dictionary phrases like 'desk', 'chair', 'table' and 'vacuum cleaner'. The dictionary would be created before compression occurs. Static dictionary-based schemes can tune their dictionary in advance. More frequently used phrases like 'chair' would be assigned fewer bits, while less frequently used phrases like 'vacuum cleaner' would be assigned more bits. However, the dictionary must be available to both the encoder and the decoder. In some instances, this may mean transmitting or storing the dictionary along with the compressed data. If the data block is sufficiently large, then the overhead of storing the dictionary may be tolerable.

Adaptive dictionary-based compression algorithms start with either no dictionary or with a general default dictionary. As the compression algorithm sees the input data, it adds new phrases to the dictionary.

Most adaptive dictionary-based compression algorithms are based on two papers presented in 1977 and 1978 by Ziv and Lempel (1977, 1978). Ziv and Lempel (1977) paper describes a sliding-window technique (LZ77) in which the dictionary consists of the phrases found in a window, or section, of the previously seen data. The window 'slides' across the previously seen input data so that the most recently compressed data are always contained within the window. LZ77 also manages a look-ahead buffer, which can be viewed as a window into the input data containing the next characters to be encoded. As data are compressed, the uncompressed characters slide out of the look-ahead buffer and into the dictionary.

Tokens in the LZ77 compression format consist of three parts, an offset back into the dictionary, a phrase length, and the first character in the look-ahead buffer after the phrase. It is clear to see that pointers and phrases must be alternated.

Consider the following example:

Synchronization and resource alloc	ation alternatives in parallel

In this example, the look-ahead buffer contains the string 'ation alternatives in paralle'. As the algorithm searches the dictionary, it finds that 'ation a' from the look-ahead buffer matches the phrase at position 24 in the dictionary. The token would be encoded as: 24,7,'l', indicating the position in the dictionary, the length of the phrase, and the next character. After writing the token to the output buffer, or transmitting the token, the windows would be shifted by 8 characters.

ion and resource allocation al	ternatives in parallel and distr

Suppose the phrase in the look-ahead buffer does not have a matching phrase in the dictionary, as indicated below:

resource allocation alternatives in	parallel and distributed syst

Because there is no matching phrase in the dictionary, the token would be encoded with a zero offset and zero length, such as O,O,'p'. This method of encoding single characters is inefficient because the token requires more space than the original character. In fact, if the offset/length pair require 16-bits and a character requires 8 bits, the token requires three times as many bits as the input character. Later variations of LZ77 attempt to reduce or remove this inefficiency.

Decoding with LZ77 is quite simple. For each token, the decoder uses the offset field to index into the window of previously decompressed text in the output buffer. It then copies the number of bytes specified by the length field from that location to the current location in the output buffer. Finally, the character is copied from the token to the output buffer.

The two major tunable parameters for LZ77 compression are the size of the dictionary and the size of the look-ahead buffer (the maximum phrase length). The average time to find the longest matching phrase is of the order of the product of the dictionary size and the phrase length. Thus, doubling the size of the dictionary or the look-ahead buffer has a similar doubling effect on the compression time. Decompression time is not significantly affected by increasing either the dictionary or the maximum phrase length.

The LZSS compression algorithm was a well-received extension of LZ77. LZSS has two primary enhancements to LZ77. First, LZSS adds a tree structure on top of the dictionary. If the tree is implemented as a binary tree, then the average time to find the longest matching string will be of the order of the base 2 logarithm of the dictionary size times the phrase length, which is far less than required by LZ77. Also, larger dictionary sizes can be used because doubling the size of the dictionary or the look-ahead buffer will cause a small impact to the compression time rather than doubling it.

Tree structures can have a condition known as being 'unbalanced'. This means that the structure deteriorates so that most of the links resolve to one direction, perhaps to the left side, while the links in the other direction are unused. In the worst case, the tree structure, and hence the performance, degrades to that of a singly linked list. While there are may techniques to rebalance binary trees, it may not be necessary in the case of LZSS dictionaries. Because of the nature of the sliding window, phrases are constantly being added and removed from the window. Therefore, nodes will be constantly added and removed from the tree. In a dynamic environment such as this, an unbalanced tree will quickly tend towards a more balanced state.

Another major performance enhancement in LZSS has to do with the compression format. Recall that LZ77 forced compressed phrases to alternate with uncompressed characters. LZSS, on the other hand, allows compressed phrases and uncompressed characters to be intermixed. Each token contains a single bit prefix which indicates whether the token is an offset/length pair or an uncompressed character. The overhead for an uncompressed character is reduced from several bytes per character to a single bit. This performance enhancement is readily apparent when the compression algorithm is just starting to compress and has an empty dictionary.

In 1989 Stac Electronics implemented an LZ77-based compression algorithm on a single chip. The fact that it was a hardware implementation meant that it could compress faster than a software implementation and that it could be used as a generally available compression method by many different hardware manufacturers. The Quarter Inch Cartridge industry group adopted this compression method as a standard and named it QIC-122. QIC-122 overcame the problem LZ77 restriction of alternating pointers and uncompressed characters in a manner identical to LZSS. Each QIC-122 token has a single bit prefix which indicates whether the token is an 8-bit uncompressed character or a offset/length pair. QIC-l22 also implemented two types of offset/length pairs. Offsets that are less than 128 bytes use 7 bits, while offsets between 128 and 2047 bytes use 11 bits. LZ77 and its variants take advantage of spatial locality of reference due to the nature of the sliding window. In 1978, Ziv and Lempel published a paper (Ziv and Lempel, 1978) describing a compression algorithm which achieves better compression with more evenly distributed data. LZ77 tokens consist of an offset, a length, and a character that follows the phrase. LZ78 tokens consist of a code which specifies a phrase in the dictionary and a character that follows the phrase. Each phrase in the dictionary has a length associated with it, so it is not necessary to store the phrase length as part of the token. Dictionaries are typically stored in multiway trees with the null string as the root (Nelson, 1992).

When the LZ78 encoder starts, the dictionary will contain one entry, a null string. The LZ78 encoder builds the dictionary as it compresses the data. As the encoder processes the data, it searches for a matching phrase in the dictionary. When a match is found, the encoder outputs a token consisting of the code for the phrase plus the character that follows the phrase. This new phrase, which is one character longer than the matching phrase, is then added to the dictionary (Nelson, 1992). The compressed data, then, can be viewed as a list of tokens indicating the phrases added to the dictionary in the order that they were added.

The initial dictionary for the LZ78 decoder also contains a single entry, a null string. Each token that the decoder processes can be decompressed based on the information in the dictionary and the new character. The token also indicates a new phrase which was added to the dictionary by the encoder and, thus, must be added to the dictionary by the decoder.

There are several choices for dealing with a full dictionary. First, the encoder can stop adding entries to the dictionary. This approach is not appropriate when compressing large blocks of data if the statistical profile of the data may change significantly. An example would be an executable module where the code portions tend to have a

different profile from the data portions. A second choice would be to discard the entire dictionary and start with a new dictionary. The UNIX COMPRESS program uses a combination of these two approaches. If COMPRESS detects a deterioration in the compression ratio, it will discard the dictionary and start with a new dictionary (Nelson, 1992). This hybrid approach has the advantage of not deleting a full dictionary that continues to compress well.

The tunable parameter for LZ78 compression is the number of entries in the dictionary, which is the same as the number of bits in the code portion of the token. If 16 bits are used, there can be 65 536 phrases in the dictionary. If 24 bits are used, the dictionary can contain 16 777 216 phrases. The price for increased dictionary size is the execution time required to manage the larger dictionary tree.

A major disadvantage of LZ77 is also found in LZ78. Both compression algorithms force phrases to alternate with uncompressed characters. The LZ78 decoder, unlike the LZ77 decoder, must maintain the dictionary tree just like the LZ78 encoder. The LZ77 decoder simply looks at an index and a length. The LZ78 decoder must translate the code into a dictionary entry so the dictionary maintained by the decoder must exactly match the dictionary maintained by the encoder.

Terry Welch published a paper (Welsh and Terry, 1984) describing a compression algorithm which improved LZ78 in the same way that LZSS improves LZ77. LZW removes the restriction that phrases alternate with unmatched characters, and LZW never outputs unmatched characters.

The LZW compression format consists of a single field, the code which specifies the dictionary entry. The initial dictionary contains entries for all possible single-character phrases, so any character can be represented, even if it has not appeared in the input data. As the LZW encoder processes the data, it searches for a matching phrase in the dictionary. When a match is found, the encoder outputs a token consisting of the code for the phrase. A new phrase, which consists of the matching phrase from the input data plus the next character in the input data, is then added to the dictionary (Nelson, 1992).

2.2. OS/2 operating system

OS/2 is a demand paging operating system. When an executable or dynamically linked library (DLL) is 'loaded', the operating system does not read the entire file into memory. The loader reads the header information from the file and simply reserves the necessary virtual address space and marks all pages as not present. No code or data are read from the file at this point. The loader also resolves all imports by loading each DLL which is referenced (Deitel and Kogan, 1992; Huynh and Brew, 1993a).

Once all references have been resolved, the operating system starts to execute the initialization routines for any DLL which requires initialization. Once DLL initialization routines have been completed, the operating system executes the starting address of the application. Because OS/2 uses demand paging, the code might not be present, in which case a page fault will occur. The pager determines where to get the page, either from the executable image on the disk, from the swap file, compressed buffer, reclaim from the idle list, or Allocate on Demand (zero fill pages), and ensures that the page is made present (Huynh and Brew, 1993).

OS/2 uses virtual addressing, so the pages do not have to be present in memory unless they are actually being used. In an overcommitted situation, where the amount of virtual memory currently in use is greater than the amount of physical memory in the computer, OS/2 will swap or discard pages as appropriate. Pages of data that contain only zeros can be easily recreated, and are marked Allocate on Demand. Pages of data that are 'dirty' or have been modified, will be written to the compression buffer or to the swap file. Pages of data that are 'clean' or unmodified, and pages of code will generally be discarded. These clean pages can be reloaded from the application file on disk if they are needed at a later time.

As an optimization in OS/2, some unmodified pages of operating system DLLs will be written to the swap file, rather than discarded. The amount of time required to reload a page from the swap file tends to be smaller than the amount of time required to reclaim a page from the executable on disk. This is primarily because swapped pages do not need to have their relocation records re-applied, while pages which are read from the executable image will need to have relocation records applied. Due to the design of the OS/2 executable module format, the loader must access at least one additional page of swappable memory for every DLL referenced by that page in order to obtain the relocation information for that DLL. This means that if a page fault occurs and the page must be reloaded from the executable image on disk, the operating system may be required to swap in many additional pages from the swap file in order to service the page fault. Even if all of the pages are in memory, the A (accessed) bit will be set for each of these pages, ensuring that they remain in memory for a longer period of time and forcing other pages to be swapped out as more memory is required.

Although one might initially suppose that a performance benefit could be obtained by swapping all code and data rather than discarding them, experimentation has showed this is not the case. Certain operating system DLLs are accessed by virtually all executables and DLLs, and many pages of these operating system DLLs are accessed very frequently. This tendency, however, has not been observed in the generic case. In general, applications and non-system DLLs are only accessed within a particular process, and many pages of code or data are only accessed a single time or never accessed at all. The amount of time to swap these pages of code and data slowed the system significantly, and the swap file showed an unreasonably large growth.

The OS/2 operating system uses a variant of not recently used (NRU) page-replacement. When the operating system needs to increase the number of pages eligible for replacement, it inspects each virtual page in a circular fashion. If the A (accessed) bit is set, indicating that the page has been accessed, then the A bit is reset to zero and the operating system continues to the next page. If the page has not been accessed, the operating system resets the P (present) bit and ages the page.

Aging the page involves checking to see if the page is a zero-filled page. If the page being aged is not zero-filled, the ager will try to compress the page and put it in the compression buffer. If the page cannot be compressed, the page will be linked to the idle list. Pages on the idle list can be reclaimed at page fault time without incurring an access to secondary storage.

There are two optimizations in the ager worth noting. The first is the fact that the ager will detect a page that is full of zeros. This page can be easily recreated. When OS/2 takes a page fault on an Allocate on Demand page, it returns a page filled with zeros. The ager takes advantage of this fact by marking zero-filled aged pages as Allocate on Demand and linking the physical page onto the free list.

The second optimization is compression. The ager will attempt to compress the page using an algorithm which will replace repeating zeros with a token indicating the number of zeros and the number of bytes (non-zero bytes or single zeros) which follow. This compression algorithm compresses sparse data very well. If the ager can compress an aged page to be smaller than or equal to the size of the entries in the compressed buffer (currently 64 bytes), it will place the compressed data into the compression buffer, link the page frame onto the free list, and mark the virtual page as not present. The time to reload a page from the compressed buffer is much smaller than the time to reload a page from the swap file because no DASD I/O is required. Unlike NRU page-replacement, OS/2 does not inspect the M (modify) bit when the page is aged. This is because OS/2 does not write the modified pages to secondary storage as part of the aging process. At page fault time, the operating system first tries to get a page from the linked list of free page frames. If no free page frames exist, the operating system inspects the first page on the list of idle page frames. If the M bit is clear, indicating the page has not been modified, then the system will simply update the virtual page information for the virtual page which was using the idle page frame, and will use this page frame to satisfy the page fault. However, if the M bit is set, indicating the page has been modified, the operating system must write the page to secondary storage and update the virtual page which was using the idle page frame before the page frame can be stolen.

A dedicated ager thread is used to select page-replacement candidates. When the number of pages on the free page frame list and the idle page frame list drops below a threshold, the operating system wakes up the ager and allows it to execute. When the ager runs, it examines the page table entries (PTE) for all virtual pages for all processes in a circular fashion to find pages which have not recently been accessed.

The algorithm for the OS/2 ager can be summarized as follows:

```
while (true)
  for all page directories in the system
    if all pages in this page directory have the P (present) bit set to 0
      Reset P bit to zero for the page containing the page directory link page frame to idle list
    else
      for all pages in this page directory
        if page is swappable or discardable
          if A (accessed) bit set to 1
            reset A bit to 0
          else (A bit equal to 0)
            if page is full of zeros
              mark page as Allocate On Demand
              link page frame to free list
```

```
            else if page can be compressed
              place compressed contents in buffer
              link page frame to free list
            else
              link page frame to idle list
            endif
            reset P (present) bit to 0
          endif
        endif
      endfor
    endif
    if enough page frames are on the idle list and free list
      block
    endif
    if the ager is monopolizing the CPU
      yield the CPU
    endif
  endfor
endwhile
```

As previously stated, OS/2 is a demand-loading operating system. When an application is started, or when a DLL is loaded, no code or data are actually loaded; the operating system simply reserves the virtual address space. The first time that a page of code or data is accessed a page fault occurs. If a page has been aged (marked not-present), discarded or swapped, a page fault will also occur.

When OS/2 resolves the page fault, it determines where to get the page from the following options: (a) from the swap file, (b) from the compression buffer, (c) from the executable image on disk (from the Loader), (d) Allocate on Demand (a zero-filled page) or (e) reclaim from the idle list. For simplicity, the Compressed Buffer can be viewed as a secondary swap file, and the idle list can be considered a cache for data being written to the swap file. A special type of page fault is a guard page fault. A guard page fault type of fault occurs when an access is made to a guard page, and is typically used to increase stack space.

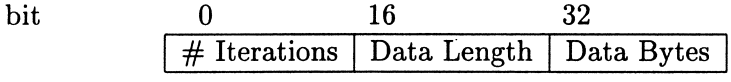
Because OS/2 ages and swaps out pages containing the page directories, a page fault might occur because a page directory is not present. If a page (the target page) which is mapped by a swapped-out page directory is referenced, both the target page and the page containing the page directory will not be present. A page fault will be generated on the page table. After the operating system loads page directory into memory, the faulting instruction will be restarted, causing a reference to the target page, which is still not present. A second page fault will occur which will cause the operating system to bring the target page into memory

The OS/2 Linear Executable (LX) format describes three attributes of each page, 32-bit offset to the page data within the file, 16-bit count of the number of data bytes, and a 16-bit flags field. The flags field describes the characteristics of the page, and is defined as follows:

00 = Valid Page
01 = Iterated Data Page
02 = Invalid Page
03 = Zero Filled Page
04 = Range of Pages

Zero-filled pages do not exist in the executable file, because they can be easily created by the operating system without generating an I/O request to secondary storage. A valid page is a page of uncompressed data. Invalid pages and ranges of pages are defined by the LX format, but have not been implemented.

An iterated data page is a page that has been compressed using an enhanced run-length encoding algorithm. An iterated data page consists of one or more iteration records. The format of the iteration record is the following:



The first field describes the number of times that the data are to be repeated. The data length field describes the number of bytes in the data which follow. The data bytes field is the data which are to be repeated. Iteration encoding differs from the simper run-length encoding in that iteration records can represent repeating patterns of bytes, while the simpler run-length encoding can only represent repeating bytes.

Because of the nature of run-length encoding, pages of code do not compress very well. Many pages of data, however, will compress well. Because of this fact, only data pages are candidates of this type of compression. In general, however, 90% of the pages in an executable are code pages, which means that most of the pages cannot be compressed with this rudimentary compression technique.

3. Proposed approach

3.1. Overview of approach

In order to measure the effects of page-based compression on a demand paging operating system, we first selected an operating system. We chose the OS/2 operating system for several reasons. First and foremost, OS/2 is a demand paging operating system. Second, we had access to the operating system source code, so we could make modifications to the operating system and measure our changes. Third, the operating system runs on widely available hardware, no special hardware is needed.

The second step was to selected a suitable compression algorithm, one which would provide a significant compression ratio as well as a high compression rate.

The OS/2 executable format was then extended to include pages which have been compressed. The OS/2 linker was modified to produce executables with the new type of compressed page. Finally, the OS/2 loader and Workplace Shell were enhanced to recognize and decompress the new type of compressed page.

In addition, software trace points were added to the page fault path and the ager path in order to measure the effects of page-based compression on aging and page faults. A utility application was written to collect and process the data.

Finally, a test scenario was devised which approximates the sequence of events followed by a typical user.

3.2. Exepack2 page-based compression

We had two primary criteria for a compression algorithm. First, it must have a significant compression ratio. If a 4 KB page can only be compressed to 3.7 KB, then the time saved reading the page from secondary storage might be less than the time required to decompress the page. In that case, page-based compression would be detrimental to performance rather than improving performance.

Our second criterion, just as important as the first, is decompression rate. A high decompression rate is essential in order to improve the performance of the page fault path. The compression rate is of lessor importance because the compression will occur at link time. An application developer may see a negative performance impact when building his executable, however the user of the executable should receive a performance benefit. However, because we are interested in application run time, the speed of decompression is of much greater importance.

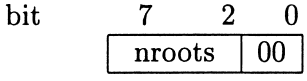
We selected a variant of the LZ77/LZSS compression algorithm to add to the executable format of the OS/2 operating system. This compression algorithm is referred to by the flag which was then added to the linker to produce this new type of compressed page, Exepack2 (IBM, 1996).

This particular compression algorithm was chosen because it met both of our criteria. First, the compression ratio averages 56% for pages of code and data. Second, the decompression rates for LZ77 and derivatives are extremely high. Decompressing does not require a lot of processing, typically the algorithm wiH repeatedly sets a source and destination pointer and performs copies, based on a sequence of tokens. There are no trees or dictionaries to manage, no extraneous bits to examine.

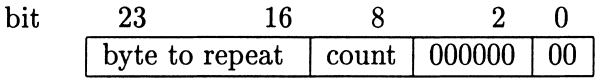
The Exepack2 compression format is a derivative of the LZ77 and LZSS compression algorithms. Recall that LZSS provided two main extensions to LZ77. The first enhancement was adding a tree structure on top of the dictionary for the compressor. This enhancement is of little value in our study because we are concerned with decompression. The second improvement was to the compression format. LZ77 forced compressed phrases to alternate with uncompressed characters. LZSS allows compressed phrases and uncompressed characters to be freely intermixed. Each token contains a single bit prefix which indicates whether the token is an offset/length pair or an uncompressed character.

Exepack2 provides two major enhancements over LZSS. First, Exepack2 provides a two-bit prefix to indicate the primary token type, allowing for four primary token types: nRoots, Short String, Mid String and Long String. Exepac1:2 also maintains byte granularity, which eases processing of compressed data and improves the decompression rate.

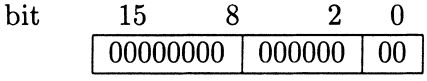
The nRoots token is indicated by a 00 prefix, and can take three forms. The first, and simplest form is a Pure Root token. Bits 2 through 7 specify the number of uncompressed bytes that follow the token header in the compressed data. This field is the nroots field. A value of zero is invalid, so up to 63 uncompressed bytes can be encoded by one token. The primary advantage of the Pure Root token is encoding runs of incompressible bytes, especially when compression is starting and the dictionary is nearly empty. The format of the Pure Root token is the following:



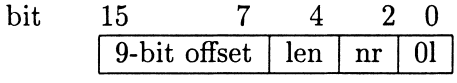
The second form of the nRoots token is a Repeating Bytes Root token, which is indicated by a zero value for bits 2 through 7, which is an invalid value for the Pure Roots token. The third byte in the token is the byte which will be repeated. The second byte of the token is the number of times to repeat the byte, which can be a value from 1 to 255. A value of zero is invalid. The Repeating Bytes Root token provides the advantage of run-length encoding when a character is repeated many times, as in zero-initialized data. The format of the Repeating Bytes Root token is the following:



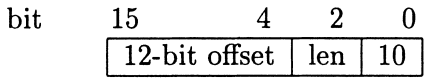
The third form of the nRoots record is the End Code Root token, which is indicated by a zero value in bits 2 through 7 and a zero value for the second byte. Providing an end code allows more streamline (i.e., faster) decompression, without having to check for an end-of-data condition after each token. The End Code Root token is three bytes long, which is acceptable because there will be only one end code for each compressed block of data. The format of the End Code Root token is the following:



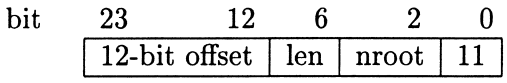
The second token type is the Short String token, indicated by a Ol prefix. Bits 2 and 3 contain the nroots field, which is similar to the nroots field of the Pure Root token. The nroots field specifies the number of uncompressed bytes which follow the token header in the compressed data. The remaining bits contain an offset/length pair. Bits 4 through 6 contain the length field. The number of bytes to copy is computed by adding 3 to the length field. Bits 7 through 15 contain the offset field. The offset is an negative offset from the current location in the decompressed data. The format of Short String token is the following:



The third token type is the Mid String token, indicated by a 10 prefix. The Mid String token does not contain an nroots field. The remaining bits specify an offset/length pair. Bits 2 through 3 contain the length field. The number of bytes to copy is computed by adding 3 to the length field. Bits 4 through 15 contain the offset field. The offset is an negative offset from the current location in the decompressed data. The format of the Mid String token is the following:



The fourth token type is the Long String token, indicated by a 11 prefix. the Long String token is similar to the Short String token, but has larger fields. Bits 2 through 5 contain the nroots field. The nroots field specifies the number of uncompressed bytes which follow the token header in the compressed data. The remaining bits contain an offset/length pair. Bits 6 through II contain the length field. The number of bytes to copy is computed by adding 3 to the length field. Bits 12 through 23 contain the offset field. The offset is an negative offset from the current location in the decompressed data. The format of the Long String token is the following:



3.3. Linker modifications

We extended the LX format to include a fifth type of page:

OS = Compressed Page

A compressed page contains a page which has been compressed using the Exepack2 compression algorithm.

L1NK386, the OS/2 linker, was modified to accept a command line parameter which would indicate pages of type OS (compressed with Exepack2) that should be produced. Compression routines were added in order to produce page type 05.

The existing versions of L1NK386 use the /EXEPACK command line parameter to indicate that iteration compression should be performed on pages of data. If /EXEPACK is not specified, iteration compression is not performed so L1NK386 will produce pages of type 00 (valid) and 03 (zero-filled). If /EXEPACK is specified, LINK386 may also produce pages of type 0 1 (iterated). We extended the /EXEPACK parameter to take three forms

/EXEPACK
/EXEPACK: 1
/EXEPACK: 2

When the first form, /EXEPACK is specified, LINK386 will enable iteration compression. If a page of code is to be produced, it is emitted as a valid page and the Object Page Table is updated to indicate a valid page. If a page of data is to be produced, the linker first checks to see if the page is full of zeros. If so, a zero-filled page is indicated in the Object Page Table and no data page is emitted. The linker then performs iteration compression on the data page. If the compression results in space savings in the executable file, the linker emits the iterated page and indicates so in the Object Page Table. If the compression will not result in space savings then the linker emits the uncompressed (valid) page and indicates that the page is valid in the Object Page Table.

When the /EXEPACK:1 form of the parameter is specified, L1NK386 performs exactly as if /EXEPACK was specified. It may seem unnecessary to produce a redundant form of the /EXEPACK parameter, however, we feel that further modifications might be made in the future which would necessitate extending the /EXEPACK parameter further, perhaps adding /EXEPACK:3, /EXEPACK:4, etc. In this case, the /EXEPACK:1 form would be more consistent with the newer forms, while the /EXEPACK form would be required for backward compatibility.

The third form, /EXEPACK:2 indicates that the linker should enable both iteration compression and Exepack2. If a page of data is to be produced, the linker first checks to see if the page is full of zeros. If so, a zero-filled page is indicated in the Object Page Table and no data page is emitted. Otherwise the linker will not distinguish between pages of code and pages of data. The linker will then perform iteration compression and Exepac compression on the page of code or data. If the Exepac page is smaller than the iterated version, and is at least 512 bytes smaller than the valid version of the page, LINK386 will emit the Exepack2 version of the page. Otherwise LINK386 will emit the smaller of the valid and iterated version of the page. The correct page type will be indicated in the Object Page Table.

Before emitting an Exepack2 page, L1NK386 verifies that the page size is at least one sector size (512 bytes) smaller than the size of the valid version of the page. This check ensures that the I/O request for the Exepack2 page will be for fewer sectors than the I/O request for the equivalent valid page.

In addition, a post-link application, REPACK is now available which can be executed against a module to convert the pages from one type to another. REPACK follows the same rules as L1NK386 with one notable exception. REPACK recognizes the /EXEPACK:0 option as an indicator that no iterated or Exepack2 pages should be produced. The same result can be obtained by invoking L1NK386 without the /EXEPACK parameter.

3.4. Loader modifications

The changes to the OS/2 loader were quite straightforward. First, decompression routines were added to the loader to perform the decompression. Then the code path was modified to call the decompression routines when an Exepack2 page was encountered.

The loader interrogates the page type in only two instances. In the first instance, the loader is differentiating between zero-filled pages, which do not necessitate a read from the executable file, and valid or iterated pages which do require that the page be read from the executable file. We added Exepack2 pages to the valid or iterated page path.

The second time the loader checks the page type is in an 'if' statement. The loader checks for an iterated page 50 that decompression can be performed. We simply added an 'else if' case to the existing 'if' statement. In the case of an Exepack2 page, the loader calls a routine to decompress the page.

3.5. Workplace shell modifications

The OS/2 Workplace Shell is the graphical user interface layer of the operating system. The shell examines executables modules to extract resource data, such as bitmaps, icons, menus, strings, etc. The shell code for extracting resources is isolated to a single code path, which was easily modified to check for an Exepack2 page. If a Valid page is found, the shell will allocate a buffer from the heap, read the data into the buffer, and copy the data into the application's buffer. If an Exepack page is found, the shell will allocate a buffer from the heap, read the data into the buffer, and decompress the page into the application's buffer.

3.6. Software trace additions

In order to measure the effects of page-based compression on a demand-paging operating system, we modified the OS2KRNL (kernel) to maintain information on the pages which are being page-faulted into memory and the pages which are selected as victims for page-replacement.

We identified four actions which the kernel may perform on page-replacement victim pages. As the Ager thread ages pages at idle time, it examines the page-replacement candidate to determine if the page contains all zeros. If so, the ager will mark the page frame as free and mark the virtual page as Allocate on Demand. If a page fault occurs on an Allocate on Demand page, the operating system will return a zero-filled page.

The second action performed by the Ager thread at idle time is to attempt to compress the page-replacement candidate. If the 4 KB page can be compressed to 64 bytes or less, and an entry is available in the compression buffer, the page frame will be marked free, the compressed data will be copied to the compression buffer, and the virtual page will be updated to indicate that it is not present (the P bit will be reset) and that its contents are available in the compression buffer.

If the page-replacement candidate is not zero-filled and cannot be compressed, the ager will reset the P bit and will link the page to the idle list. Further actions on the page-replacement candidates take place at page fault time.

If there are no free page frames when a page frame is required at page fault time, the pager will steal a page from the idle list. If the first page on the idle list is a discardable page or a clean swappable page, it is selected as the victim for replacement. The virtual page information is updated to indicate that the page has been discarded and the page frame is removed from the idle list.

However, if the first page on the idle list is a dirty swappable page, the contents cannot be simply discarded. The pager will attempt to find up to seven more dirty swappable pages on the idle list. The dirty pages are removed from the idle list and written to the swap-file in a single write request. When the write request has been completed, a page frame will be donated to satisfy the page fault. The remaining page frames will be linked to the free list.

The software trace counters added for pages going out of physical memory are: (a) pages going out of memory, (b) number of dirty pages swapped out, (c) number of discardable pages discarded, (d) number of clean swappable

pages discarded, (e) number of zero-filled pages detected/discarded and (f) number of pages going to compression buffer

We also added software trace points to collect information on pages being page faulted into memory. There are five general categories of page faults, based on where the page comes from.

The first and simplest case is a page fault on a page which is on the idle list. A valid copy of the page exists, however the present (P) bit was reset when the page was selected as a candidate for page-replacement by the ager. In this instance, the operating system simply removes the page from the idle list and sets the P bit to 1. When the faulting operation is restarted, the page will be accessed and the hardware will set the accessed (A) bit to 1. Reclaiming a page from the idle list is very fast because no I/O is required and the contents of the page do not have to be created.

The second case for a page coming into memory is a page fault on a locate on Demand page. The operating system will get a page frame from the free list or select a victim page from the idle list, in the manner described above. Then the operating system will fill the page with zeros using three Intel machine instructions to set up parameters and a single Intel machine instruction to fill the page.

```
LES     EDI,[PagePointer]     ;Set ES:EDI registers to point to the page
XOR     EAX,EAX               ;Set EAX register to 0
MOV     ECX,1024              ;Set ECX register to PAGESIZE/4
REP     MOVSD                 ;Fill ES:EDI with EAX, size = ECX × 4
```

The REP MOVSD instruction will cause the processor to copy the contents of the EAX register into the location pointed to by the ES:EDI registers, the number of times indicated by the ECX register, incrementing the EDI register by 4 after each repetition. The page information will be updated to link the virtual page to the page frame, the P bit will be set to one, and the faulting instruction will be restarted.

The third case for a page coming into memory is a page coming from the compression buffer. Again, a suitable free or idle page frame must be selected. Then the operating system decompresses the contents of the compression buffer entry into the page. The compression buffer entry is marked free so that it may be used again, if necessary. The page information will be updated to link the virtual page to the page frame, the P bit will be set to one, and the faulting instruction will be restarted.

The fourth case for a page coming into memory is a page from the swap file. After acquiring a page frame from the free or idle list, the operating system starts an I/O request to read the requested page from the swap file into the page frame. The thread then blocks until the I/O is complete, during which time the processor is available to other threads that are ready to execute. When the I/O is completed, the page information will be updated to link the virtual page to the page frame, the P bit will be set to one, and the faulting instruction will be restarted. The entry in the swap file, however, will not be freed but will remain linked to this virtual page. The swap file copy of the page will remain valid unless and until the page is modified. If the page is again selected as a victim for page-replacement and it has not been modified, then no I/O will be required to copy the contents to the swap file because a valid copy exists.

The fifth and final case for a page coming into memory is a page which comes through the loader. The first time that a page of code or data is accessed, the page fault handler will call the loader to load the page. Pages of data will be marked swappable, so if subsequent page faults occur (i.e., after the page was selected as a victim for page-replacement) then the fault will not resolve to the loader. Pages of code, however, will be marked as discardable. If subsequent page faults occur on pages of code, the fault will resolve to the loader.

Page faults from the loader can be further classified by page type. Page types include valid, iterated, Exepac and zero-filled. In addition, there are 'implied' pages. A valid page is not compressed in the executable module on secondary storage. The contents of a valid page can be read directly into the page frame. Iterated and Exepac pages, however are compressed. The contents of the page must be read from the executable into a buffer, then decompressed into the page frame. Zero-filled pages require no access to secondary storage, but can be created in a manner identical to Allocate on Demand pages. After the contents of the valid, iterated, Exepack2 or zero-filled pages are in the page frame, the relocation records, or fixups, can be applied.

A fifth type of loader page is an 'implied' page. These are zero-filled pages which do not have fixups and no pages after them in the code or data object have fixups. Implied pages are treated as zero-filled pages with no fixups. Implied pages have two advantages over marking these pages as zero-filled pages. First, implied pages do not have entries in the Object Page Table or the Fixup Page Table of the executable file, so they save space in the executable file. It also saves

memory because the loader maintains a copy of the Object Page Table and the Fixup Page Table in swappable memory. The second advantage is speed. The loader simply calculates the page number within the object for the faulting page. Then the loader checks to see if the page number is greater than the number of Object Page Table entries for the object. If so, then the page is an implied page 50 and the loader does not even have to access the Object Page Table or the Fixup Page Table. Accesses to the Object Page Table or the Fixup Page Table could generate an additional fault, further slowing the system.

A software trace counter was added to calculate the total number of page faults. We also added software trace counters to determine where the pages were coming from: (a) swap file, (b) compression buffer, (c) loader, (d) reclaim (from aged list) or (e) Allocate on Demand (zero-filled). We included more counters to further distinguish the pages coming from the loader by page type: (a) valid, (b) iterated, (c) Exepac, (d) zero-filled or (e) implied.

3.7. Data collection

We modified the DosQuerysyslnfo Application Programming Interface (API) to return the data which were collected from the software trace points. We also wrote a collection utility which would gather the results and store them in files on secondary storage.

The DosQuerySyslnfo API takes four parameters, starting index, ending index, address of the buffer to return results, and size of buffer (in bytes). 26 indices were defined, each associated with a 32-bit data value, such as minimum timeslice in milliseconds, total physical memory in megabytes, and the process ID of the foreground process. If only one data value is being requested, Ending Index will be equal to Starting Index. A consecutive range of indices may be queried with one API call by setting Starting Index equal to the lowest requested index and setting Ending Index equal to the largest index requested. The size of the buffer must be greater than or equal to the value computed by ((Ending Index − Starting Index + 1)×4). Simply stated, the buffer must be large enough to hold the data requested.

We extended the DosQuerySyslnfo API to contain additional index definitions for each of the software trace counters which are described in Section 3.6.

We also wrote a utility which invokes the DosQuerySyslnfo API to read the new counters. The application, QSYSTEM, queries the values of the new indices on start-up and upon user request (by pressing the Enter key when QSYSTEM has the foreground focus). QSYSTEM produces three output files, OUT, IN and LOADER. OUT contains information associated with pages going out of memory. IN contains information associated with pages coming into memory. LOADER contains information associated with pages coming into memory via the loader. Each file contains the initial values of each of the counters, the delta of the counter values each time the values are queried, the sum of the deltas (which gives the total number of increments for the scenario), as well as the final value of the counters. When the Q key is pressed, the QSYSTEM application closes the output files and exits.

3.8. Test sequence

We created a test sequence which would approximate the sequence of events followed by a typical user. The first step of the test sequence is to power up the computer, which will load the operating system. Then a series of folders and applications, including word-processor, calendar application and realtime video, are opened. The applications are executed, then the folders and applications are closed.

This test sequence is an approximation of the steps followed by a user during a session at the computer. First, the user must power on the computer. Then the user will start and execute a number of applications. In order to start an application, the folder containing the application must be opened. Finally, when the user is finished with an application and/or folder, it will be closed. At the end of the session, all objects which have been opened will be closed before the system is powered off.

The sequence of events used to collect data is listed below. The system was allowed to quiesce completely between each step before the data ensuring all paging activity has finished.

Test Sequence

boot	Power on the computer
1	Start IBM Works folder
2	Open IBM Works Application
3	Create new document
4	Close IMB Works application

5	Open Daily Planner application (implicitly starts Event Monitor app)
6	Open OS/2 Command Window
7	Open OS/2 Tutorial
8	Close OS/2 Tutorial
9	Open Multimedia folder
10	Open Digital Video application
11	Play MACAW.AVI file
12	Close Digital Video application
13	Close Multimedia folder
14	Close IBM Works folder
15	Close OS/2 Command window
16	Close Daily Planner application
17	Close Event Monitor application

3.9. Hardware

Our test computer consisted of an IBM PS/2 Model 8595 with a 66 MHz Intel 80486 processor. The video configuration was composed of an XGA/2 adapter and an IBM 8514 (14-inch) display running in I 024X768 video resolution. The computer had an IBM 32-bit SCSI-2 adapter connected to a 540 MB Maxtor SCSJ hard file.

We tested four memory configurations, 7 MB (severely constrained), 8 MB (moderately constrained), 10 MB (mildly constrained), 16 MB (nearly unconstrained) and 32 MB (unconstrained).

3.10. Software setup

We installed OS/2 on the test computer, allowing the operating system to format the hard file at the beginning of installation. The optional items which we selected to install were the High Performance File System (HPFS) installable file system, Multimedia, and the IBM Works application from the Bonus Pack.

In our first scenario, we executed the REPACK application against all system modules using the /EXEPACK:0 option. REPACK converted all iterated and Exepack2 pages to valid pages. We then ran our test scenario on the five memory configurations and collected the results.

In our second scenario, we executed the REPACK application against all system modules using the /EXEPACK:2 option. REPACK performed iteration compression and Exepack2 compression on each pages, and emitted the smallest type of page, valid, iterated, or Exepack2. We again then ran our test scenario on the five memory configurations and collected the results.

4. Results

The effect of page-based compression on the page fault rate is investigated, as well as its effect on performance. Finally, the effect of page-based compression on the seal ability of the system is analyzed.

4.1. Analysis of data

Analysis of the data shows a slight increase in the number of page faults in a system with severely constrained memory when page-based compression is added, but a reduction in page faults in all other memory configurations. The total time spent processing page faults, however, decreased in all memory configurations. This savings was calculated based on the type and number of page faults, and was compared to actual measurements of the time saved.

4.2. Effect of page-based compression on page fault rate

Our testing indicates a slight increase in paging activity with the introduction of page-based compression in a severely memory constrained environment, but a decrease in all other memory configurations, as shown in Table 1.

Table 1
Total page faults

	Without compression	With compression	Decrease
7 MB	15631	16744	−7.2%
8 MB	12186	12234	0.4%
10 MB	9627	9288	3.5%
16 MB	7251	7032	3.0%
32 MB	6785	6637	2.2%

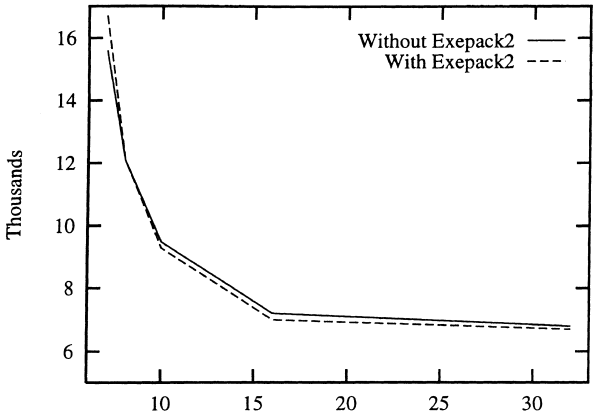


Fig. 1. Total page faults.

As seen in Fig. 1, plotting a graph with system memory on the *x*-axis and total page faults on the *y*-axis indicates an exponential distribution. The curve showing total page faults with page-based compression is shifted slightly towards the bottom of the graph and slightly towards the right of the graph.

The shift toward the bottom of the graph indicates that a system with page-based compression generates fewer page faults than a system without page-based compression, when the shift towards the right is ignored. Therefore, a system with page-based compression uses less virtual memory than a system without page-based compression. The differences in page faults on an unconstrained system are almost exclusively Allocate on Demand pages during step 2, loading the IBMWorks application.

Further inspection of the operating system code shows that the difference is due to the shell's treatment of resource data, such as bitmaps, fonts and icons. When a resource is contained in a Valid page, the shell reads the page into a freshly allocated section of the heap, then copies it into the application buffer. When a resource is contained in an Exepac page, the operating system reads the compressed page into a newly allocated buffer on the heap, then decompresses it into the application buffer.

The heap management decommits memory whenever complete pages within the heap are unused. Decommitting a page will free the page frame, if one was associated with the virtual address, and will cause future accesses to the virtual address to generate an access violation. When a heap allocation request will span an unused page, the heap management will commit the page. Committing the page will cause the virtual address to become valid and be marked Allocate on Demand.

A heap allocation request for a buffer for a valid page will require 4096 bytes, while a heap allocation request for a buffer for an Exepac page will on average require 2294 bytes. Thus, there is a higher probability that accessing a resource in a Valid page will cause a page fault, versus accessing a resource in an Exepac page.

The shift towards the right of the graph indicates that a system with page-based compression requires slightly more physical memory than a system without page-based compression. This behavior is due to the loader treatment of compressed pages. When the loader resolves a page fault for a Valid page, it reads the page into the target page frame. However, when the loader resolves a page fault for a compressed page, it must allocate a resident (also known as wired, locked or non-swappable) buffer to read the page into. The page is then decompressed from the buffer into the target page frame. There is no page fault on the resident buffer because it is not swappable. However, a victim page may need to be selected for page-replacement if no free page frames exist.

In an unconstrained environment, the allocation of the buffer will not cause a victim page to be removed from memory because free frames will exist. However, in a severely constrained environment a victim page will be

selected for page-replacement and there will be a high probability that the page will be needed again in the near future.

Therefore, slightly more memory is required to process a compressed page during a page fault, thus reducing the effective amount of memory in the system. In a severely constrained environment, reducing the available memory even by a modest amount will cause a noticeable affect on system performance as the system approaches a thrashing state. In a less constrained environment, the system can afford to give up a bit of memory for a short period of time because it takes less time to process a page fault for a compressed page than for a Valid page. The time saved due to page-based compression will offset the cost (time) due to having less available memory. The break-even point is just below 8 MB where the two curves crosses.

4.3. Effect of page-based compression on performance

The amount of time required to satisfy a page fault for each type of page is shown in Table 2.

Based on these timings, it is clear that the time required to load a page with the new page-based compression (Exepack2) is two-thirds of the amount of time required to load an uncompressed (Valid) page. Using these timings, the time required by the system to process the page faults during system boot is shown in Table 4. The actual time required to boot is shown in Table 3. The values in Table 3 are the average of measurements from three boots for each memory configuration. Each measurement was within 0.2 s of the average.

The graph of measured boot time is shown in Fig. 2. Again, the curves show an exponential growth. The important point to notice is that the break-even point along the curve is no longer just below 8 MB. Because it is quicker to load an Exepack2 page than a Valid page, the break-even point is below 7 MB. Even though a severely constrained system (7 MB) causes more page faults, the page faults are processed quicker. An Exepack2 page can be loaded in approximately two-thirds the time it takes to load a Valid page, 50 the number of page faults that the system can handle before thrashing should theoretically increase by 50%. This is only true, however, if all page faults resolved to Valid pages before, but now resolve to Exepac pages.

The time required to process all page faults during the test scenario is shown in Table 5 and Fig. 3. The graph in Fig. 3 is virtually identical to the graph in Fig. 2. This indicates that page-based compression will improve performance at all memory configurations at boot time as well as at application execution time.

4.4. Scalability analysis

Scalability, strictly defined is the ability of a parallel algorithm to use an increasing number of processors efficiently (Kumar et al., 1994). We shall use a looser definition of the word for our research: The ability of an algorithm or program to use an increasing amount of a resource efficiently.

Table 2
Number of machine cycles to satisfy a page fault

Page type	Cycles
Loader – valid	301103
Loader – iterated	152859
Loader – Exepack	201307
Loader – zero filled	10578
Loader – implied	9864
Swap in	93914
Reclaim	2034
Allocate On Demand	9728

Table 3
Measured seconds to boot

	Without compression	With compression	Savings
7 MB	42.3	41.2	1.1
8 MB	39.2	37.9	1.3
10 MB	36.0	34.8	1.2
16 MB	35.5	34.2	1.3
32 MB	35.7	34.5	1.2

Table 4
Calculated seconds to process page faults through boot

	Without compression	With compression	Savings
7 MB	7.56	6.24	1.32
8 MB	6.97	5.24	1.73
10 MB	6.09	4.52	1.57
16 MB	5.99	4.44	1.55
32 MB	5.99	4.44	1.55

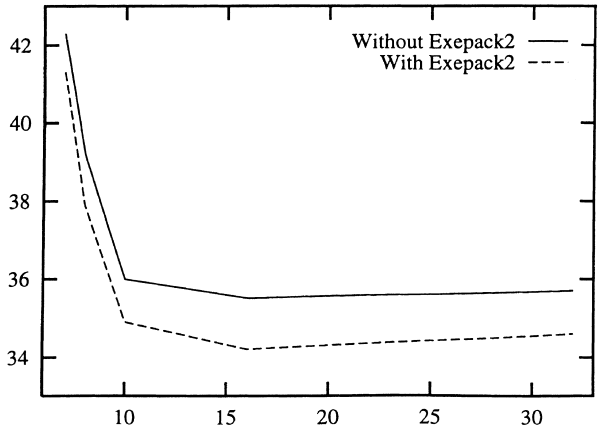


Fig. 2. Measured seconds to boot.

Table 5
Seconds to process page faults for test scenario

	Without compression	With compression	Savings
7 MB	24.86	22.53	2.33
8 MB	18.03	14.08	3.95
10 MB	13.60	10.05	3.55
16 MB	10.85	7.94	2.91
32 MB	10.64	7.77	2.87

Tables 3 and 4 and Fig. 2 indicate the performance of system boot as memory is added. It is clear that the system benefits from adding memory up to a point. However, going from 16 to 32 MB offers no advantage because the system uses less than 16 MB to boot.

Similarly, Table 5 and Fig. 3 depict the performance of our test scenario as memory is added. As we expect, the system again exhibits better performance as the amount of memory is increased. However, our test scenario utilizes more memory than system boot, so the system displays improved performance when moving from 16 to 32 MB.

Hence, the performance improvement due to increasing the amount of physical memory on the system does not increase linearly, but is bounded by a law similar to Amdahl's law (Amdahl, 1967) which states that if a problem size W has a serial component W_s, then W/W_s is the upper bound of its speedup, no matter how many processors are added. In our case, if a problem size S has a non-page fault component S_{exec}, a page fault component due to initial page load S_{pfinit} and a page fault component due to paging activity S_{pfpage} then the speedup (ratio of specific run time to run time with additional memory) is bounded by

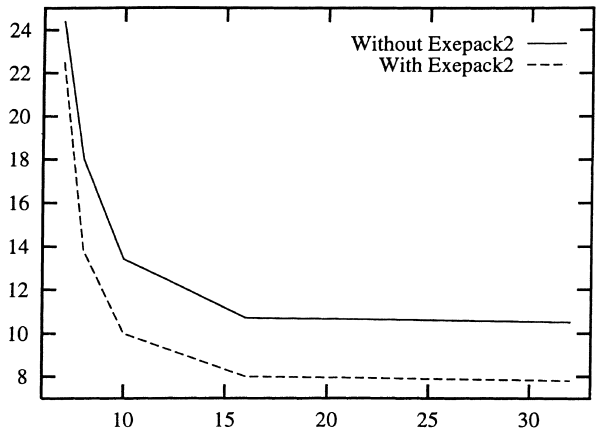


Fig. 3. Seconds to process page faults for test scenario.

$$\frac{S_{\text{exec}} + S_{\text{pfinit}} + S_{\text{pfpage}}}{S_{\text{exec}} + S_{\text{pfinit}}}. \tag{1}$$

In our study, we have found that page-based compression improves performance in all memory configurations, however it does not affect scalability. This is due to the fact that page-based compression reduces the value of S_{pfinit} and S_{pfpage} by as much as 1/3 (see Table 2, Valid pages vs Exepac1:2 pages), but does not redistribute the workload in any other way. Hence, the profile of the performance curves in Figs. 2 and 3 are virtually identical between a system with page-based compression and a system without page-based compression. The curves-based compression, which indicates S_{pfinit} and S_{pfpage} are increased by a constant factor.

4.5. Validation of measurements

The modified loader and QSYSTEM utility have been executed on several systems including an IBM Model 8580 with a 16 MHz 80386 processor and a 300 MB IDE hard file, an IBM Model 8595 with a 33 MHz 80486 processor and an IBM SCSI adapter with a 300 MB IBM hard file, and an IBM Model 8571 with a 33 MHz 80486 processor and an IBM SCSI adapter with a 400 MB IBM hard file. The measurements on each system are consistent with results on other Systems with similar memory configuration. Differing hardware platforms (excluding varying the amount of memory in the Systems) do not seem to have a significant impact on the measurements attained.

5. Conclusion

5.1. Summary

An overview of compression and lossless and lossy algorithms was presented in Section 2. LZ77 and LZ78 and variants were explored in detail. The OS/2 operating system application loading and page-replacement policy, including aging and the page fault process were also described.

Our approach was described in Section 3. The Exepac compression format, which is based off LZ77 and variants, was described in detail. Modifications to utilities and to the operating system were listed, as well as the hardware setup, software setup and test sequence.

The results of this study were detailed in Section 4. First, the effect of page-based compression on the page fault rate are analyzed, then the effect of page-based compression on performance is analyzed. Finally, the effect of page-based compression on the scalability of the system is provided.

5.2. Contribution

Based on the results in Section 4, we find that a system with page-based compression uses less virtual memory than a system without page-based compression. This results in fewer page faults on a system with page-based compression in an unconstrained environment.

However, a system with page-based compression requires more locked (non-swappable) memory than a system without page-based compression. This effectively reduces the amount of physical memory available for the remainder of the system. In an unconstrained environment, this will produce no noticeable effect. However, in a memory constrained environment, a small reduction in memory will increase the number of page faults generated.

Interestingly, we find that the savings generated from page-based compression offsets the additional time required to process the increased number of page faults, even in a memory constrained environment. Therefore, page-based compression improves the performance of a demand paging operating system, regardless of the memory configuration.

5.3. Future work

This study was conducted on a uni-processor computer and a uni-processor version operating system. It would be interested to study the effects of page-based compression in a Symmetrical Multiprocessing (SMP) environment, where the results could be very different.

In a severely or mildly constrained system, many page faults resolve to the swap file. It might be advantageous to compress pages before writing them to the swap file. The benefits would be twofold. Both the swap-out time and the swap-in time would be faster. Many operating systems have fixed size swap frames. Adding compression would cause variable size frames, the change to the operating system pager would not be trivial. Alternately, adding a second swap file with a fixed, but smaller, frame size could be added. If a page can be compressed below the frame size of the smaller swap file it can be written to the second swap file.

This study dealt with DASD I/O on a stand-one workstation. Work could be done to study the effects of compression when working across a high-speed network and in a cluster-based system.

Operating Systems typically divide their address space into several arenas or regions. The paging activity could be studied on a per-arena or per-region basis to better understand how the virtual memory is utilized. With this data, the aging and/or paging algorithms can be modified to treat greater utilized and lesser utilized regions differently.

The OS/2 Linear Executable Format was designed with demand paging in mind. Other executable formats do not contain page-based structures and tables. There are key aspects of several executable formats, such as LX, ELF and COFF, that could be changed to increase performance. For example, the relocations associated with a page do not have a spatial locality of reference with the actual page within the executable file. A detailed study of these formats and the memory/structures used at page fault time is needed in order to optimize and extend the executable formats to reduce memory usage and improve performance.

6. Trademarks

References

Amdahl, G.M., 1967. Validity of the single processor approach to achieving large scale computing capacities. AFIPS Conference Proceedings, 483–485.

Deitel, H.M., Kogan, M.S., 1992. The Design of OS/2. Addison-Wesley, Reading, MA.

Gersho, A., Gray, R.M., 1992. Vector Quantization and Signal Compression. Kluwer Academic Publishers, Boston, MA.

Huynh, K.D., Brew, G.E., 1993. Memory Management in OS/2 2.0: Part II, IBM Innovations, Issue 1.

Huynh, K.D., Brew, G.E., 1993. Memory Management in OS/2 2.0: Part III, IBM Innovations, Issue 2.

IBM OS/2 16/32-bit Object Module Format (OMF) and Linear eXecutable Module Format (LX). 1996. IBM Corporation, Austin, TX, available by request from opsys2@bcrvm1.bocaraton.ibm.com.

Kumar, V., Grama, A., Gupta, A., Karypis, G., 1994. Introduction to Parallel Computing: Design and Analysis of Algorithms. The Benjamin/Cummings, Redwood City, CA.

Nelson, M., 1992. The Data Compression Book. MT Books, New York.

Welsh, Terry, 1984. A technique for high-performance data compression. IEEE Computer 17 (6), 8–19.
Wirth, N., 1995. A plea for lean software. Computer 28 (2), 64–68.
Ziv, J., Lempel, A., 1977. A universal algorithm for sequential data compression. IEEE Transactions on Information Theory 23 (3), 337–343.
Ziv, J., Lempel, A., 1978. Compression of individual sequences via variable-rate coding. IEEE Transactions on Information Theory 24 (5), 530–536.