RC22338 (W0202-041) February 14, 2002
Computer Science

IBM Research Report

A Note on Guaranteed Forward Progress in Compressed Memory Systems

Peter A. Franaszek, Dan E. Poff

IBM Research Division
Thomas J. Watson Research Center
P.O. Box 218
Yorktown Heights, NY 10598

A Note on Guaranteed Forward Progress in Compressed Memory Systems

Peter Franaszek and Dan Poff

IBM T. J. Watson Research Center

Abstract: In a computing system where the contents of main memory are held in compressed form, a possible problem is that changes in compressibility will lead to a condition where pageouts need be done, but the process to accomplish this may require more physical memory resources than are available, resulting in a system hang. The avoidance of this condition we term *guaranteed forward progress,* or GFP. There have been a number of proposals for system structures which ensure this property. In this paper, we briefly review these, then concentrate on two new directions, termed respectively *orthogonal paging* and *dynamic address disabling*. Either of these approaches appears to provide a feasible and possibly preferable means for obtaining GFP. For convenience in the discussion, we generally assume an IA32 processor architecture, and IBM's Memory Extension Technology (MXT).

1. Introduction

Main or random access memory is generally the most expensive component of the central electronic complex of server class machines. This observation has led to a number of system design efforts where part or all of the memory contents are held in compressed form, and decompressed on access (1, 2, 3, 4, 5). An example of such a system is shown in Figure 1. Here the cache hierarchy has three levels: L1, L2, and L3, where L3 is furthest from the processor(s). This is the configuration used in IBM's announced MXT (memory extension technology) design (2). There L3 cache lines contain 1K bytes, and the content of a line is compressed on write back to main memory. The content of main memory is decompressed on a cache line miss when data is being fetched to the cache from main memory. Typically, all contents of main memory may be held in compressed form. Accesses to memory involve an extra level of indirection, the translation between processor 'real' addresses and the physical location(s) of the contents. The translation is done via a content translation table or CTT. Physical memory space is allocated in units of 256 bytes. These units are termed *sectors*. The physical space or number of sectors occupied at any given time is a function of the number of pages in memory, and of their compressibility. Some page contents are fully compressible; that is, they are fully described in the CTT, and thus do not require any allocation of sectors. In the current MXT implementation, the real memory size can be as great as twice the nominal physical space required. That is, the number of real addresses corresponds to twice the amount of physical memory.

Compression results in a situation where there is no longer a one to one relation between the number of pages held in memory and the amount of physical space occupied. This means that the system must include a mechanism for monitoring the amount of free physical space available. This can then lead to pageouts if the physical space is running low, and conversely to increasing the number of assigned real addresses when space is available. This mechanism would operate in conjunction with, and in a manner roughly analogous to today's virtual memory managers, which attempt to ensure that there is always a sufficient number of available page frames.

However, compressibility can vary much faster than the I/O bandwidth, since data can be modified and recompressed at the full memory bandwidth. Further, the act of deleting pages from memory can itself cause expansion due to the modification of various data structures, and also as a result of write backs from the caches during this process. This means that some provision need be made to avoid a situation where pageouts need be done, but the amount of available free space to do so is insufficient; this condition causes a system crash or hang.

There appear to be two general approaches to the problem of guaranteeing forward progress, which we respectively term *safety* and *dynamics*. In the safety approach, the number of pages to which read/write access is permitted is limited by the physical space available. That is, the number of pages which can be modified and thus require additional space is limited. Attempts to access additional pages results in an interrupt, which then permits pageouts to occur. A requirement of this method is that all essential memory contents required for pageouts are accessible. In the dynamic approach, actions need be taken to stop further memory degradation, followed by paging to recover required space.

We illustrate these approaches via some examples. The following two are safety based:

A1) Compressed Page Store:

This is the approach taken by Loughbourough University (3). Here a set of pages are compressed at any given time. These are accessed as in today's machines. The rest of memory is essentially a paging store holding compressed data. Accesses of any of these pages results in an interrupt, a check on the available physical space (with the possibility of a pageout), and finally the decompression of the desired page, as well as possibly the recompression of a currently compressed page. The operating system operates as in today's machines, with decompression or movement of pages done only under guarantees of adequate space.

A2) M-Space:

A technique termed M-space, or mapped space (7.). Here a subset of pages is subject to read/write access, with read-only access for the others. A read/write request to a read-only enabled page causes an interrupt. Access to the desired page is then granted only if there is sufficient physical space, as in A1. Differences here from the prior technique include: a) essentially all pages may be held in compressed form, b) read-only pages are always accessible, and c) a read/write request can be satisfied without decompressing an entire page.

Potential drawbacks of M-Space are associated with the overhead of moving pages between say read-only and read-write status. This can require TLB invalidates, updates to data structures and flushes of cache contents, as well as a requirement for substantial free-space reserves. However, this technique offers advantages over A1 in that only L3 lines which are actually accessed need be decompressed/compressed. Also it is not necessary to keep a large part of memory unmapped. Another problem with the above two approaches is related to the fact that all programs / data needed for paging needs to be accessible independently of the amount of free physical storage. This includes metadata describing disk addresses. This can be a problem with applications such as web accelerators, which occupy space in the system cache, and where metadata required for paging may be scattered throughout the system cache, and thus difficult to identify. Here the entire system cache, comprising the bulk of storage, may be need to be held in uncompressed or mapped space, thus eliminating any advantage of compression.

The second or dynamic type of approach requires that the shortage of free space be detected, and some action taken before a point of no return. That is, that the action taken ensures that whatever additional space is used during the process of recovery is not larger than the reserves of physical space. There are a number of variations:

B1) Knowledge:

Here analysis/experimentation is used to determine how large a reserve of free physical space is required for safe operation. Any required paging is done by the operating system, with no modifications for restricting address ranges etc. A potential problem here is that it may be difficult to analytically determine what is worst case behaviour, and that the worst case reserves computed with simplifying assumptions may be excessively large. A simple approach to GFP with knowledge is to have all except for a limited number of pages required to be unmodified and listed on a special data structure (the outlist discussed below) with a guaranteed small footprint. Pages on this list can then be eliminated as necessary for space recovery, with a guaranteed bound on the space required. Knowledge is the basis for the current MXT support software (8).

B2) Dynamic address disabling (DAD):

Here sufficient free space is maintained to guarantee access to operating system pages, for example reserving sufficient space for the maximum expansion of such pages. Access to user space is disabled given a low available memory interrupt. This limits the amount of expansion which can occur. The disabling could be either by software (as described below), or by hardware, the latter requiring a change in processor architecture.

B3) Orthogonal paging (OP):

Here a software component with a guaranteed small footprint 'steals' and outputs pages after the operating system is stopped by an interrupt. These pages could be written to a swap area on disk. After sufficient space is recovered, normal operation can resume, with special treatment of pages which have been stolen. We show that if this operation is done properly, the GFP property can be obtained. A hypervisor, if properly structured, can be used to implement orthogonal paging.

B4) Cache-based Interrupts:

As a basis for an alternative approach, a property of the above-mentioned MXT architecture is that expansion of memory contents occurs only on cache cast outs. A machine might then be designed so that a cache miss to an address outside some specified range (e.g. including the kernel) would cause an interrupt. A problem with this method is that in modern processors, there are multiple instructions in flight at any given moment. Here it may not be feasible to identify the instruction associated with the interrupt. An advantage of this technique is that applications need not be stopped unless they cause cache faults to locations outside allowed areas. This limits expansion to the cache contents plus the permitted area.

B5) Combinations:

In some cases, such as web accelerators running in kernel space, it may be necessary to use a combination of the above. (Alternatively, a web accelerator, which works in conjunction with a user mode web server, could be bypassed whenever physical memory runs short.}

Each of the above dynamic approaches will in general require some space reserves, as well as restrictions on incoming I/O. We discuss this in section 3. The amount of such reserves corresponds to the sum of possible expansion for the Kernel, for cache write backs, for incoming I/O, and for the operations associated with recovering physical space.

Tools useful for implementing the approaches and/or minimizing the amount of space required to be held in reserve include:

i) Interrupts with high priority. In particular, we will use the non-maskable interrupt, which is employed in Microsoft NT only for 'Blue Screen', but does find some additional use in Linux.

ii) Classing. This is a feature of the MXT architecture, where physical memory usage of a specified set of pages can be monitored.

iii) Flipped operations. This is appropriate for the current MXT design, where all I/O is done through the L3 cache. The pageop mechanism in MXT permits the memory controller to change the 'real' address of a page without modifying the physical locations of its contents. This permits

I/O operations to be done out of a small set of real addresses. The result is a limit on the number of L3 cache lines that would be written back, and thus a tighter limit on required memory reserves.

iv) Outlist (1). This is a data structure with a guaranteed small footprint which a) keeps track of pages which are unmapped and clean or unmodified, which can be erased given a need for more space, and b) is used to record which pages have been so erased. This avoids the potentially costly (in physical space requirements) operation of walking through and modifying page tables and other structures during space recovery. The content of the outlist are thus closely equivalent to standby or reclaimable pages. Once sufficient space has been recovered using the list, the normal OS data structures can be updated.

Of the dynamic methods, B2 may be easiest to incorporate into an operating system or computer architecture, while B3 is appears well suited for incorporation in a hypervisor.

The following is an outline of this paper. Section 2 describes some properties of the MXT and Intel architectures, and of some data structures in Microsoft's NT operating system. Also described are properties of the MXT compression algorithm (9), which are related to the question of how much change in required physical space can occur as a function of changes to the data. Section 3 contains a brief discussion of how the mechanism chosen to ensure forward progress affects the steady-state memory management. Section 4 describes dynamic address disabling, how it might be implemented in the NT operating system, and its guarantee of forward progress. Section 5 describes orthogonal paging, along with a proof of guaranteed forward progress. Section 6 considers applications (such as web accelerators) running in Kernel space. Here a combination of dynamic address disabling and orthogonal paging is discussed. Finally Section 7 contains some concluding remarks.

2. Background

We first consider some further details associated with the MXT system shown in Figure 1. At any time, the memory controller maintains a count of available free 256 B sectors. As mentioned above, the classing mechanism has the capability of maintaining a count of the sectors in use by each of a number of classes of pages, with these classes specified by the OS. The hardware supports two thresholds for available 256 B sectors. If this count falls below specified numbers, interrupts are triggered. The first we term an alert interrupt, the second a low memory (LM) interrupt. The priority of the interrupt is a parameter that can be set by the OS.

The MXT compression algorithm is a parallel variant or extension (9, 10) of the well-known LZ77 algorithm. Here a block of data (1K bytes) to be compressed in MXT is partitioned into four subblocks. Each is sent to a different compressor, which uses as part of it compression dictionary the data in all the subblocks, subject to constraints on decodability. Its compression performance is similar to that of LZ77, with a fourfold parallel speedup obtained at the cost of additional hardware complexity. From the point of view of this paper, however, a property of interest is what might be termed its compression stability: suppose a 64B L1 line is modified. What then is the effect on the compression of the entire 1K block? There is no tight upper bound known to the possible expansion, although it is known that such a change can cause an increase of at least 2 sectors or 512B in space requirements. Moreover, a property of the MXT memory organization (2) is that some 1K lines, if sufficiently compressible, are stored in the compression directory or CTT. The result is that a trivial upper bound for the additional memory required as the result of a 64B line modification is an additional four 256B sectors or 1K. This means that for example deleting a page which is fully compressible (i.e. does not require any sectors), might in principle require an additional

1K of storage for each entry in a page table or data structure that is modified as the result of this deletion. Deleting pages from memory so as to recover space thus needs to be done with some care.

A typical operating system (e.g. NT) partitions its virtual memory space into OS space and user space. In this paper, we will make some assumptions regarding accesses to these spaces.

> a) Assumption 1: Accesses to user space are interruptible, except DMA or I/O activity. More precisely, no access to user space, by the O/S or user code, can prevent paging activity. Access to the file system cache must be interruptible as well, for systems incorporating a web accelerator.
>
> b) Assumption 2: Any OS operation can be treated as time-independent. For most cases, this is true only within limits, due to phenomena such as I/O time-outs. If the OS is running under a hypervisor, this may be true in general.

Assumption 1 is necessary for the dynamic address disabling (DAD), where accesses to (at least some subset of) user space are blocked pending space recovery. Basically, this means that no OS thread holds locks on data structures associated with paging while accessing user space or the file system cache. In particular this implies that once the NMI is complete, OS threads can get locks that are needed for pageouts while keeping threads which access user space interrupted. Assumption 2 is a requirement for orthogonal paging, which, while essentially invisible to the OS, may cause time lags in operations.

Some applications do not conform to the above assumptions. Examples are some web accelerators, which reside, along with large amounts of data, in kernel space. Access to this space may be needed to do I/O, so that say dynamic-address disabling is not applicable. However, a combination of the techniques discussed in the introduction can be appropriate.

We now consider the addressing structure of NT (essentially a feature of IA32), and the data structures which may be affected by paging.

Figure 2 shows a version of the addressing structure which is sufficient to illustrate the techniques under discussion. Here each process has a 4K page which contains *page directory entries* or PDEs. Each entry in this page points to a page which holds *page table entries* or PTEs. The PTEs in turn point to individual pages. Each PTE or PDE describes certain attributes associated with the page it points to.

For example, the page could be valid or invalid, read-only or read-write etc. Given an access to a page which currently is not in the processor table look-aside buffer or TLB, the processor successively reads the PDE and the PTE associated with this page, obtaining the information regarding the page's real address. If any page in this sequence is marked as invalid, an interrupt hands control over software which for example may page in the requested page.

The pages holding PDEs and PTEs act as paths for processor addressing. However, they are also data from the point of view of the OS. Thus the OS could mark certain PDEs as invalid for purposes of processor access, while elsewhere noting their correct status.

Other data structures affected by paging activity include the *page frame data base* which records attributes of pages, such as their presence on say free lists, data structures associated with keeping track of working set memberships, and data entries noting page sharing between processes.

This tally of the various data structures associated with the presence of a page in memory, coupled with the lack of a tight upper bound on compression expansion means that in principle the deletion of a page from say a working set may require changes to three or more L1 cache lines, each residing in a different L3 cache line. The above observations on compression stability indicate that as many as 3KB of expansion may be produced by a pageout operation.

Another issue of interest is the functioning of the I/O adapters. At any time, each adapter may have a queue of commands. Further, processing of the queue may require interactions with the OS, and in general there is no preemption possible. This property constrains approaches such as orthogonal paging.

3. Space Management

The technique chosen for avoiding system hang-ups due to running out of physical memory is a basis for the structure of the virtual memory manager. In each case there is an added level of complexity as compared to today's systems, which attempt to maintain an adequate number of available page frames. In a compressed memory system such as one with MXT architecture, page frames correspond to *real addresses*, or entries in the compression directory. This directory is usually of fixed size, so that such entries are a limited resource. This resource corresponds closely to today's list of page frames. Since in a compressed-memory system the number of real addresses exceeds the corresponding nominal physical space, there is the additional dimension of available physical space. Even in a A1 or safe system, there is the problem of ensuring that there is sufficient space in the part of memory holding compressed data so as to maintain efficient operation.

Thus all compressed memory systems of the types we consider require keeping some amount of free or readily reclaimable physical memory. In an A1 system the reclaimable memory can be simply an ordinary reclaim list, as all data structures modified due to a pageout are assumed to be in the ordinary or uncompressed part of memory. In an A2 type system, much of M-space may hold compressed data. Physical space must be available for expansion of this data. However, as noted above, a large part of this space may hold pages on an Outlist. This means that a system of type A2 can be more space efficient than an A1. Further, if sufficient space is maintained it is shown in a recent patent application (7) that it is not necessary to do an immediate cache invalidate when a page is removed from M-space. This is because the number of modified lines held in the cache belonging to pages removed from M-space is limited. However, care must be taken regarding the amount of free space to be maintained, as any interrogation of the memory controller to determine the number of free sectors may be out of date before the OS can act on the information. The above mentioned application (7) also provides an inequality for the amount of free storage which guarantees sufficient space despite this problem. We modify this slightly to obtain a requirement for entering a new page into M-space:

$$G > (C+M-B)+Q+4kB \qquad [1]$$

where G is the measured amount of free space at time t, and the other quantities represent:

C is the size of the L3 cache (whose contents include those of L1 and L2)

M is the nominal (uncompressed) size of the M membership at this time.

B is the actual amount of physical space occupied by the M membership.

Q is the space required for pending I/O.

4kB represents the size of the page to be added to M.

The above inequality represents that current cache contents, plus I/O, plus the current page, cannot take up more space than G, even with delays in measuring this quantity.

Consider an A1 or an A2 type system. Here, as mentioned above, the uncompressed (A1), or read/write (A2) pages are assumed to hold those parts of the OS which are required for paging.

That is, no new pages need be added to these spaces in order to delete or pageout others. In general, the required pages are identified by ranges of virtual addresses. Access to pages outside these ranges are then not needed for the recovery of space. User pinned pages will not be removed from memory, but generally need not be mapped or accessible for space recovery.

In a DAD system, the low memory interrupt disables access to pages outside ranges required for paging. An advantage here is that the system operates similarly to one without compression, except in the unusual case of low memory.

In an A1 or A2 system, some mechanism needs be present to determine the set of pages that are accessible. This could be the normal working set mechanism, but this is probably too inefficient. Instead, the usual OS memory management software could determine the set of pages to be paged out given a need for more space. Membership in the accessible set is probably best determined via a separate facility with a short path length. An example might be a FIFO stack, where a page is entered at the head when access is granted, and eventually aged out. This is less efficient from a 'page fault' point of view than an LRU mechanism, but requires no updates until a page is to be removed. An MXT A2 system might then provide read-only access to all pages in memory, and read-write access to pages on this

FIFO list. A wrinkle incorporated in a recent design (11) for a follow-on to MXT maintains a FIFO stack of uncompressed L3 cache lines. A line about to be removed is placed on the top of this stack if it has been recently referenced (as determined by a cache-like structure). A similar mechanism could be used to improve the hit ratio to the mapped pages here.

In all these systems, where pages on a reclaim list are held in compressed form, the amount of free or readily available space needs to be estimated, or alternatively measured via the classing mechanism discussed above. This is somewhat more complicated if I/O is done through the L3 cache (as in one version of MXT), or where substantial numbers of pages are allocated before their contents are inserted. Here there is the possibility of what in (1, 12) is termed *allocated but unused storage*, where the physical space which will be occupied by these pages is counted as free, while these pages are counted as part of the number of allocated. The result is an apparent but spurious or temporary measured improvement in compressibility, which could potentially cause some instability. (Typically for I/O, there is a max. of 64mb per device, due to DMA addressing limitations.) If free plus readily reclaimable (e.g. on an Outlist, plus reclaim list) is monitored, or if interrupts are set at various levels, the OS can take action before the scarcity of space becomes critical. Pageouts can begin, and/or user processes suspended etc. However, the amount of free space required may be underestimated, or high priority tasks may consume free space. Thus, for dynamically controlled systems, it may be desirable to ensure that physical memory cannot be exhausted, and the system has guaranteed forward progress.

The above discussion has ignored the mechanics of I/O. In an A1 system, this could simply be done into the uncompressed area. Alternatively, in A1, and necessarily for the others, space needs to be set aside for incoming pages. The way this is done is system dependent. For example, in Microsoft NT, a page is pinned before it is written out, but also before I/O into its page frame. The request to the OS for the pinning operation would need to distinguish between these two alternatives. For incoming I/O, the physical storage thresholds which control paging could be adjusted, and such pinning could be denied under certain conditions, causing an interrupt. We discuss this operation further below in the context of a B2 or DAD system.

The following sections respectively discuss orthogonal paging and dynamic address disabling. As mentioned above, these techniques provide bounds on the amount of physical reserves required for space recovery.

4. Dynamic Address Disabling

In some systems, given a low-memory interrupt, it may be feasible to stop all writes (except for ongoing I/O) into user space. For example in NT, an I/O driver is required to pin a page before writing into it. It also needs to pin a page before writing it out. If the former type of pinning is stopped by the OS, then space utilization due to adapter actions is blocked. If one then reserves sufficient space for kernel expansion, plus ongoing I/O, one then has GFP (1). However, in general it may not be feasible or easy to suspend kernel or user processes writing into user space. In an M-Space system, this problem is handled by keeping a large set of pages not critical to the pageout process unmapped or in read-only status, with the disadvantage of loss of efficiency due to mapping and unmapping overhead. In this section, we discuss the alternative of essentially instantaneous unmapping of nonessential pages.

Figure 2 shows the addressing path for an IA32 processor. Given a TLB miss, the processor references the *page directory* for the process. which is a page with 1024 entries, each a PDE (Page Directory Entry) pointing to a *page table*. Each page table page has 1024 entries, each a PTE. A PTE points to a page frame (or in MXT a real address) allocated to a process or to a kernel page. The point here is that a currently active process on a processor has a single page which forms the root of its address tree. Invalidating all PDEs on this page which point to user space, and invalidating the TLB contents, ensures that attempts to access user space will result in interrupts. Under assumptions *a* and *b* of Section 2, the associated interrupted processes can then be suspended until sufficient physical space is recovered.

The above process of address disabling B is well suited to the IA32 architecture. Other systems have different addressing paths. However, a means is generally implemented for efficient switching between different user address spaces. For example, in PowerPC, addresses generated by program code (termed 'effective addresses' in PowerPC parlance), are prefixed by segment ID's to form 'virtual addresses' which are then translated into real addresses via the TLB. Address disabling here could be done on the segment level.

We now return to the IA32 case. The system would maintain (possibly in an Outlist) sufficient reserve space for I/O, cache and kernel expansion. Given the crossing of the T0 threshold, an NMI would be triggered. This would cause the disabling of the PDE's of the currently active process(es) pointing to user address spaces. Any process that becomes active would have its appropriate PDEs disabled. Appendix A provides a high level view of the program logic to carry out this operation. Two examples are given: the straightforward case where two spare bits in the PDEs are used for this, and an alternative with shadow PDEs.

Once the relevant addresses are disabled, control is returned to the OS which is, however, placed in what we term the E-recovery state. It can then proceed to reduce working sets, do pageouts etc. References to pages whose address paths have been disabled result in interrupts. In Windows NT, process PTEs are in kernel space, so that user address disabling does not affect pageout operations. Once sufficient space has been recovered, the OS is returned to its normal state. This involves re-enabling the PDEs, just prior to context-switch.

5. Orthogonal Paging

The basic idea here is that, given a low memory condition, special software, with low memory usage requirements, takes over the process of pageouts. After sufficient space has been recovered, control is returned to the OS. It is especially suited to cases where the OS runs under control of a hypervisor which actually does all I/O. Otherwise, an I/O adapter may need to be reserved, or a means provided for interrupting or preempting current ongoing I/O.

A special case is where orthogonal paging is done in conjunction with dynamic address disabling, as discussed in the following section. Here the OS is in control during the orthogonal paging operation, so no special I/O adapter is required.

A possible implementation of orthogonal paging as a modification to NT, is as follows. The system maintains a list (which we term the *Paging list or PL)* of pages that are not pinned or shared. Pinned pages cannot be written out, and shared pages (at least in Windows NT) are associated with data structures (Prototype PTEs) that are awkward to change. This awkwardness is due to the property that there is no backward pointer from the page frame data base to the PTEs for this page in the sharing processes. Pages not on the PL, if fully uncompressed, must fit in memory. That is, the amount of physical memory in the system, minus that taken for the CTT, is required to be at least as great as the number of such pages, times 4K bytes per page.

The PL could be viewed as having a subset which is an Outlist. That is, a subset of the paging list which contains pages which are unmodified, and on the reclaim list, so that they can be simply erased, and which in total occupy a required amount of space. The following is an overview of the operation. (If the OS runs under the control of a hypervisor, operation is somewhat simplified, as will be described below).

a) Stop all operations via a non-maskable interrupt.

b) By various operations described below, recover sufficient space by paging to a special swap area of disk some number of pages. Pages so written out are marked as 'stolen' or invalid (the stolen bit is an additional entry in the PTE).

c) When the OS is restarted, references to the stolen pages cause an interrupt. The OS then fetches the stolen page from the special area.

Note that membership in working sets in not changed. Pages on the Outlist are reclaim pages, so that no PTEs are affected. The above is an outline. A description is required for the data structures used, and on the sequence of operations required so as to have GFP. First, the structure of the PL needs to be consistent across the NMI. That is, if the NMI occurs in the middle of on update to this data structure, no data is lost, and the list is eventually updated correctly. Appendix A describes a structure with these properties. The trick used combines an atomic operation with a technique for bypassing the item being updated.

We now consider the sequence of operations. Once the NMI occurs, pageouts and or page deletions need be done. The NMI would occur when free physical space falls below a given threshold , say L0, space would be recovered until a threshold L2 is exceeded, with say L1 the level at which the OS first starts to react to a space shortage.

Pageouts would be done by a program triggered from the NMI. A page on the reclaim or Outlist would simply be erased. In the current version of MXT, I/O is done through the L3 cache. This means that potentially 4K of cache contents might be written back into memory, with up to 4K of potential expansion for each pageout. One way of handling this is to reserve sufficient space for such I/O, essentially the size of L3. An alternative is to use the flipping technique mentioned in the Introduction. Here the real address of the page to be written out is changed prior to the pageout. This results in the invalidation of a TLB entry for this page, as well as the write back of all cache lines from this page.

The real address of the page is changed, and the page is then written out. The advantage is that a few real addresses can be used for such pageouts, thus avoiding much cache damage. The amount of reserve space needed is then reduced to the size of the special paging program, plus the number of real addresses used for such pageouts, plus the number of pages in the paging program that are subject to change (i.e. not read-only). It should be noticed that the amount of space that is recaptured by removing a page from memory is not known in advance. Thus in particular, there are cases where no space is recovered. Further, each change to a PTE or other data structure might in principle cause an expansion of 1K in physical memory requirements. This means that updating of PTEs and other OS structures must be separated from paging. The procedure can then be done as follows:

i) Trigger an NMI when the amount of free space falls below L0.

ii) Remove (by erasing or writing out) enough pages on the paging list so as to recover some desired number of megabytes of physical memory. (The choice of this number affects efficiency, but not the guarantee of forward progress.)

iii) Update the PTEs and Page Frame Database entries of the paged out pages. If during this process the level of physical memory falls below L0, return to (ii).

iv) Repeat the above until either (a) the amount of free physical space is increased to more than T2, or (b) alternatively until the number of pages contained in memory, times 4K bytes, is equal to the memory size minus the size of the CTT. Note that if the condition reached is (b), it is guaranteed that the updates in (iii) can be done.

v) Restart the OS.

Separating the updates of OS data structures from the paging of individual pages ensures that the process can continue until either sufficient space is recovered, or the nominal size of the memory contents is no greater than the physical size. Thus there is GFP.

Pageouts after the NMI in a normal OS mean that certain I/O operations need to be preempted. This either requires that the I/O adapters have this capability, or that there is an adapter and disk dedicated to this purpose. This problem does not arise in a system with a hypervisor, which handles all I/O, and could handle structures such as the PL.

Above did not include use of an Outlist, as this could be regarded as part of the free space. In the above, as the PL processing causes expansion, additional members of the Outlist would be erased. After the OS is restarted, the Outlist would be reconstructed.

6. Combined Approaches

In the previous sections, it was generally assumed (except for orthogonal paging) that metadata required for paging could be separated from the bulk of the memory contents. An example of applications where this is not true are web cache accelerators. Here a possibly very large number of pages is stored in the system cache, which is in kernel space. The file system metadata is generally interspersed throughout this space.

Approaches to guaranteed forward progress such as the paging store (A1), or M-space, are dependent on restricting access to the bulk of the memory contents. If metadata cannot be easily separated from the data, these approaches are impractical. Similarly, dynamic address disabling, which would in this case block access to the system cache, is unworkable due to lack of necessary access to the metadata. Orthogonal paging could be used, but is awkward due to the above-mentioned requirement on interruptible or reserved I/O adapters.

The above difficulties can however be addressed by a combination of orthogonal paging and dynamic address disabling. This could work by first disabling access to the system cache. Once this is done (with adequate space reserves for expansion of the remainder of the kernel, and ongoing I/O), the operating system can be restarted, and some contents of the system cache paged out orthogonally, so as to recover space. Once some desired level of space is recovered, access to the system cache could be restored, with orthogonally paged out pages marked as “stolen”, following the orthogonal paging approach described above.

Some OSs have a paging system that is independent of the file system. Independent paging systems could still operate even if access to the file system cache were blocked. However, during space recovery it still may be necessary to page out memory-mapped pages that otherwise would be written back home.

7. Conclusion

In today’s systems, if the number of available page frames declines below a given threshold, the OS can stop the allocation of additional pages. This, coupled with the restriction that all data structures and programs required for paging are resident in memory, ensures that no additional page frames will be required during the process of deleting pages from memory. The result is that the system will not run out of page frames required for paging.

Ensuring a similar property for a compressed memory system requires in addition that the system will also not run out of physical space. In this report, we consider techniques for insuring this property. Such techniques fall into two main categories which we term respectively *safety* and *dynamics* . A safe system operates in a way analogous to today’s. Allocated physical space is controlled so there is no danger of a space shortage. In a dynamic system, a space shortage can occur unless the OS must takes actions such as suspending user processes, or blocking access to all but a subset of the address space.

In this report, we discuss various approaches within these two classes, including two new techniques, which we term orthogonal paging (OP), and dynamic address disabling (DAD). Dynamic address disabling may be easier to implement in NT-like operating systems, while orthogonal paging appears well suited to hypervisors.

The discussion is largely centered on systems where applications run in user space. For exceptions such as web accelerators, both OP and DAD have implementational difficulties. Here a combination of the two, as discussed in Section 6, may prove attractive.

Appendix A: Dynamic Address Disabling

We discuss two approaches to implementing dynamic address disabling. The first uses two spare bits in the PDEs. This is easier to explain, but is probably less practical than the second approach. In either approach, the OS is prevented from allocating additional pages.

1) *Spare bits.* Given an NMI indicating a low physical memory condition, the 'page directory' for the currently active process has all its non-kernel space entries, PDEs, marked as invalid. One spare bit indicates the actual status of the page (i.e. valid or invalid), the other the OS status: either space recovery or normal processing. The result is that access to the private space of a process triggers an interrupt, but status information can be entered in PDEs. Once space recovery is complete, the spare bits in the currently active processes are reset. PDEs are updated atomically, for example using 'compare-and-swap'.

2*) Shadow.* Here no spare bits are required in PDEs. Instead, the system maintains shadow copies of the page directories. The shadow directories are the same as the originals, except the PDEs mapping user space are marked 'invalid'. In space recovery mode, the shadow page directories are used in place of the original page directories - blocking addressability of user space. The switch to shadow directories occurs initially within the NMI handler, and subsequently at context switches. After space recovery, context switches are resumed with the original directories and accessibility to user space returns.

A variation would be to create the shadow PDEs only when necessary - when the NMI occurs, a shadow would be built for the current process, and subsequently context switch would build shadows as long as the system remained in space recovery mode. In this case, the shadows would be used to save the original page descriptor. PDEs addressing user space would then be marked invalid or read only. Updates to the PDEs would be made to the shadows while the system remained in space recovery mode. This scheme would need a way to 'detour' updating a PDE from the NMI if memory management were updating the same PDE. With this scheme, the shadow does not need to be updated every time the principal is.

Figures (3, 4) illustrate the data structures involved. There is a status data structure reserved for indicating the current mode of operation for the system and a per-process entry for the address of its shadow page directory. Given an NMI the shadow directory is made active for the current process. Subsequent context switches establish addressability using shadow directories as long as the system remains in space recovery mode. Once space recovery is complete, the status data structure is returned to normal operation and context switches will revert to the original page directories, reestablishing user addressability.

Appendix B: Orthogonal Paging

We now describe data structures used for orthogonal paging. There is a value Tfree denoting the total amount of free space. A list PL is maintained of pages that are not pinned or shared. The total number of pages in the system is denoted by Nt, the number on this list by Nl. During normal operation, it is required that (Nt-Nl)*4K be less than the size of memory minus the size of the CTT so that removing (either by erasing or writing out) the pages PL guarantees that the remainder will fit. The CTT in turn, has 64B per page, and its size, in the current MXT design, is set so that the number of real addresses is twice the nominal physical space. Thus say four megs of physical space translate to eight megs of 'real space'. The size of the CTT in the current design cannot be changed, but this may not be the case in future releases. The amount of space occupied by items on PL is denoted by LFree.

Operations on L include:

A) Operations during normal processing:

i) Adding an item to PL, updating LFree.

ii) Deleting an item from PL, updating LFree.

B) Operations during space recovery. This starts upon the processor receiving an NMI. It ends when sufficient space has been recovered so that the OS can be restarted. If part of PL comprises an outlist, the operation might start by erasing a sufficient number of pages on this list to recover space for the other operations. Outlist entries in PL could be linked by pointers, and the total amount of space covered denoted by OFree.

The procedure is as follows:

i) Delete items from the Outlist until sufficient free space is obtained. This space is as mentioned above the size of the orthogonal paging program, plus the pages that will be modified (i.e. data areas) plus the area used for I/O (via the flipping operation), plus sufficient space to cover cache castouts.

ii) Erase, or write out enough pages so that so as to recover some amount delta of free space, delta typically being guaranteed enough so as to update PTEs as well as other data structures for say N(delta) pages. Each update requires no more than 1K bytes of extra space. After the updates, the amount of free space TFree is checked. Additional updates made, or alternatively additional pages written out. The result is that either the required amount of space is freed, or the number of pages reduced to where they are guaranteed to fit in memory without respect to their compression ratio.

iii) Return from the NMI. At this point, pages which have been written out are still potentially members of process working sets. Attempts to access them result in page faults, where the page fault handler is aware that they have been written out by the orthogonal pager. The orthogonal pager might typically write out pages to a swap area on disk, so as to minimize the amount of data lookup which would otherwise be required to determine an appropriate disk address.

Figure (5) illustrates the data structure for PL. It is a doubly linked list, with one entry per real address, so that an entry can be found via direct mapping. Each entry indicates whether a page is on the Outlist (these could also be linked). In order to preserve correctness given an NMI, a detour is used. Here the item about to be changed or updated is entered into a Detour field via an atomic operation (i.e. a compare with zeros and a swap with the real address). The NMI handler would examine the Detour fields, and bypass the related items on the list. That is, the entry with the corresponding real address, as well as those addresses linked to this one. The result is that an extra three pages of space need to be reserved.   Once the NMI processing is complete, the interrupted program can continue with its update.

Acknowledgments:

The authors gratefully acknowledge valuable discussions with M. Auslander, M. Hack, P. Heidelberger, C. Schulz, and T.B. Smith.

References:

1. P.A. Franaszek, P. Heidelberger, D.E. Poff, and J.D. Robinson, " Algorithms and data structures for compressed-memory machines", IBM J.R.&D, Vol.45, No.2, pp.245-258, March 2001.

2. R.B. Tremaine, P.A. Franaszek, J.T.Robinson, C.O. Schulz, T.B. Smith, M.E.Wazlowski, and P.M. Bland, "IBM Memory Expansion Technology (MXT)," IBM J.R.&D., Vol.45, No. 2, pp.271-286, March 2001.

3. M. Kjelso, M. Gooch, and S. Jones, " Performance evaluation of computer architectures with main memory compression", J. Syst. Arch. 45, 571-590, 1999.

4. J.-S. Lee, W.-K. Hong, and S.-D. Kim, " Design and evaluation of a selective compressed memory system." Proceedings of the International Conference on Computer Design, IEEE, 1999, pp. 184-191.

5. F. Douglis, " The compression cache: using on line compression to extend physical memory", Proceedings of the Winter 1993 USENIX Conference, USENIX Association, San Diego, 1993, pp. 519-529.

6. W.P. Hovis, K.H. Hazelhorst, S.W. Kerchberger, J.D. Brown, and D.A. Luick, " Compression architecture for system memory applications," U.S. Patent 5,812,817, Sept.22, 1998.

7. P.A.Franaszek, M. Hack, C.S. Schulz, and T.B. Smith, " Space management in compressed main memory," IBM patent application, August 1996.

8. B. Abali, H. Franke, D. E. Poff, R. A. Saccone, Jr., C. O. Schulz, L. M. Herger, and T. B. Smith, "Memory Expansion Technology (MXT): Software support and performance", IBM J.R.&D., Vol. 45, No. 2, pp. 287 - 302, March 2001.

9. P.A. Franaszek, J. Robinson, and J. Thomas, " Parallel compression with cooperative dictionary construction," Proceedings of the DCC 1996 Data Compression Conference, IEEE 1996, pp.200-209.

10. P.A. Franaszek, J. Robinson, and J. Thomas, " Parallel compression and decompression using a cooperative dictionary," U.S. Patent 5,729,228, March 17, 1998.

11. C.D. Benveniste, P.A. Franaszek and J.T. Robinson, "Cache-Memory Interfaces in Compressed Memory Systems," IEEE Transactions on Computers, Vol. 50, No. 11, November 2001

12. P.A. Franaszek, P. Heidelberger, and M. Wazlowski, "On Management of Free Space in Compressed Memory Systems," Proc. International Conference on Measurement and Modelling of Computer Systems, pp. 113 - 121, 1999

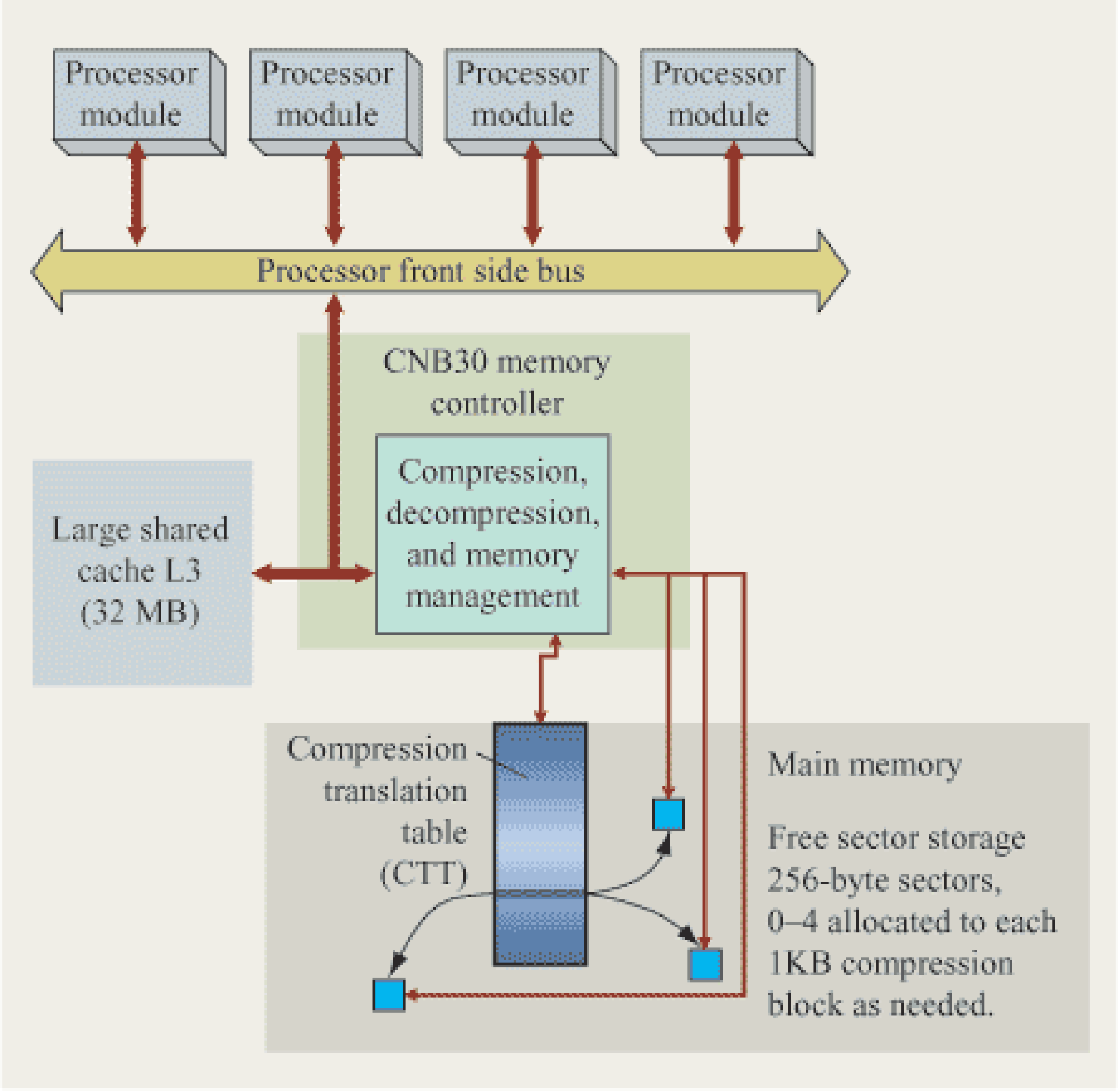
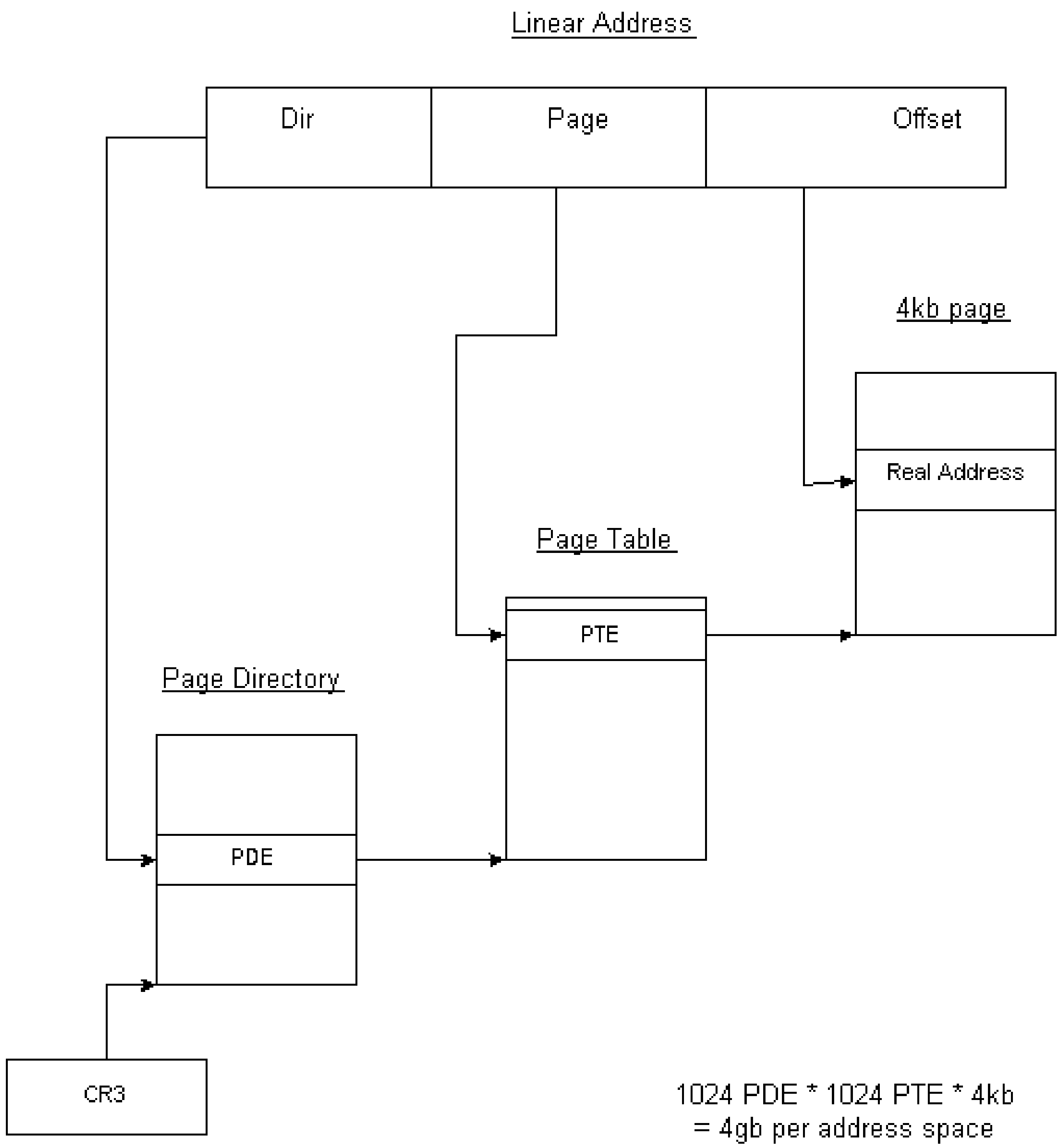

Processor module
Processor module
Processor module
Processor module
Processor front side bus
CNB30 memory controller
Compression, decompression, and memory management
Large shared cache L3 (32 MB)
Compression translation table (CTT)
Main memory
Free sector storage 256-byte sectors, 0–4 allocated to each 1KB compression block as needed.

Figure 1 (IBM, JRD, vol 45, no 2, 2001: MXT Software Support and Performance)

Overview of a system with memory expansion technology (MXT).

IA32 Address Translation



Intel's IA-32 Software Developer's Manual, Volume 3: System Programming Guide

Figure 2

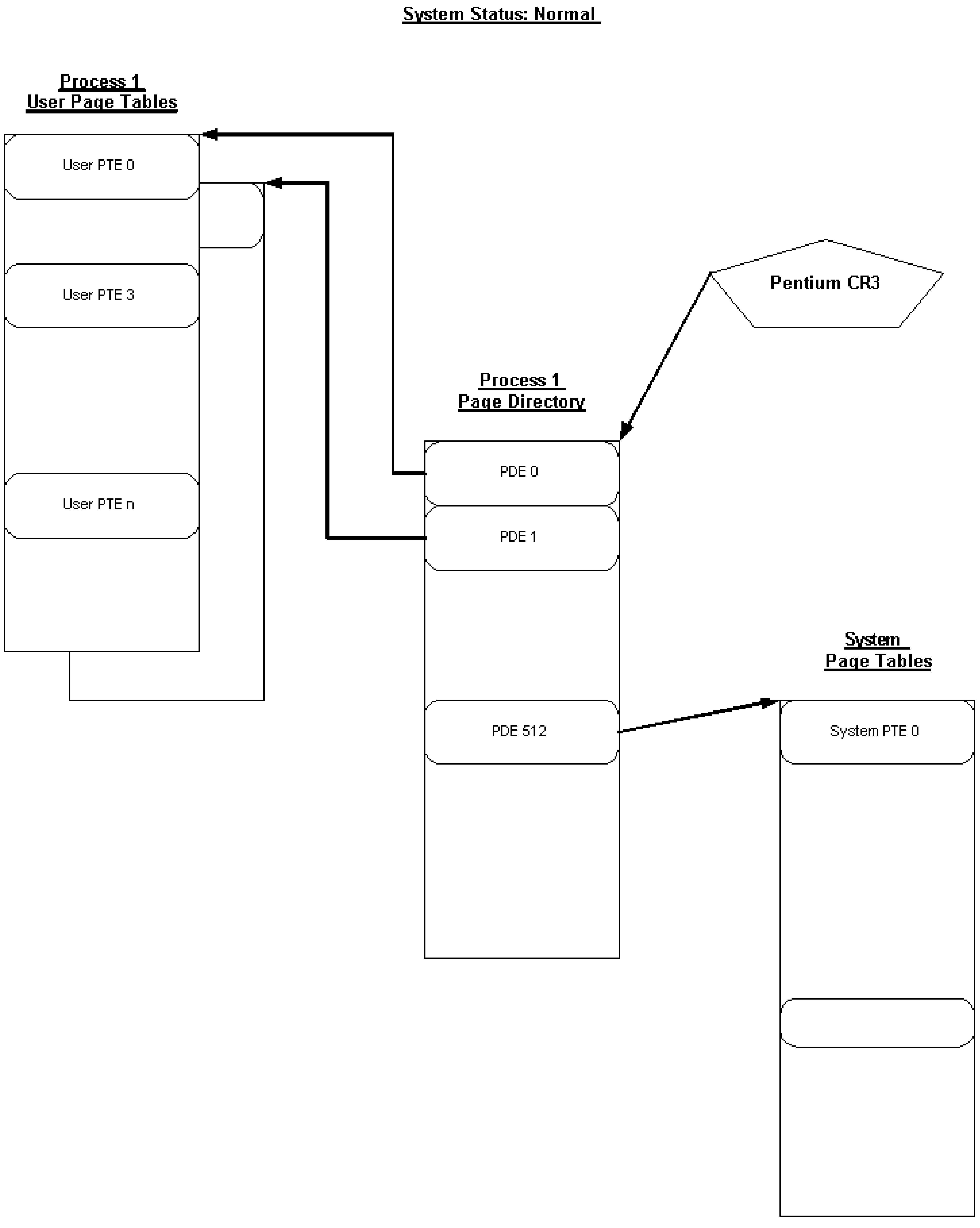

System Status: Normal
Process 1
User Page Tables
User PTE 0
User PTE 3
User PTE n
Pentium CR3
Process 1
Page Directory
PDE 0
PDE 1
PDE 512
System
Page Tables
System PTE 0

Figure 3

System Status: E-recovery

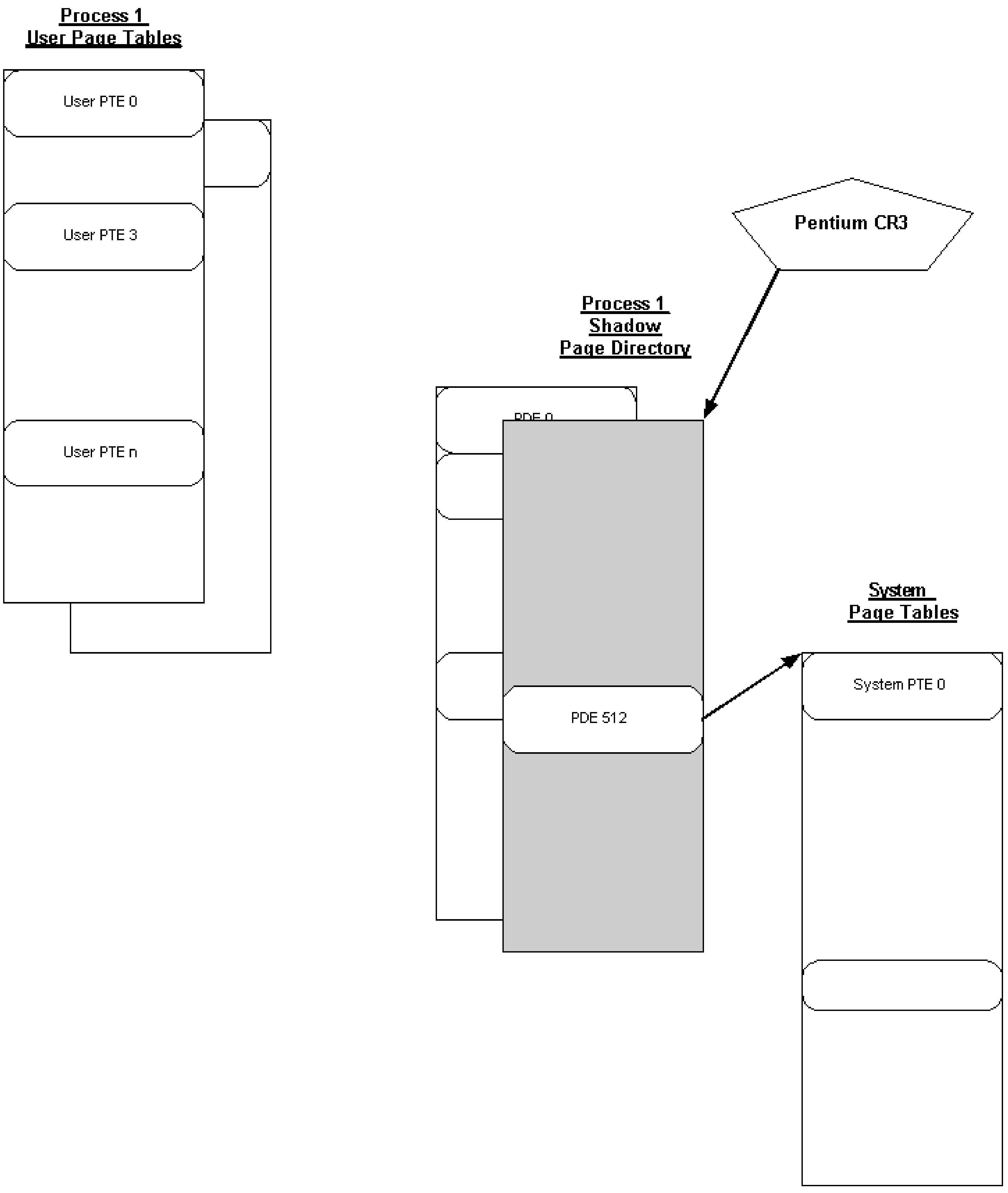


Figure 4

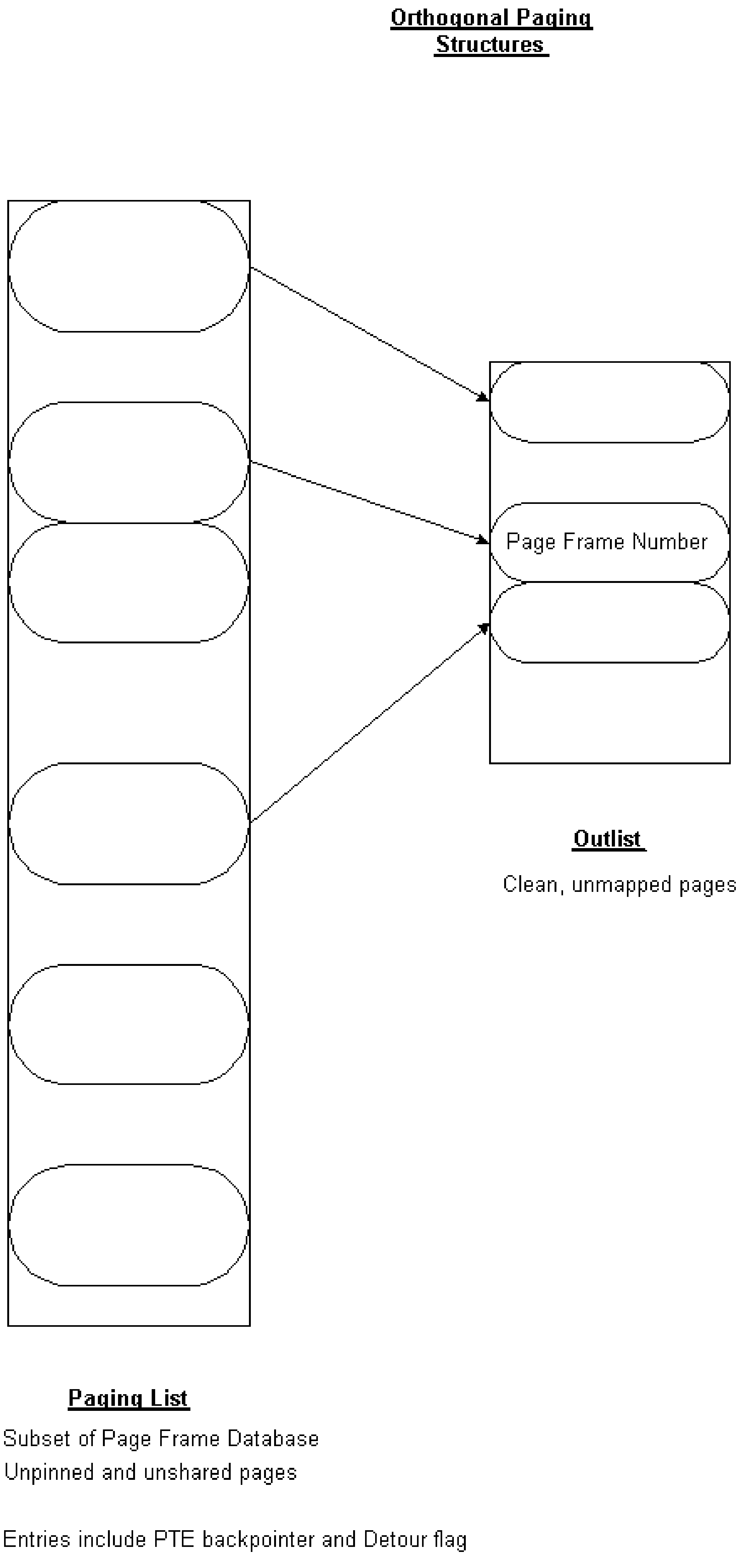

Orthogonal Paging Structures
Page Frame Number
Outlist
Clean, unmapped pages
Paging List
Subset of Page Frame Database
Unpinned and unshared pages
Entries include PTE backpointer and Detour flag

Figure 5