An Evaluation of the Linux Virtual Memory Manager to Determine Suitability for Runtime Variation of Memory

by

Vijay Kumar Muthukumaraswamy Sivakumar

Thesis submitted to the faculty of the Virginia Polytechnic Institute and State University in partial fulfillment of the requirements for the degree of

Master of Science

In

Computer Science and Applications

Dr. Godmar Back

Dr. Calvin J. Ribbens

Dr. Kirk W. Cameron

February 2, 2007

Blacksburg, VA

Keywords: virtual memory management, runtime variation of memory, application performance

An Evaluation of the Linux Virtual Memory Manager to Determine Suitability for Runtime Variation of Memory

Vijay Kumar Muthukumaraswamy Sivakumar

Abstract

Systems that support virtual memory virtualize the available physical memory such that the applications running on them operate under the assumption that these systems have a larger amount of memory available than is actually present. The memory managers of these systems manage the virtual and the physical address spaces and are responsible for converting the virtual addresses used by the applications to the physical addresses used by the hardware. The memory managers assume that the amount of physical memory is constant and does not change during their period of operation. Some operating scenarios however, such as the power conservation mechanisms and virtual machine monitors, require the ability to vary the physical memory available at runtime, thereby making invalid the assumptions made by these memory managers.

In this work we evaluate the suitability of the Linux Memory Manager, which assumes that the available physical memory is constant, for the purposes of varying the memory at run time. We have implemented an infrastructure over the Linux 2.6.11 kernel that enables the user to vary the physical memory available to the system. The available physical memory is logically divided into banks and each bank can be turned on or off independent of the others, using the new system calls we have added to the kernel. Apart from adding support for the new system calls, other changes had to be made to the Linux memory manager to support the runtime variation of memory. To evaluate the suitability for varying memory we have performed experiments with varying memory sizes on both the modified and the unmodified kernels. We have observed that the design of the

existing memory manager is not well suited to support the runtime variation of memory; we provide suggestions to make it better suited for such purposes.

Even though applications running on systems that support virtual memory do not use the physical memory directly and are not aware of the physical addresses they use, the amount of physical memory available for use affects the performance of the applications. The results of our experiments have helped us study the influence the amount of physical memory available for use has on the performance of various types of applications. These results can be used in scenarios requiring the ability to vary the memory at runtime to do so with least degradation in the application performance.

Table of Contents

Table of Figures

1. Introduction

All contemporary operating systems support the virtual memory technique, which allows processes to address more memory than is available in the system. This technique is made possible by the use of virtual addressing and demand paging. The available physical memory is virtualized by the use of virtual addresses. Page tables exist to convert these virtual addresses into actual physical addresses. Processes use the virtual addresses and thus operate under the assumption that a large amount of physical memory is available for them to use. The processes are not aware of the actual physical addresses that they use. Demand paging allows data to be brought into memory only when it is accessed. When the pressure on the physical memory is high, data not currently in use is swapped to disk, thereby relieving memory pressure.

The memory managers of the systems supporting virtual memory manage both the virtual address space of the processes and the physical address space. They keep track of the virtual addresses used by the processes and they also remember what purposes various virtual addresses are being used for. They manage the allocation and freeing of physical memory and are responsible for the reclamation of physical memory when the system runs low on memory. They also maintain the mapping between the physical memory and the virtual memory of the processes.

Memory managers work under the assumption that the amount of physical memory available in the system is constant throughout the period of their operation. This is not a valid assumption to make in all the scenarios the system might operate in. Some of the scenarios where this assumption does not hold true are:

- Power Conservation: Memory consumes a significant amount of power in high performance machines and is thus a potential candidate to be tuned for power conservation [1, 2]. By turning off memory that is in excess of what is needed power can be conserved. This technique can also be applied to mobile devices that are powered by batteries, so that the duration of operation can be increased [3, 4].

- Virtual Machines: Virtual machine monitors are capable of running more than one Operating System at the same time. Depending on the load experienced by these guest operating systems, the underlying monitor might have to reclaim memory from one operating system for use by another [5, 6].
- Hot Pluggable Memory: It supports addition and removal of RAM modules from the system at runtime [7, 8]. Servers might sometime need more physical memory than what they were started with or some defective module might have to be removed or replaced without having to power the machine off.

In this work we studied the suitability of the existing memory management techniques for scenarios like those mentioned above where the available physical memory varies with time. We have implemented an infrastructure that allows the users to vary the memory at run time using system calls. The available physical memory is divided into logical banks, where each bank is an atomic unit that can be turned on or off independent of the others. These banks can be turned on or off using system calls that were added to the Linux kernel 2.6.11. Apart from adding support for these new system calls, the memory manager was modified as necessary to support the runtime variation of memory.

Due to the lack of hardware support, the banks are not physically turned off or on by the system calls. Instead, when a bank is turned off, the pages belonging to that bank are logically removed from the pool of available memory so that they cannot be used by the system. When a bank is turned on, the pages belonging to the bank are reintroduced into the pool, thereby making them available for use by the system. Though the current memory technologies do not have any support for turning parts of the memory off, the yet to be introduced Fully Buffered DIMM technology [9] supports this feature. The only restriction imposed by this technology to turn off memory modules is that the modules farthest from the controller have to be turned off before any module closer to the controller could be turned off. Since this is the only technology that is known to support dynamic resizing of the memory, our experiments were performed in a manner so as to

simulate performance on this technology. Thus when we turned off banks, we turn off a bank with a larger index before a bank with the smaller index.

We have performed experiments with varying memory requirements and studied the impact the available physical memory has on the performance of the applications. We have compared the performance obtained on a machine that varies memory at run time to the performance obtained on a machine started up with an equal amount of memory. We have observed that the performance obtained on the system by varying the amount of physical memory at runtime is lower than the performance obtained in a system started up with an equal amount of memory. This suggests that the current design of the Linux memory manager, without changes, is not well suited for the purposes of dynamically varying the physical memory.

The results of our study can be used to understand advantages and drawbacks of using the existing virtual memory management techniques for dynamically varying the physical memory and to identify the overhead incurred when these memory managers are modified to support runtime variation of memory. This in turn would lead to a better design of the memory managers to support such features. The results are also expected to benefit some of the applications that require the ability to vary memory at run time. For instance, our results might be used by virtual machine monitors in order to make decisions about resizing the physical memory space available to the operating systems running over it in such a way that the application performances are affected the least.

Based on the results of our experiments, we have identified the limitations of our implementation. We were not able to turn off banks that contained certain kinds of pages. The important factors that hindered our ability to reduce memory are the frequency of usage of a page and the length of the term for which it is used. We have identified a way to extend our infrastructure to handle such pages.

With the increase in the demand for the ability to vary the memory at runtime, the memory managers are expected to support this feature in the future. In this work we have

implemented an infrastructure over Linux that enables the user to vary the memory at runtime. Using this infrastructure, we evaluated the suitability of the Linux memory manager for the purpose of dynamically varying the memory. We have also performed experiments to study the impact memory has on the performance of applications and the results of these experiments can be used by applications requiring the ability to vary memory to do so with least degradation in performance.

2. Memory Management in Linux

Linux is one of the widely used operating system that supports virtual memory. The source code is freely available and thus it was an ideal choice of an operating system to build our infrastructure over. This section describes in brief the design and working of the parts of the Linux Memory Manager that are affected by our implementation. We implemented our infrastructure on the x86 platform and hence some parts of the following discussion are specific to this architecture. Detailed information about the 2.6 Linux kernels can be found in [10], whereas [11] concentrates on just the memory management system of Linux.

2.1. Physical Memory Management

The memory manager is responsible for abstracting the physical memory and managing the allocation, deallocation, and reclamation of the physical memory. The physical memory is divided into page frames of a fixed size (4096 bytes for the x86 architecture). It is at the granularity of a page (i.e., the page frame) that most of the operations related to the physical memory are carried out.

2.1.1. Physical Memory Hierarchy

Each page in the system is represented by an instance of *struct page*. This structure maintains all the information about the page. The *mem_map* array contains the *struct page*s of all the page frames in the system. Some of the important members of the *struct page* are:

- *page_flags* help determine what the page is being used for.
- *_count* is used to determine the number of distinct users of the page. When the value of *_count* is greater than or equal to zero, the page has (*_count* + 1) users. The value of *_count* is -1 if the page is not being used.
- *_mapcount* is used to determine the number of the page table entries pointing to the page. When the value of *_mapcount* is greater than or equal to zero, the page has (*_mapcount* + 1) page table entries pointing to it. The value of *_mapcount* is -1 if the page is not being pointed to by any page table entry.

- *lru* is a *struct list_head* element that is used to link together *struct page*s to form lists.

The pages are grouped into zones. The number of zones that the pages are grouped into depends on the number of pages present. The different zones in the x86 architecture are ZONE_DMA, ZONE_NORMAL and ZONE_HIGH. These zones are represented by *struct zone*. Each of these zones is suitable for a specific purpose. ZONE_DMA is used by the some DMA processors which cannot address beyond the first 16 MB of the physical memory. ZONE_NORMAL is the zone commonly used by the kernel. It is mapped directly in the kernel address space as the processor cannot use the physical addresses directly. The *mem_map* array is located at the beginning of the ZONE_NORMAL zone.

The remaining memory falls in ZONE_HIGH and is not directly mapped to the kernel address space. To use the memory in this zone, these physical addresses have to be mapped to a particular region in the kernel virtual address space. This zone is not mapped directly to the kernel virtual address space in the x86 architecture due to the lack of virtual addresses in the kernel address space. The virtual addresses in the x86 architecture are 32 bits long and only the upper 1 GB of the 4 GB virtual address space belongs to the kernel. There is no ZONE_HIGH in the x86_64 architecture as 64 bit virtual addresses are used thereby mapping all physical memory directly into the larger kernel address space. Thus the ranges of physical memory represented by the zones in the x86 architecture are:

ZONE_DMA – First 16MB of physical memory

ZONE_NORMAL – 16MB to 896MB

ZONE_HIGH – 896MB to end of physical memory

Each page belongs to exactly one of these zones. Based on what purpose a requested page is going to be used for, different zones are preferred. ZONE_DMA is preferred for use by DMA controllers. ZONE_NORMAL is preferred for use by the kernel as the kernel requires the memory to be always accessible using a specific kernel

virtual address. ZONE_HIGH is used for purposes that do not require the memory to be mapped always. As the zones contain pages that are suitable for a specific purpose, the tasks of page allocation and page reclamation are zone specific. The zones maintain lists of free pages that link together all the *struct page*s of the free pages in that zone that are used for satisfying page allocation requests. The zones have two LRU (Least Recently Used) lists, active and inactive lists, which contain pages from that zone that are in use. As their names indicate, the active list holds the page frames that are believed to be in active use and the inactive lists hold the page frames that are not believed to be in active use. These LRU lists are used by the page frame reclamation algorithm to choose the pages for eviction when the system runs low on memory.

Linux supports the concept of Non Uniform Memory Access (NUMA) according to which physical memory is grouped into nodes based on the access times incurred which in turn depends on their distance from the processors. All the pages in a particular node are assumed to be at the same distance from any processor. These nodes are represented by *struct pglist_data*. All the nodes present in the system are kept in the list *pgdat_list*. The nodes contain an array of *struct zone*s to represent the zones present in the node. The number of zones on a node depends on the number of pages on the node.

Not all architectures differentiate between the pages based on their distance from processors. These architectures have only one node that contains all the available memory in the system; they are said to be Uniform Memory Access (UMA) architectures. The x86 architecture is one such architecture and since it is this architecture that our infrastructure is built on, in our work all the memory is contained in only one node.

2.1.2. Physical Memory Allocation

Physical memory allocation is managed by the zone based allocator using the buddy system [12, 13]. The contiguous free pages in a zone are grouped into blocks containing 1, 2, 4, …, 1024 pages. A block containing 2^n pages is said to be a block of order n. The order of the blocks varies from 0 to 10. As mentioned in [11], the physical address of the first page of a block is a multiple of the block size. The idea behind

grouping contiguous pages into blocks is to avoid external fragmentation that might occur in cases where there is enough memory in the system to satisfy the request but the memory available is not contiguous. The other advantage it has is the ease of address computation for splitting and coalescing blocks thus leading to better performance.

It is not required that all virtual addresses used by a process be mapped to physical addresses at all times. But when a virtual address is being used by a process, it has to be mapped to some location in physical memory. Before the virtual addresses are mapped onto physical addresses, unused physical memory has to be allocated. When requesting physical memory, the number of pages needed is specified along with the Get Free Page (GFP) flags that specifies how the allocator should look for the pages to satisfy the request with. The GFP flags are used by the allocator as a hint to determine the zone to return a page from and how hard to try to reclaim pages to satisfy the allocation requests in the case where enough free pages are not available.

The *struct page*s representing the first pages of the blocks are placed in the free list of the corresponding order. To satisfy a request of order n, the free list corresponding to order n is searched. If the list is not empty, then the request is satisfied with the first block in the list of order n. But if there are no entries in the free list corresponding to order n, the higher order lists are scanned to satisfy the request. The higher order block is split into two halves and the request is satisfied using one half or a part of it. This splitting creates new lower order blocks, which are inserted into the appropriate free lists.

Allocation requests of order 0 are considered as a special case as a large number of allocation requests are order 0 requests. In order to optimize the performance for the common case, a list of random free pages is maintained for each zone and for each CPU so that no lock has to be acquired before accessing the list as each of these lists would be accessed by only one CPU. These lists are called Per CPU free lists. Single page allocations are satisfied from these Per CPU lists.

When pages are freed, adjacent free blocks of the same size, called buddies, are coalesced to form larger blocks of twice the size. This process is repeated until no more coalescing is possible. The physical address of the first page in the block is used along with its size to determine the physical address of the first page of the block's buddy. The physical address of the first page of the buddy block can be used to check if the buddy block is free and if coalescing has to be done. Freeing of single pages is also handled as a special case. The page freed goes to the Per CPU free list instead of the zone's free list. If the size of this list grows to more than a threshold value, the pages are returned to the zone's free lists.

2.1.3. Slab Allocator

The kernel allocates many objects that are small in size compared to the size of a page. Hence it is not a good idea to use up an entire page for such objects. Also the initialization and the destruction of these objects take time comparable to the time taken for their allocation. In order to reduce the time taken for allocation, initialization and destruction of these objects, the slab allocator [14] is used. It maintains caches of objects, where all the objects belonging to a cache are of the same type. The slab allocator reduces the time taken to initialize and destroy an object by reusing it. The memory being used by these caches is called a slab [10]. The pages comprising the slab contain free and in-use objects. When there is no memory left in the slabs for creating new objects, the caches are grown by requesting more memory from the buddy allocator. Since it is only these caches that know about how the pages comprising the slabs are being used, these pages are not reclaimed directly by the Page Frame Reclamation Algorithm. The pages that are used by the slab allocator have their *PG_slab* bit set in their flags.

2.1.4. Page Frame Reclamation

As the kernel allocates pages for various purposes the number of free pages available becomes less and the kernel starts reclaiming pages. In order to determine when to reclaim pages, the zones have threshold values called watermarks associated with them. Each zone has three watermarks called *pages_low*, *pages_min* and *pages_high*. The

values of the watermarks are such that *pages_min* is less than *pages_low* which is less than *pages_high*. The values of these watermarks are proportional to the number of pages present on the zone.

When handling an allocation request, the number of free pages in the zone is compared against the zone's watermarks. Based on the results of the comparison, the allocator might either have to wake up the *kswapd* daemon or start reclaiming pages directly. It invokes *kswapd* if the number of free pages falls below *pages_low*. *kswapd* starts to reclaim pages in parallel to the allocator trying to satisfy the request. If the free page count falls below *pages_min* it starts reclaiming pages directly. The allocation request cannot be satisfied before the direct reclamation has resulted in enough free pages. Thus *kswapd* is invoked before resorting to direct reclaim so as to avoid the latency associated with the direct reclaim.

The pages to be reclaimed are chosen by the Page Frame Reclamation Algorithm (PFRA). Previous work [15-17] suggests that retaining pages accessed by processes recently yields best results. Linux's PFRA implementation mimics the working of the Least Recently Used (LRU) algorithm. The PFRA works on the LRU lists. The LRU lists contain the pages belonging to the user mode address space of the processes and the pages belonging to the page cache. Pages in the LRU lists have their *PG_lru* flag set in their page descriptor. The active list contains the pages that have been accessed recently. The pages have their *PG_active* bit set in their page flags. The inactive list contains pages that have not been accessed recently; their *PG_active* bit is clear. Pages in the active list are moved to the inactive list if they are not accessed for some time. Similarly, pages from the inactive list are moved to the active list if they are accessed frequently. Pages are reclaimed only from the inactive list.

In order to prevent moving pages across the list frequently, the *PG_referenced* bit is used to make decisions about moving [10]. When the page is on the inactive list, it has to be referenced twice before it is moved to the active list. On the first access the *PG_referenced* bit is set and on the next access the page is moved to the active list as the

page already has the *PG_referenced* bit set indicating it was referenced earlier and this is the second time in the being referenced in a short duration. If a page that is referenced once is not referenced again within a given interval, the *PG_referenced* bit is cleared. Similarly, two accesses have to be missed before the page is moved from the active to the inactive list.

The pages on the inactive list are processed one by one to see if they can be reclaimed. Before a page is processed, an attempt is made to lock the page. If the page is already locked it cannot be reclaimed. If the page was not locked previously, then the lock is acquired. If the page is being written back to disk, it is not considered for reclamation. If the page has been referenced lately and if the page is memory mapped or is in the swap cache or is pointed to by PTEs the page is considered not reclaimable and is added back to the active list.

Based on the purpose a page is being used for, pages receive different treatment from the PFRA. Anonymous pages are pages that exist only in physical memory; they do not have a backing store. When such pages are reclaimed, they are added to the swap space. Pages pointed to by the PTEs are reclaimed by modifying the PTEs so that they do no longer point to the page being reclaimed. The *rmap* facility is used for this purpose. Once the mappings are removed, the page is written back to disk if it is dirty. Pages having buffers attached to them are reclaimed by detaching the buffers from the page. Finally, if the page is not being used it is freed. Pages in use are added back to the LRU list.

The pages that are used by the slab allocators are not reclaimed directly by the PFRA as it is not aware of how the pages are being managed and used by the slab allocators. Rather, the PFRA requests the slab caches to shrink themselves and relinquish the pages used as slabs. It does so by calling the shrink function of each of the slab allocators. The shrink functions of the slab allocators release the slab pages that do not currently have any objects allocated on them.

2.2. Virtual Memory Management

Each process has its own address space, which can address up to 4GB of memory in the x86 architecture. Out of these 4GB, the first 3GB addresses can be accessed from both the user and the kernel modes, but the last 1GB is accessible from kernel mode only. The lower 3GB changes with every context switch but the higher 1GB does not.

2.2.1. Kernel Address Space

The last 1GB is the kernel address space and it is shared by all the processes in the system. The first 896 MB of the kernel address space is directly mapped to the ZONE_DMA and ZONE_NORMAL. The remaining 128 MB of the kernel address space is used for various purposes as indicated below in the figure adapted from [11].

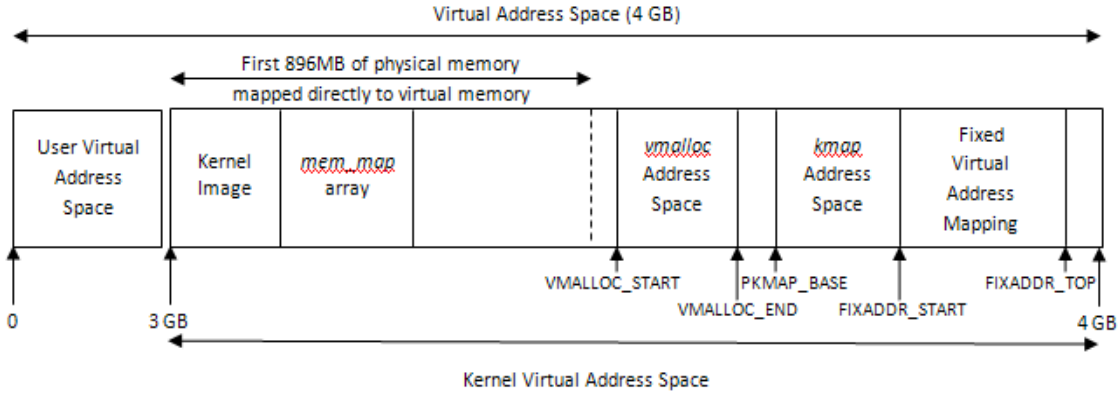

Figure 1: Address Space as seen by the kernel

The address range between VMALLOC_START and VMALLOC_END is used by vmalloc to represent physically discontinuous memory in continuous virtual space. Depending on the amount of physical memory detected at boot time, the following two regions may or may not be present. These regions are present only in machines having more than 896 MB of memory. The address range between PKMAP_BASE to FIXADDR_START is used to map physical address of pages from ZONE_HIGHMEM. They are not mapped directly into the kernel address space; instead, they are temporarily mapped to some virtual address in this address range before they can be accessed. The region between FIXADDR_START and FIXADDR_TOP is used for virtual addresses that are known at compile time. As mentioned in [11], FIXADDR_TOP is statically

defined to be 0XFFFFE000. The value of __FIXADDR_SIZE is determined at compile time and thus the value of FIXADDR_START is determined at compile time too.

2.2.2. Process Address Space Management

Each process has its own address space, which is described by an instance of *struct mm_struct*. The address space of a process is usually not used in its entirety. The address space consists of many page aligned regions of memory that are in use. Each of these distinct regions of memory is represented by a *struct vm_area_struct*. Examples of some of the regions represented by these *struct vm_area_struct*s are the process heap, code segment, and mmaped regions (mmap is used to map the contents of a file to a consecutive set of addresses in the process's address space). These *struct vm_area_struct*s of a process are linked together by a list in the *struct mm_struct*.

2.3. Reverse Mapping

Introduced in kernel version 2.6, a reverse mapping (*rmap*) helps in identifying the Page Table Entries (PTEs) pointing to a page. It is used in the process of page frame reclamation to reclaim pages pointed to by PTEs. The PTEs pointing to a page are identified using *rmap* and subsequently modified so that they do not point to the page any more. In the Linux implementation of the reverse mapping, each *struct page* is used to identify all *struct vm_area_struct*s that contain virtual pages mapped to the page frame. This technique of reverse mapping is called 'Object Based Reverse Mapping' as the reverse mapping is based on *struct vm_area_struct*s rather than on PTEs. The retrieval of the *struct vm_area_struct*s from the *struct page* is implemented differently for anonymous pages and for file backed pages.

2.4. Page Cache

The Page Cache is the cache that stores the data read from the disk. It is checked every time the disk has to be accessed to see if the data already resides in the memory, thereby preventing unnecessary disk I/O. As the name indicates, the page cache is used to cache entire pages. The page cache is used to store contents of the files and thus the file

inode and the offset within the inode are used to identify pages in the page cache. The inode is unique to a file; it stores the information needed by the file system to access the file [11]. It is represented by a *struct inode*, which stores a pointer to the *struct address_space* of the file, which in turn stores the information about how the pages containing the data of the file have to be handled.

The page cache is implemented as a radix tree because a radix tree supports fast lookups even when a large amount of data is stored in it. The pages in the page cache store in their page descriptor the pointer of the *struct address_space* of the inode that owns the page and the offset of the page within the file. Each file has its own *struct address_space*, which contains the root of the radix tree used to store the information about the pages holding the data belonging to the file. When the page cache is searched for data from a file, the radix tree of the file is looked up to check if it has an entry corresponding to the offset of the required data within the file. If the data is present in the memory, the pointer to the page descriptor storing the required data is returned.

The page cache is also used to store blocks read from disk that are not part of any file. For instance, inodes and super blocks read from disk are cached to prevent reading the same data from disk multiple times. The pages that store such data are called buffer pages; they have *struct buffer_head*s associated with them. The *struct buffer_head*s store information about the block being cached. As the size of the page is usually greater than the size of the block, more than one block may be stored in a page. Thus more than one buffer head may be associated with the page, one for each block. These buffer heads are linked together by a list. The head of the list is stored in the private member of the page descriptor.

The swap cache is a part of the page cache that is used to hold anonymous pages while they are being swapped in or out. The swap cache is checked to see if the required page is in it before starting any swap in or swap out operations. If the page is found in the swap cache, no swap operation needs to be performed and the page found in the swap cache is used. Thus it is used for synchronization between two processes trying to swap

in the same page and between the PFRA and a process that tries to swap in a page being swapped out by the PFRA. As anonymous pages are not backed by any files on disk, they are located in the cache using a common global instance of *struct address_space* called the *swapper_space*, which is indexed by the location in the swap space.

3. Modifications to the Linux Memory Manager

The infrastructure we have designed and implemented allows the user to vary the amount of physical memory used by the system at run time. The available physical memory was divided into logical banks. Each bank can be turned off or on independent of the others.

In order to enable the user to turn off memory banks, we had to provide and implement the following items:

- We had to provide an interface for the user to interact with the kernel and instruct it to turn a bank off. We used a system call for this purpose. The user specifies the number of the bank that he wants to turn off and the kernel tries to turn the bank off.
- Once the user instructs the kernel to turn a bank off, the pages from the bank that are being used must be reclaimed before turning the bank off. Reclaimable pages belonging to the bank are contained in the LRU lists of the corresponding zone. We iterate through the LRU lists, freeing the pages on the concerned bank after migrating their contents to pages on the other banks. Migrating means copying the contents of the page to a new page, changing the page table entries pointing to the page to point to the new page and changing the pointers in the caches, though not all of these are applicable to all kinds of pages. As an alternative to migrating the contents of the pages, we could have written the pages back to disk or discarded them, as appropriate. One of the drawbacks of this approach is that it would lead to performance degradation as these pages might have to be read back in from the disk if they are accessed in the future. Also, turning off a bank might take more time if disk accesses were involved. So this approach was not taken.
- After reclaiming the pages belonging to the concerned bank, we remove the free pages belonging to the bank from the buddy allocator's free lists so that memory allocation requests made in the future are not satisfied using these pages.
- Not all pages are reclaimable. The only pages that the page frame reclamation algorithm considers as potentially reclaimable and acts directly on are the pages on the LRU list. Not all pages that are in use are present on the zones' LRU lists. It is not

possible to directly reclaim or move the pages that are not present in the LRU lists. Apart from pages not on the LRU lists, certain pages such as the pages used to hold information about the super block of a mounted file system, which are used for the entire duration during which the file system is mounted, can also not be reclaimed. The pages used to hold information about super blocks are part of the page cache; they have *struct buffer_head*s associated with them as mentioned in chapter 2. If the bank the user wants to turn off contains pages that are not reclaimable, the bank cannot be turned off. To deal with this problem proactively, we had to redirect some of the allocation requests to the first few banks in the ZONE_NORMAL, which are not expected to be turned off. We call these consecutive banks 'low banks.' Thus, an assumption made by our design is that a certain minimum number of banks has to remain turned on at all times.

In order to enable the user to turn banks on, the following was done:

- The user uses a system call to turn banks on. Similar to the system call that turns the bank off, this system call accepts the bank number of the bank to turn on.
- The pages reclaimed from the bank are returned to the buddy allocator's free lists, thereby putting them into circulation again.

3.1. Modified Page Descriptor

We have added two new flags to the page flags. The first flag is called *PG_captured*. This bit is set for all the pages on the banks that are turned off and is clear for all the pages in the banks that are turned on. This bit is set when pages are removed from the buddy allocator's free lists. The value of this bit is checked in all the pages of the bank to determine if the bank could be physically turned off. Though this functionality of determining whether all the pages of the bank have been reclaimed could have been implemented by counting the number of pages removed from the free lists, we resorted to having this flag as it helps us make sanity checks in multiple parts of the kernel. This flag is cleared when adding the pages back to the free lists at the time of turning a bank on.

The other flag added is called the *PG_migration* flag; it is used for synchronization between the page fault and the page migration code. This bit is set before a page is migrated and is cleared right after the migration is complete. In order to prevent write accesses to the page being migrated, the PTEs pointing to the page are temporarily marked read-only. Write accesses to the page under migration thus result in page faults. The *PG_migration* bit is used by the modified page fault handler to differentiate between illegal write accesses to the page and write accesses to pages under migration.

3.2. Modified Buddy Allocator

The buddy allocator had to be modified to differentiate between allocations that should be directed to the set of banks that would never be turned off and other allocations. This has been done by partitioning the ZONE_NORMAL zone and by adding and modifying GFP flags. The other change made to the buddy allocator was to make sure it did not use pages from the bank being turned off to satisfy the allocation requests made for replacement pages when turning a bank off.

The zone ZONE_NORMAL has been logically divided into two sets of banks – low banks and non-low banks. These low banks are the banks that have to remain turned on all the time. The *low_bank_mark* variable denotes the index of the last low bank. The intent is to allocate pages that cannot be migrated in the low banks and other allocation requests are satisfied from the non-low banks.

A new flag __GFP_LOWBANK was added to the existing set of GFP flags. The page frame allocator was modified to redirect requests with the __GFP_LOWBANK flags to the lower banks in ZONE_NORMAL. This new flag affects the behavior of the allocator only if the zone being considered for allocation is ZONE_NORMAL. Other allocations proceed as they would have without the introduction of the new flag.

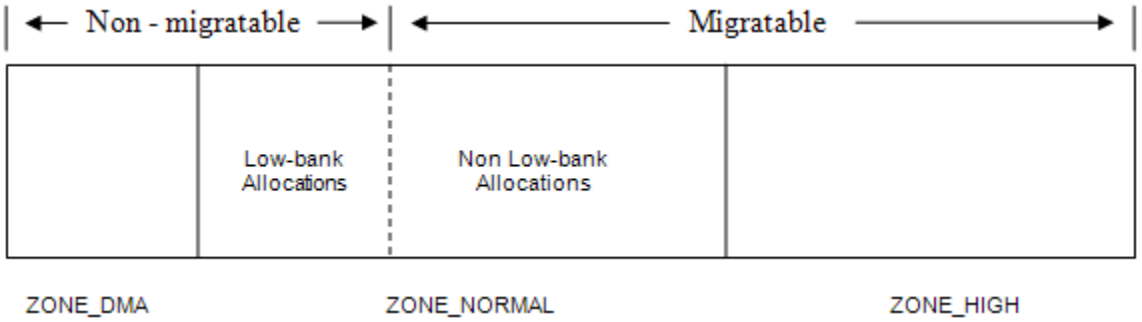

Figure 2: Migratable and non-migratable parts of the physical memory

GFP_KERNEL was redefined to include the new flag introduced because GFP_KERNEL is passed to the allocator when allocating memory for many kernel data structures. These pages are usually hard to migrate or cannot be migrated as they are not added to the LRU lists. With the redefinition of GFP_KERNEL to include __GFP_LOWBANK, such pages are allocated on the low banks and thus we do not have to worry about migrating them as the low banks are never turned off.

The pages that belong to the banks that are turned off are removed from the free lists before turning the banks off. Thus the allocator will not satisfy allocation requests using pages from a bank that is being turned off. But the pages belonging to the bank being turned off are removed from the free lists only after reclaiming all pages from the bank that are in use. During the reclamation process, free pages are needed to move the contents of the pages being reclaimed to. Care has to be taken to prevent the allocator from returning free pages belonging to the bank being turned off. The allocator was prevented from satisfying the requests using such pages by marking that bank to be turned off in the *banks_on* array, the array that maintains the on/off status of the banks, - even though its pages are still on the free lists - and by making sure that any page that is returned by the allocator is always from a bank that is turned on.

Before pages are returned, the *free_pages_count* array, which keeps track of the number of free pages on each bank, is updated. Similarly, when pages are freed, the *free_pages_count* is updated.

3.3. System Calls Added

Two new system calls, turnoffnode and turnonnode were added to the kernel to turn off and turn on a bank, respectively.

3.3.1. System Call 'turnonnode'

The system call turnonnode marks the specified bank to be turned on in the *banks_on* array and releases the pages that had been captured from that bank. These pages are freed using the function *free_pages* that is also used by other parts of the kernel to free pages. Before the page is freed, its *PG_captured* bit is cleared. These freed pages are put into the buddy allocator's free list once again and thus can be used by the system. As the values of the watermarks must be proportional to the number of pages present in the zone, the zone watermarks are recalculated after a bank is turned on. When a bank is turned on, the number of pages present in the zone increases and hence the values of the watermark must be increased too.

3.3.2. System Call 'turnoffnode'

This system call marks the specified bank to be turned off in the array *banks_on*. It then scans the active LRU list and moves the pages on it that belong to the concerned node onto a temporary list 'list1'. The pages on this temporary list are then considered one by one for migration. Removal of the pages being considered for migration from the LRU lists prevents them from being concurrently acted upon by the PFRA.

If the page is locked or if it is marked as reserved, then the page cannot be migrated. When such a page is encountered, the process of turning the bank off is aborted. For the pages that are not locked or reserved, a new page is requested from the buddy allocator to copy its contents to. If the bank being turned off is a high memory bank (i.e., a bank from ZONE_HIGHMEM), the __GFP_HIGHMEM flag is included in the GFP flags used for allocating the replacement page, indicating that a page from ZONE_HIGHMEM is preferred. The reason for preferring a page from ZONE_HIGHMEM is to keep the pressure on ZONE_NORMAL minimal as it is the

performance critical zone. Also, pages from ZONE_HIGHMEM cannot be used for all purposes. When reclaiming a page from ZONE_HIGHMEM, it can be assumed that any page from ZONE_HIGHMEM can be used as a replacement for the page being reclaimed. If the bank does not belong to the ZONE_HIGHMEM, __GFP_HIGHMEM is not included in the GFP flags when requesting a page. If for some reason the allocation request fails, then the process is aborted.

If the page can be migrated, then the page is freed after its contents are migrated to the new page just allocated. The new page is added to a second temporary list 'list2.' If the page cannot be migrated, then it is added to a list 'list2' and the new page that was just allocated is freed. Once all the pages that were moved from the active LRU list to the temporary list are processed, the pages on the list 'list2' are added back to the active LRU list. In order to handle the cases where the process of turning off the bank is aborted due to the above mentioned reasons, the pages from the list 'list1', if any, are also added back to the active LRU list. By the end of this process, the active LRU list has either the original page from the bank or the page that the contents of the page from the bank have been moved to. But at this point of time, the order of the pages in the active list would have changed thus affecting the behavior of the PFRA.

This entire process is then repeated for the inactive LRU list. Instead of migrating the pages on the inactive list, we considered the possibility of evicting dirty pages to disk and freeing the pages. But we decided against this idea due to the time involved in disk accesses and the fact that a page on the inactive list could be referenced in the future.

The pages that were freed are placed on the free lists, but the buddy allocator will not use these pages to satisfy requests because the bank has been marked as turned off (even though it has not yet been turned off.) The free lists and the per cpu page lists are scanned for pages from the concerned bank and they are removed from those lists. The zone watermarks are then adjusted accordingly.

3.4. Page Migration

Page migration is the technique adopted to move the pages from one bank to another. In the following discussion, the page that is being moved is referred to as the source page and the page that the source page's data is being moved to is referred to as the destination page. The steps involved in moving a page are outlined below:

1. The source and the destination pages are locked. The *PG_migration* bit of the source page is set.

2. The source page is checked both for membership in the page cache and the swap cache (a page can be present in only one of these two caches at a given point of time). Pages in the swap cache have their *PG_swapcache* set in the flags of the page descriptor. Membership in the page cache is identified by retrieving the page descriptor of the page that is storing the data corresponding to the address space stored in the source page descriptor's 'mapping' field and the offset stored in field 'index'. It is then compared with the source page's page descriptor. If the pointer matches the *struct page* pointer of the source page, then the page is part of the page cache. Some of the page cache pages have buffer heads associated with them; such pages have their *PG_Private* bit set in their flags.

   If the source page belongs to the swap cache, then the swapper space's radix tree lock is acquired in write mode. If it belongs to the page cache, then the radix tree lock of the address space it belongs to is acquired in write mode. This is done to prevent anyone from looking up the page or the swap caches and start using this page.

   The pages belonging to the page cache and swap cache can be considered for migration only if the pages are not being used at the moment. Pages holding the data of files that are memory mapped are part of the page cache. These pages are also pointed to by page table entries. It is fine to migrate such pages, as long as they are not also being used otherwise at the moment, because the page table entries pointing to the page can be modified to point to the page the data is migrated to. A page in the

cache is considered to migratable only if _count is greater than _mapcount by at most 1.

3. Before the data on the source page can be moved it has to be write protected in order to prevent the loss of writes that could occur at the time of migration. In order to write protect the page, the PTEs pointing to the source page, if any, should be marked as read-only. The reverse mapping (rmap) technique outlined in the previous chapter is used to identify the PTEs pointing to the source page. Not all PTEs pointing to a page necessarily have equal access rights. Hence the write access of each PTE is recorded before it is made read-only. The access rights are stored in an array along with the *struct vm_area_struct* pointer and the virtual address. This array is looked up later to restore the original write permissions. After recording the original write permissions, the PTEs are marked read-only.

4. The data is then copied from the source page to the destination page. The kernel virtual addresses of the pages are used for this purpose. Therefore, high memory pages that are not already mapped into the kernel address space have to be mapped into the kernel address space to obtain a virtual address by which to address those pages.

5. The contents of the page descriptor of the source page are copied over to the page descriptor of the destination page.

6. If the page being moved is a buffer page, the associated buffer heads have to be changed to point to the destination page. Before the buffer heads can be manipulated, the address space's private lock is acquired. The buffer heads are checked for their usage count. If the buffer head is currently in use the source page cannot be moved. Also, if the buffer head is locked the page cannot be moved. Once all the buffer heads pointing to the source page are checked for these two conditions, the buffer heads are modified to point to the destination page. Each *struct buffer_head* stores the address of its associated data within a page if the page is a non-high mem page. For high mem

pages for which no virtual address is available, the offset within the page is stored instead. This address/offset field is updated as well. The mapping's private lock is released after all the operations on all the buffer heads are complete.

7. If the source page is a part of the page cache, the entry corresponding to it is deleted from the radix tree of its address space and a new entry is created that points to the destination page. If the source page is part of the swap cache, the entry corresponding to it is deleted from the radix tree of the swapper space and a new entry pointing to the destination page is created. Once the entry is changed, the corresponding radix tree's lock that is being held is released. Any further lookups will return the destination page.

8. The PTEs are modified to point to the destination page. The original write access rights are restored if the PTEs were marked read-only earlier. Once again the reverse mapping is used to access the PTEs pointing to the source page; the array storing the access rights is looked up to retrieve the original access rights.

9. The *PG_migration* bit of the source page is cleared and the source and the destination pages are unlocked.

If the page cannot be migrated after the PTEs are marked read-only, the access rights are restored and the new page is freed.

3.5. Page Fault Handler

The page fault handling code had to be modified in order to handle write faults on the page that is under migration. This case is identified by the following conditions:

1. The page table entry is present, i.e., there exists a page that is mapped to the virtual address the process is faulting on.
2. The PTE is marked as non-writable.
3. The page that is pointed to by the PTE has its *PG_migration* bit set.

This case is handled by making the faulting code wait for the page to be unlocked. As the page is locked during migration, this page fault handler cannot process the fault until after the page is migrated.

3.6. Modified Allocation Sites

So far we have explained how various types of pages are identified and migrated. To deal with pages that are not reclaimable either because they are not on the LRU lists or because they are being used for a long duration, we allocate them on the lower banks which are assumed to be turned on at all times. To allocate such pages on the lower banks, GFP_KERNEL was redefined to include __GFP_LOWBANK. The redefinition of GFP_KERNEL redirected all the allocations that passed GFP_KERNEL as the GFP flag at the time of requesting memory to the lower banks.

Some of the sites where memory was allocated for non-reclaimable pages did not use GFP_KERNEL as part of the GFP flags passed to the allocator. Such sites had to be changed to include the __GFP_LOWBANK flag in the GFP flags when requesting pages. In order to identify the sites that these non-reclaimable pages were allocated at, we modified the page descriptor *struct page* to include an array of addresses and an integer to track the current index in the array. The array is used to remember the calling context of the process by storing the return addresses retrieved from the call stack at the time the page is being allocated.

The pages that could not be moved were those pages from a bank that were not present on the LRU lists and the free lists. At the time of turning a bank off, the page frame numbers of these pages that were not migrated were printed. A system call was added to the kernel that takes the page frame number and prints the call stack recorded for that page. This system call accesses the page descriptor of the page with the given PFN from the *mem_map* array and prints the stored call stack using the kernel's built-in facility to print the name of a function given its address, allowing us to identify the allocation site of the page.

We have redirected most of the pages we were not able to move to the lower banks for the kernel we worked with. Depending on the modules included and the kernel configuration, there might be other places where changes might have to be introduced. As the changes to be made depend on the specific kernel configuration, the following discussion may not cover all required redirections.

3.6.1. Slab Allocation Redirection

Pages used by the slab allocator are not placed on the LRU lists. Therefore, they had to be redirected to the lower banks. The slab allocator lacks the ability to free pages from a particular bank on demand. This is because one or more of the objects located within those pages might be in use. The approach we have taken is to redirect the pages used by the slab allocators and hence avoid taking them into consideration when turning a bank off. The GFP flags passed to the allocator when requesting pages for the slabs in the function *kmem_getpages* now includes the __GFP_LOWBANK flag. Thus the pages returned by the buddy allocator will be on one of the lower banks that are turned on at all times.

3.6.2. Buffer Pages Redirection

As mentioned earlier, buffer pages are used to cache metadata. These metadata are used frequently and hence such pages are cached for a very long time, making such pages hard to be moved. An example of frequently used, long term cached data is information about the superblocks of mounted filesystems. We have redirected the allocation of pages used to hold the superblock by identifying the buffer page allocations when a mount request is processed. As soon as the request to mount a filesystem is issued, a flag is set in the process's task struct. The code that requests for pages to be used as buffer pages checks for this flag and redirects the allocation by requesting a page from the lower banks using the __GFP_LOWBANK flag.

We have not redirected all the pages used as buffer pages to the lower banks as we wanted to keep the low bank count small. This has led to the case where some pages

used as buffer cache pages that are used for a long time come from the higher banks of the ZONE_NORMAL; hence we are not able to turn off these banks.

3.7. Kernel Daemon kvmpowerd

A new kernel daemon was added to scan the memmap array periodically and to report the percentage of pages used for various purposes. It reports the count of the following kinds of pages on each bank

- Free pages
- Anonymous pages
- Page Cache pages
- Swap cache pages
- Buffer pages
- Slab Cache pages
- Pages on the LRU lists

3.8. Limitations

One of the limitations of our implementation is that the system cannot be made to support more memory than was available to it at boot time. Thus only the nodes that were previously turned off can be turned on. This is because the space for *mem_map* is allocated based on the amount of memory detected at boot time and it cannot be expanded during the operation of the system. A solution to overcome this problem has been mentioned in [7]. The idea is to break the *mem_map* array into multiple pieces and provide lookup tables to support the functionality achieved with a single contiguous *mem_map*. Another limitation is that we cannot turn off the low banks, as discussed earlier in this chapter.

4. Experiments and Results

The goals of the experiments were to:

1. Evaluate the suitability of the current memory manager designs to support the dynamic variation of physical memory
2. Determine the impact the available physical memory has on the performance of different kinds of applications
3. Determine the limitations of our implementation
4. Verify the correctness of our implementation

The performance of the applications was measured at various levels of memory to determine the impact the available physical memory had on the applications. The performance recorded at various memory levels was used as an indicator of the memory needs of the applications. To evaluate the suitability of the current memory manager for the purpose of dynamically varying the memory, the performance recorded on the modified kernel was compared with the performance recorded on an unmodified version of the kernel.

4.1. Experimental Setup

The experiments were performed on a machine with a Pentium D 2.8 GHz processor and 4 GB of RAM. The kernels were not configured to support SMP as we chose a platform that was most common. The experiments were repeated for various memory sizes ranging from 2048 MB to 896 MB. For the experiments performed, we did not see a substantial difference in performance for memory ranges above 2 GB of RAM; hence, we limited the memory to 2 GB.

The configurable logical bank size was fixed at 32 MB. We felt that setting the bank size at 32 MB was a reasonable choice as it is small enough to make fine grained measurements and is large enough to be considered as the unit of memory that can be turned off or on independent of the other units in a useful manner.

The modified kernel was booted with 2GB of memory and the experiments were performed for various sizes of memory without rebooting. The memory available was varied using the new system calls. An alternative option would have been to restart the machine after every run of the experiment and use the system calls to resize the memory to the desired level and run the experiment again. We decided against this option as this is not how our infrastructure is expected to be used in real world scenarios where memory resizing is expected to be applied to long running systems.

Some of the experiments were repeated on the unmodified kernel. The system was restarted after every run of the experiment for a particular memory size because we used a boot time option to set the amount of memory the kernel should use.

The /proc entries vmstat and meminfo were scanned and recorded periodically when the benchmarks were running. Among other statistics, we recorded the number of page faults incurred during the time the benchmark was running.

4.2. Experimental Results

In order to determine the relationship between physical memory and application performance, the same experiment was carried out with differing amounts of physical memory. The results obtained on the modified kernel were compared with the results obtained by running the same experiments on the unmodified kernel to evaluate the suitability of the current memory management designs for supporting dynamic variation of memory. Correctness of our implementation was determined by comparing the output of the experiments obtained on the two different kernels.

One of the limitations of our implementation was that we could not turn off some of the higher banks in ZONE_NORMAL. This was because we were not able to migrate some of the buffer pages in those banks. This is one of the reasons for limiting the experiments to the ZONE_HIGH region. The other reason for limiting the experiments to ZONE_HIGH is because of the fact that ZONE_NORMAL is the performance critical zone and we did not want to turn off banks belonging to ZONE_NORMAL. Thus the

results our experiments are not applicable to machines with less than 896 MB of memory or machines that do not have ZONE_HIGHMEM.

4.2.1. Sort

In this experiment, a very large array is allocated and is filled up with random numbers and the array is then sorted. Quick sort is used to sort the array. As it deals with no disk backed data, most of the pages it uses are anonymous pages which are resident in the memory, unless swapped out to the swap space. When running low on memory, this workload produces many swap/swap cache pages. It does not make much use of the page cache.

Quick sort makes O(n lg n) comparisons on average to sort n elements. It makes O(n^2) comparisons in the worst case. In each iteration, it chooses a pivot element from the array and compares the other elements in the array against the pivot. The elements lesser than the pivot are moved before the pivot and the elements greater than the pivot are moved after the pivot. This process is repeated for the parts of the array on either sides of the pivot. As the pivot is compared with all the elements of the array sequentially in most of the implementations of Quick sort, the pivot value can be stored in a register and a good locality of reference is achieved due to the sequential access of the array [18].

The array allocated in our experiments had 730 million integers and thus the memory requirement was about 2.71 GB. The size of the array was limited by the 3 GB restriction on the user virtual space of the process.

When the available free memory is greater than 2.71 GB, which is what the benchmark demands, the entire data is expected to fit in the memory available and thus the performance is expected to be high. When the available memory is less than 2.71 GB paging occurs and the performance is expected to degrade. The number of page faults increases proportionally to the number of banks turned off and hence the performance is expected to degrade as more banks are turned off.

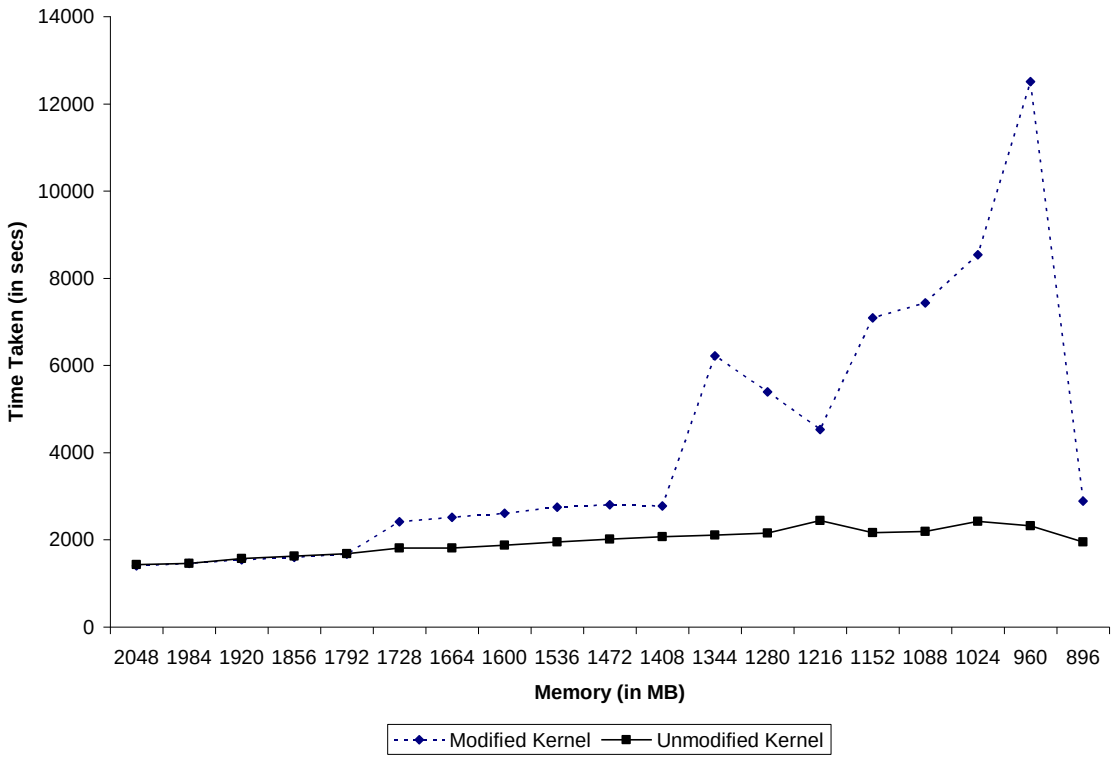

Figure 3: Times taken to sort at various memory levels

The above figure shows the performance obtained at various memory levels in the modified and the unmodified kernels. As we had expected, we see a degradation in the performance as more banks are turned off. The degradation in performance on the unmodified kernel seems gradual and is close to what we had expected. The uneven drops in the performance on the unmodified kernel were not expected. We had expected the shape of the curve denoting the performance on the modified kernel to be similar to the curve denoting the performance on the unmodified kernel but shifted to the left. The following graphs plotted with the data obtained from the /proc entries scanned during the experiments help us understand the results shown above.

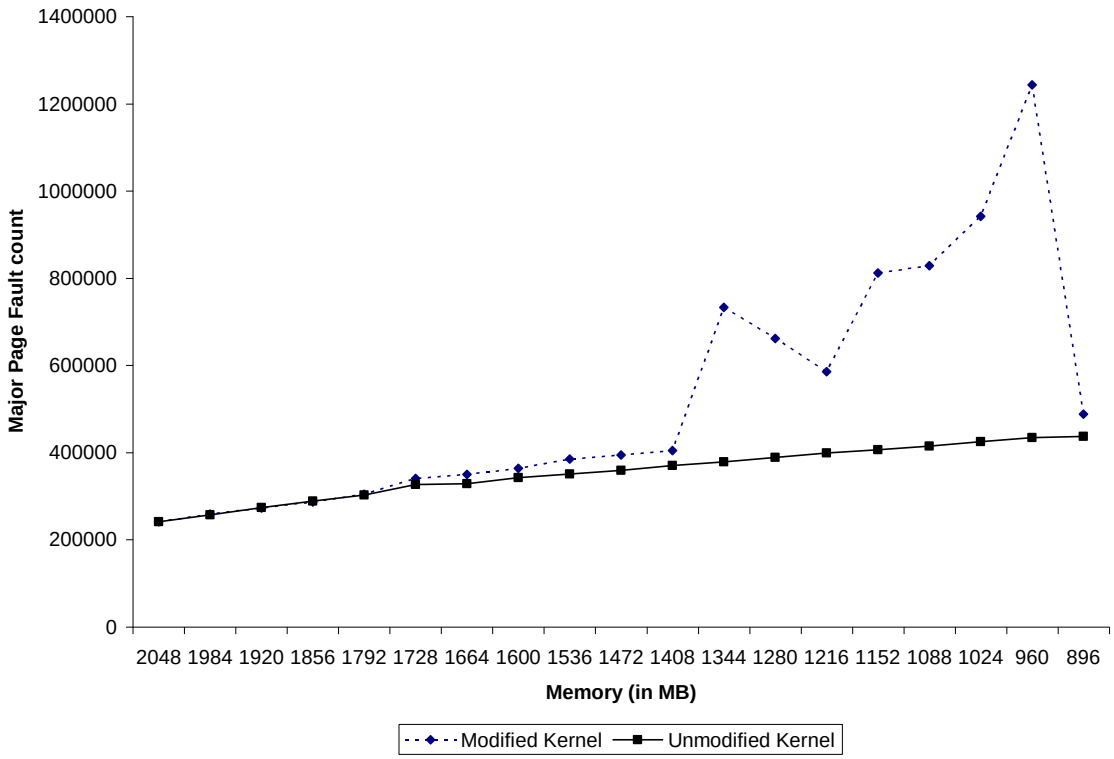

Figure 4: Sort Experiment - Major Page Faults incurred at various memory levels

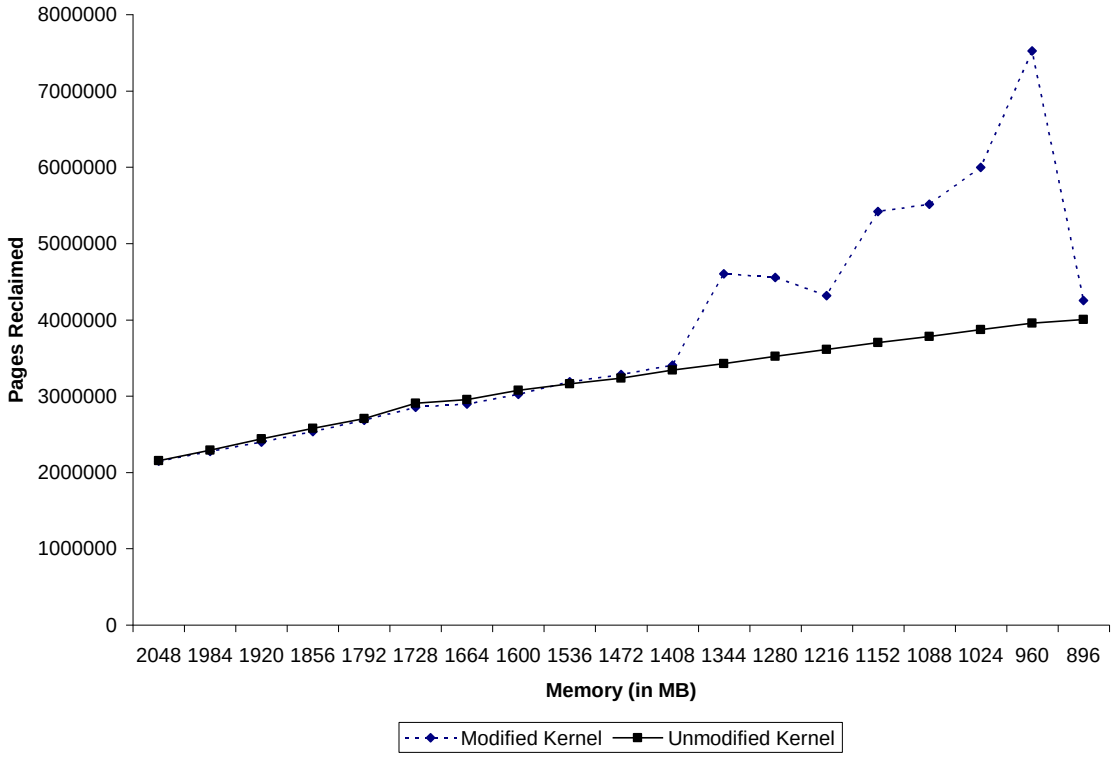

Figure 5: Number of pages reclaimed during the sort experiment at various memory levels

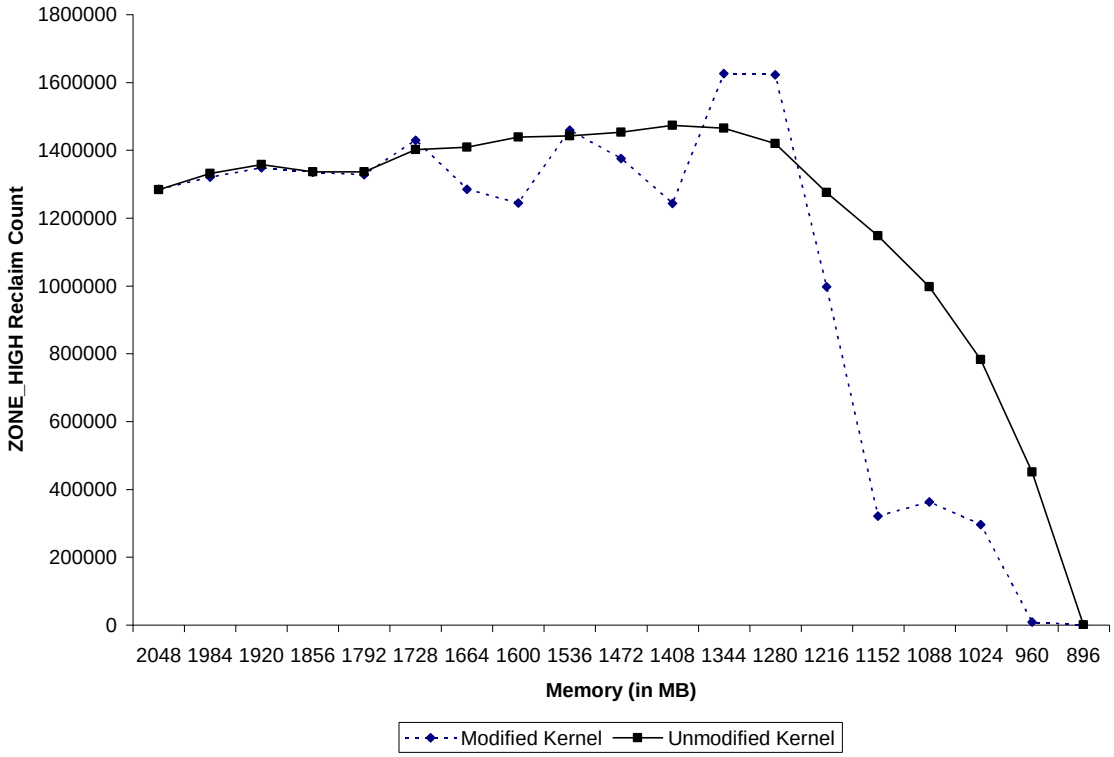

Figure 6: Number of ZONE_HIGH pages reclaimed during the sort experiment at various memory levels

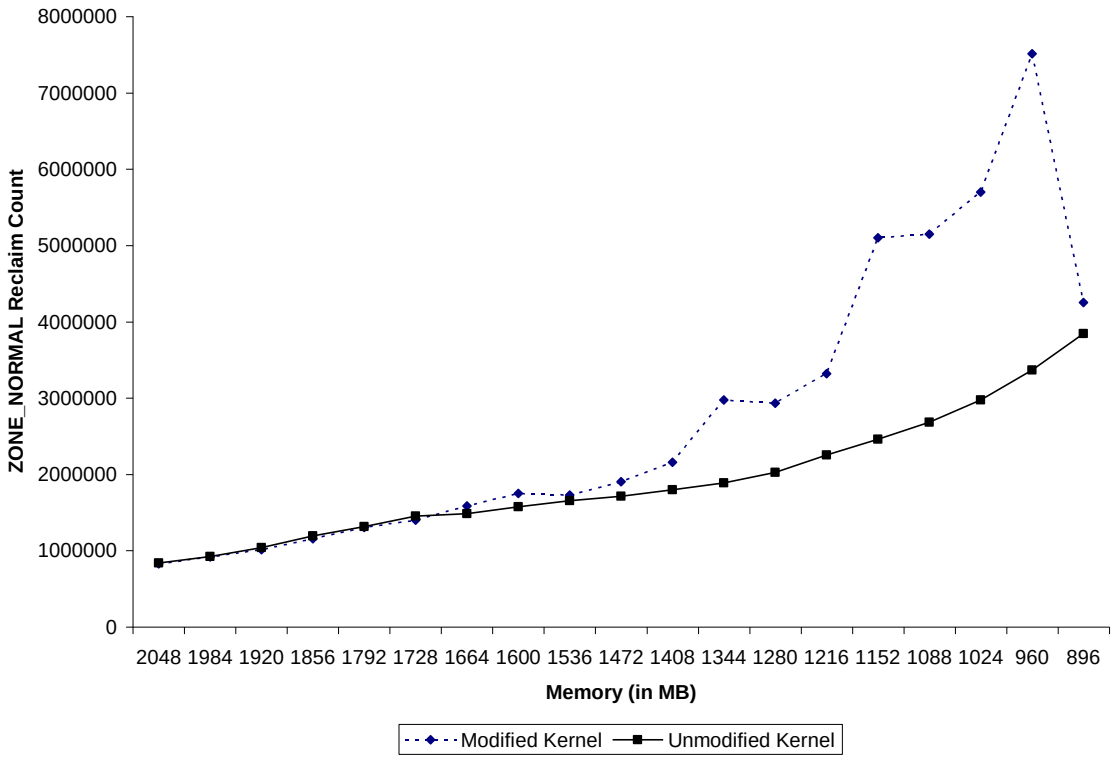

Figure 7: Number of ZONE_NORMAL pages reclaimed during the sort experiment at various memory levels

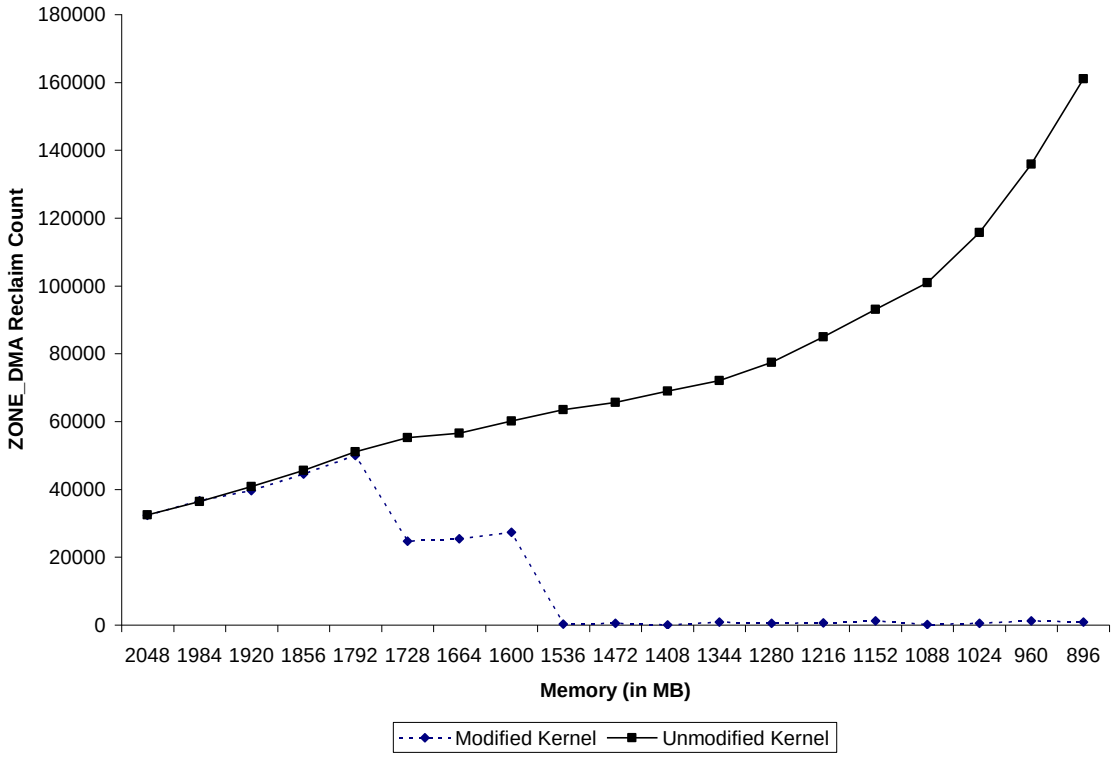

Figure 8: Number of ZONE_DMA pages reclaimed during the sort experiment at various memory levels

As seen from figure 3, the time taken on the modified kernel is much more than the time taken on the unmodified kernel for those memory sizes for which the number of pages in ZONE_HIGH is low. The higher time taken on the new kernel is due to higher reclamation activity, which leads to an increase in the major page fault count. The increase in number of major page faults signifies an increase in the number of disk accesses.

We had anticipated a higher page reclamation activity if we turned off some banks and left the watermarks unchanged. This is because the watermarks would have been higher than what they should have been for the new memory size leading to a higher reclamation activity than what is needed. In order to avoid this situation, the watermarks were decreased or increased depending on the operation. When a bank is turned off, the number of pages present on the zone it belongs to is reduced and hence we lower the watermarks. When previously turned off banks are turned back on, the watermarks are

increased. Thus the higher reclamation activity cannot be attributed to disproportionate watermarks.

The memory range we have conducted the experiments is between 2GB and 896 MB. No bank belonging to the ZONE_NORMAL or the ZONE_DMA was turned off. All the banks that were turned off belonged to the ZONE_HIGH. Had the aggressive reclamation been due to the disproportionate watermarks, the number of pages reclaimed from the ZONE_HIGH in the modified kernel should have been more than the number of pages reclaimed from ZONE_HIGH in the unmodified kernel, which is not the case here as shown in the graphs. Rather, it is the higher reclamation activity in the ZONE_NORMAL that contributes to the higher count of the total number of reclaimed page count in the modified kernel.

The higher reclamation activity in ZONE_NORMAL is due to the fact that the ZONE_NORMAL watermarks are reached sooner in the modified kernel than in the unmodified kernel. This can be attributed to the following reasons:

i) Larger space occupied by the *mem map* array, reducing the amount of memory available in ZONE_NORMAL
ii) Redirected allocations occupying ZONE_NORMAL, which would otherwise have been satisfied from ZONE_HIGH
iii) Migrated pages occupying ZONE_NORMAL

As the modified kernel was booted up with 2 GB, the *mem map* array located at the start of ZONE_NORMAL has 524288 (2 GB = 524288 pages of sizes 4096 bytes) entries. When memory is reduced the space allocated for the *mem map* array for describing the pages on the banks being turned off cannot be reclaimed thus leading to wastage of memory from ZONE_NORMAL. The memory thus wasted increases as more memory is turned off. As an example let us estimate the memory wasted because of the large size of *mem map* array when the kernel was booted up with 2 GB of memory and then reduced to 896 MB. We have entries describing 1152 MB of memory which is no more available. The size of a *struct page* is 32 bytes and thus the wastage is 9 MB. A

solution to prevent this wastage, as mentioned in [7] and [8], is to break down the *mem_map* array into segments and allocate them as needed.

As mentioned in the Section 3.6, some of the allocations have been redirected to the lower banks which are in ZONE_NORMAL. Out of the redirected allocations, some allocations would have been usually satisfied from ZONE_HIGHMEM. These redirected allocations exert extra pressure on ZONE_NORMAL.

When turning off banks after running the experiment for a particular memory size, all the pages in the banks that are turned off were migrated to pages on the other banks. Since we turned off the banks with the higher index before the ones with the lower index, these pages were moved to the lower banks. Even though we ask the allocator to allocate pages from ZONE_HIGH when migrating pages from ZONE_HIGH, the pages returned could be from ZONE_NORMAL when there are not enough free pages from ZONE_HIGH. This would occur more frequently when there is less free memory in ZONE_HIGH. This explains the additional pressure on ZONE_NORMAL, especially in cases where there was not enough memory in ZONE_HIGH.

The following graph gives a rough estimate of how many extra mega bytes of ZONE_NORMAL memory is used up in the modified kernel compared to the unmodified kernel. This data was obtained using the sysrq memory dump option right after system startup and before turning off banks. These results thus do not include the additional pressure exerted on ZONE_NORMAL because of migrating ZONE_HIGH pages to ZONE_NORMAL.

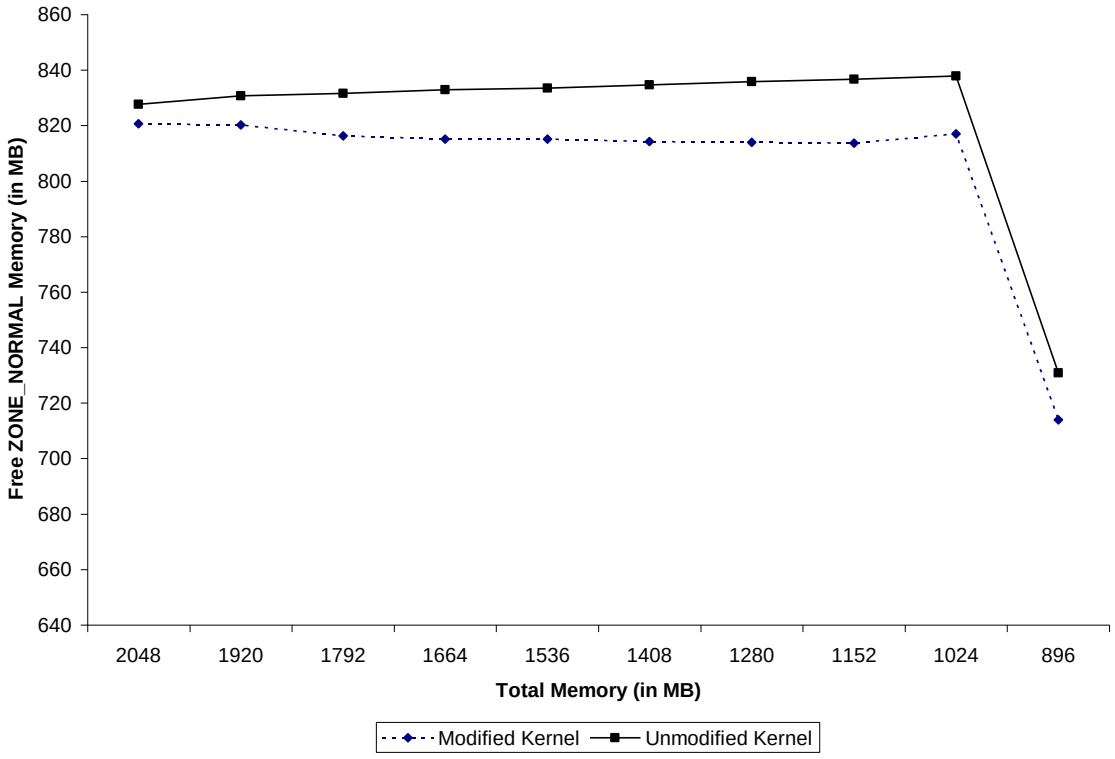

Figure 9: Differences in free memory from ZONE_NORMAL for various memory sizes

As can be seen from the graph, the difference in the free memory available in the ZONE_NORMAL increases as the total memory sizes is reduced, thereby explaining a higher reclamation in ZONE_NORMAL at these memory sizes.

Since the difference in performance between the modified kernel and the unmodified kernel is due to increased pressure on ZONE_NORMAL, the performance difference decreases at higher memory levels. For x86_64 architectures, which do not have a ZONE_HIGH zone, and therefore almost all available memory is in ZONE_NORMAL, we expect the performance difference to be minimal.

4.2.2. HPL

HPL is a portable, freely available implementation of the Linpack benchmark. It solves a randomly generated linear system represented as Ax = B. The working of the benchmark is beyond the scope of this work; more information about it can be found in

[19]. Here, we report only the time taken to run this benchmark and its memory characteristics.

This test is similar to the earlier test in the sense that it allocates a large chunk of memory initially. Like Quick sort, it predominantly makes use of anonymous pages. Thus as the memory is decreased, performance is expected to fall due to increased paging activity.

The results of this test are similar to the results obtained for the previous test. The graphs are shown below.

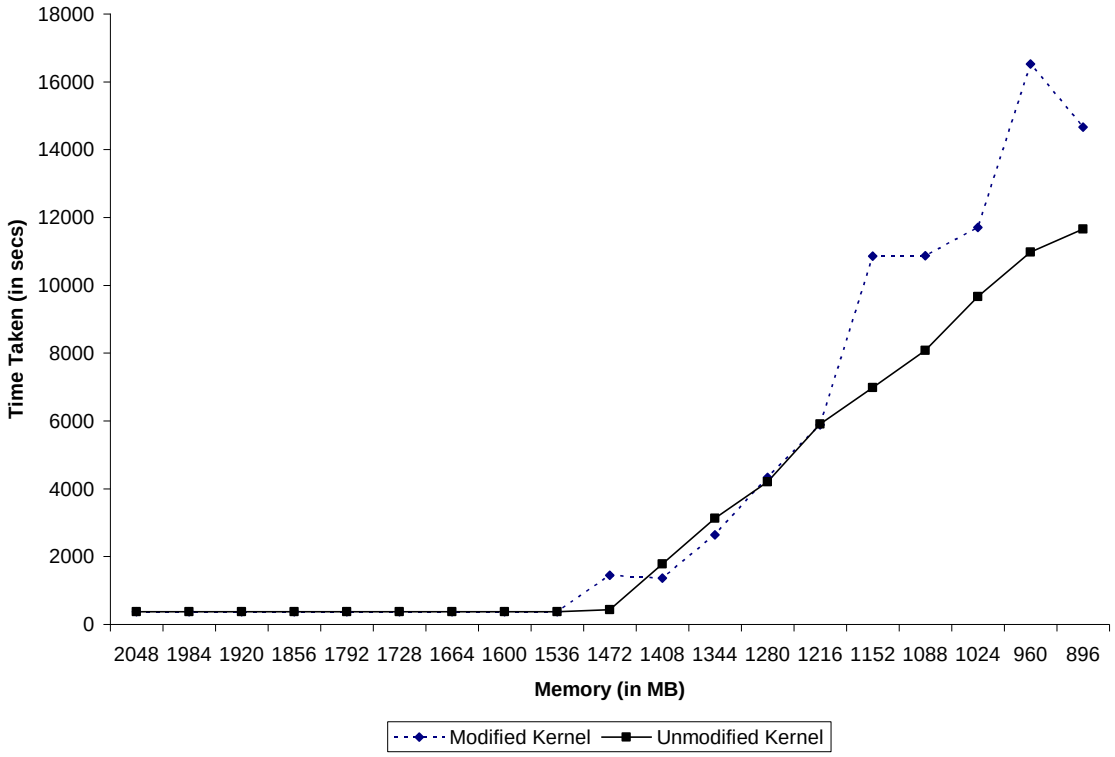

Figure 10: Run times for HPL benchmark at various memory levels

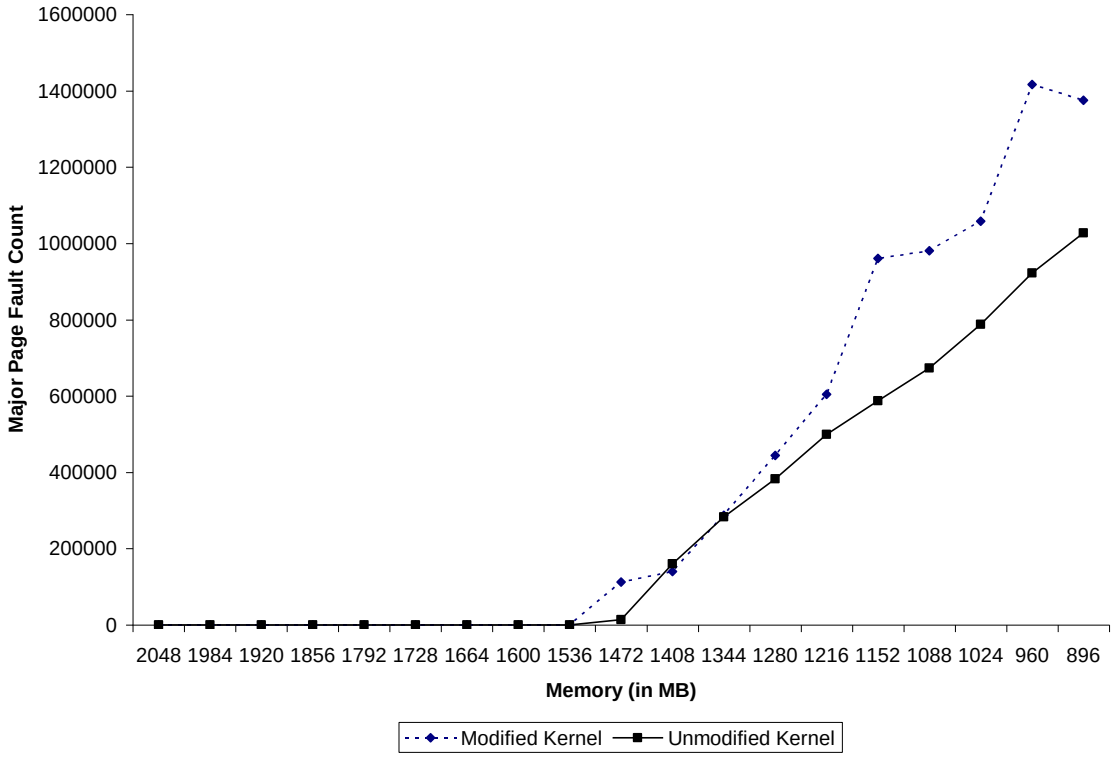

Figure 11: Number of Major Page Faults during the runs of HPL benchmark at various memory levels

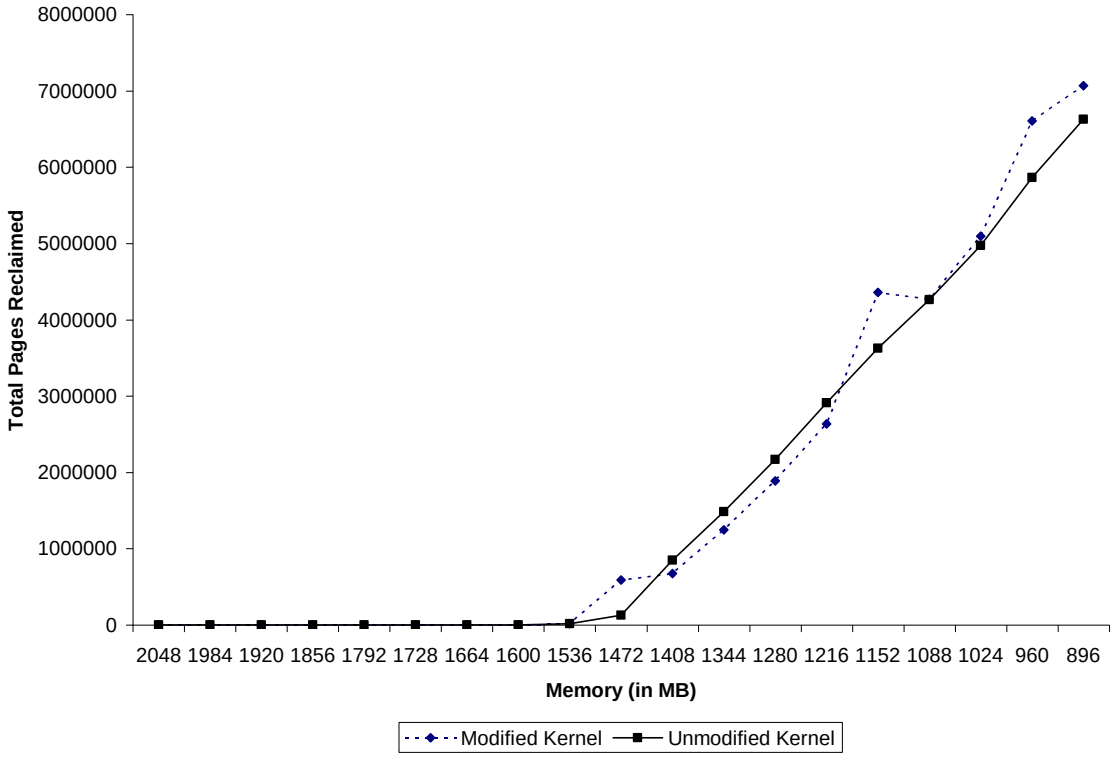

Figure 12: Number of Pages Reclaimed during the runs of HPL benchmark at various memory levels

4.2.3. File System Test

The objective of this test was to measure the influence of memory on the performance of an application that makes extensive use of the file system. For the purposes of this test, a file system consisting of a large number of files was created. The contents of the files were read into a buffer. The throughput was measured as the number of bytes of file data read per second.

A large amount of data stored in files on the disk is accessed by this benchmark. Thus it is expected to use a large number page cache pages to cache the data read from the disk. If the available memory increases, a larger amount of file data can be kept in memory. If any of the cached data is accessed again, the throughput increases since costly disk accesses are avoided. Correspondingly, with a decrease in the memory available for caching, cache misses are expected to increase, leading to a degradation in performance.

The files that were accessed were chosen in 2 different ways:

1. Random: As the name indicates, a file was chosen at random
2. Biased: In this mode, the file is chosen from a user-defined subset of files with a probability p chosen by the user. This mode was supported because in reality files are not accessed uniformly and only a minority of files account for a majority of accesses.

During the experiments, the memory was reduced from 2048 MB to 896 MB. Two banks of size 32 MB each were turned off once every five minutes. After the memory was reduced to 896 MB, two banks were turned on every five minutes until the memory was increased to 2048 MB again. The throughput was sampled once every ten seconds.

In some of the test runs, the files were accessed from multiple threads because in this scenario there is a higher probability that the same file is read multiple times. At the time the banks were being turned off, the threads were put to a dormant state after they finish reading their current file in its entirety. This had to be done as it is not possible to migrate page cache pages that are in use.

The new *proc* interface cache_stats was scanned every ten seconds to obtain the number of page cache hits and page cache misses in the past epoch. This was recorded along with the throughput. The page cache statistics obtained was at a system wide level and not specific to the process we were concerned with.

We expected a low performance as soon as the experiment was started as all accesses would result in page cache misses. As the page cache is filled with data read from the disk, we expected the hit ratio to increase thus leading to an increase in performance. We expected the hit ratio to be higher when the working set size was small.

The following graph shows the results obtained when files were chosen at random for one thread reading the contents of the files. The sum of the sizes of all the files that could have been accessed by the test is about 1500 MB.

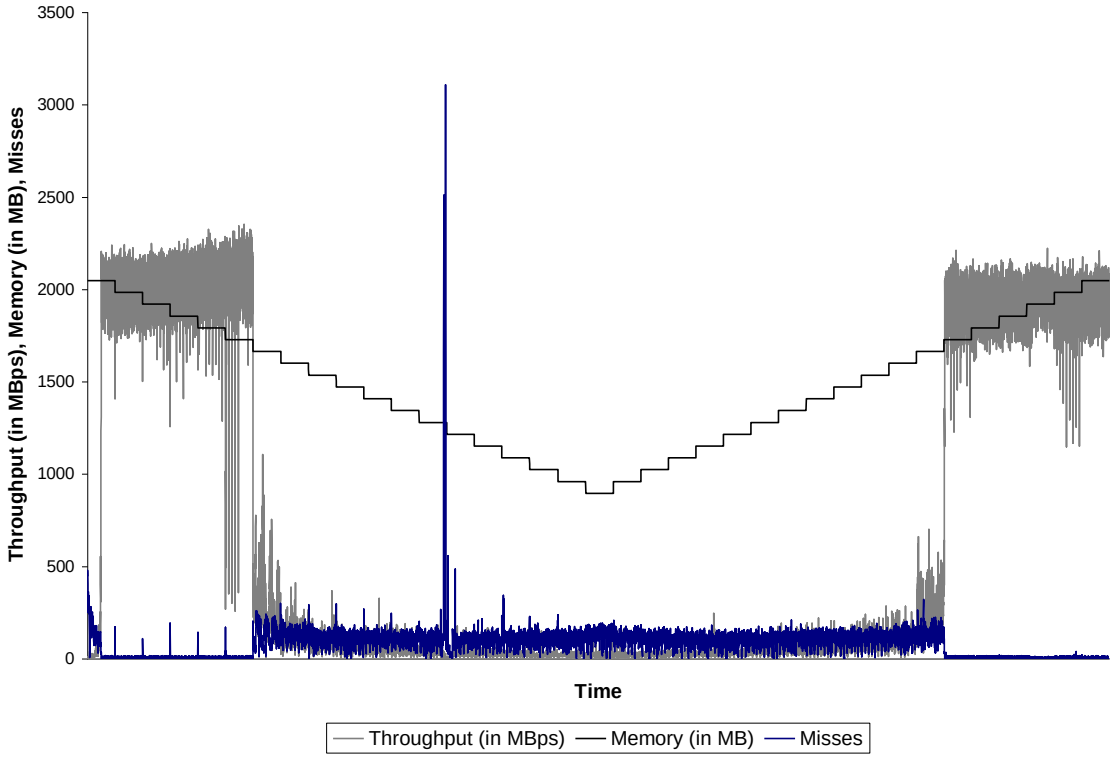

Figure 13: Throughput achieved and misses recorded during the FS test (random mode) at various memory levels

Initially when the experiment was started, no file data was in the page cache and hence all accesses resulted in cache misses. The cache was gradually filled with the file data and any subsequent accesses to the data were served from the cache as shown by the increase in the number of cache hits. The cache hits prevented the disk access thereby increasing the throughput. As the memory was further reduced, the size of the page cache shrank, resulting in a larger number of cache misses and disk accesses, thereby decreasing the throughput.

As the working set size was about 1500 MB for this experiment, when the memory available for caching was greater than 1500 MB most if not all of the accesses result in hits as the entire data can be cached. When the memory available for use as page

cache falls below 1500 MB, not all data can be cached. With the files chosen at random, cache misses occur and the higher costs associated with the disk accesses result in an abrupt drop in performance.

The momentary degradations in the performance as seen in the graph are because the threads are made dormant when the banks are being turned off. Another reason is the reclamation activity that is triggered when a bank is being turned off as pages are needed to copy the contents of the bank being turned off to. This leads to cache misses (as seen in the graph) and hence poor performance.

The graph also shows the variation in performance as memory was turned on periodically. As before, 2 banks of size 32 MB each were turned on every five minutes and the performance was sampled every 10 seconds. The cache misses and hence the number of disk accesses increases with an increase in memory leading to a higher throughput.

With this test we have tested the system call that is used to turn on the memory that was previously turned off. These results show that our implementation of the system call indeed turns memory back on, which is then being used to cache file contents. This results in a throughput increase once enough memory is used to cache the file data.

The following graph shows the results obtained for a similar setting but with ten threads reading the file data.

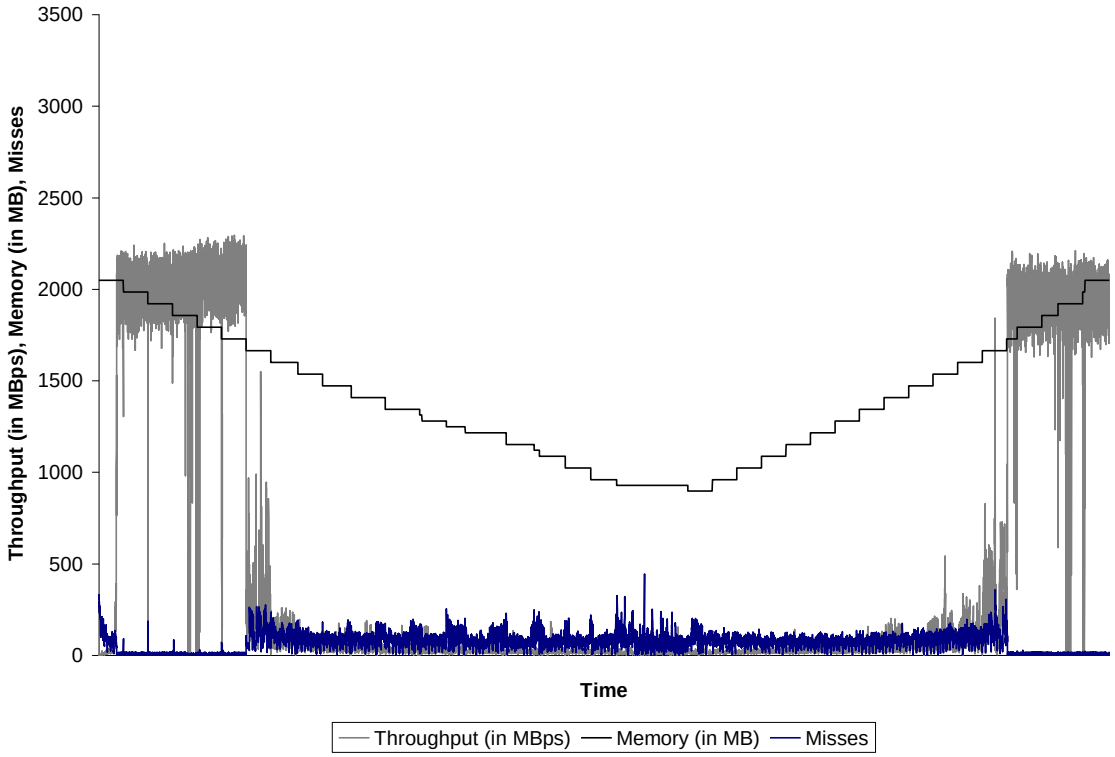

Figure 14: Throughput achieved and misses recorded during the FS test (random mode, multiple threads) at various memory levels

These results are similar to the results obtained in the previous case, except that it was harder to turn memory off as page cache pages were accessed more frequently than before.

The following graph shows the results obtained when the files were chosen in a 'biased' mode. Ten threads were reading the contents of the files. The sum of the sizes of all the files that could have been accessed by this test was around 1500 MB. In the biased mode, 80% of the time a file was chosen from a fixed set of 150 files and the remaining 20 % of the time the entire set of files was considered. The sum of the sizes of the files in the fixed set of 150 files was about 50 MB. As the size of the data that was accessed frequently was small (~50 MB), we expected the performance to better than the performance obtained in the previous cases at lower memory levels.

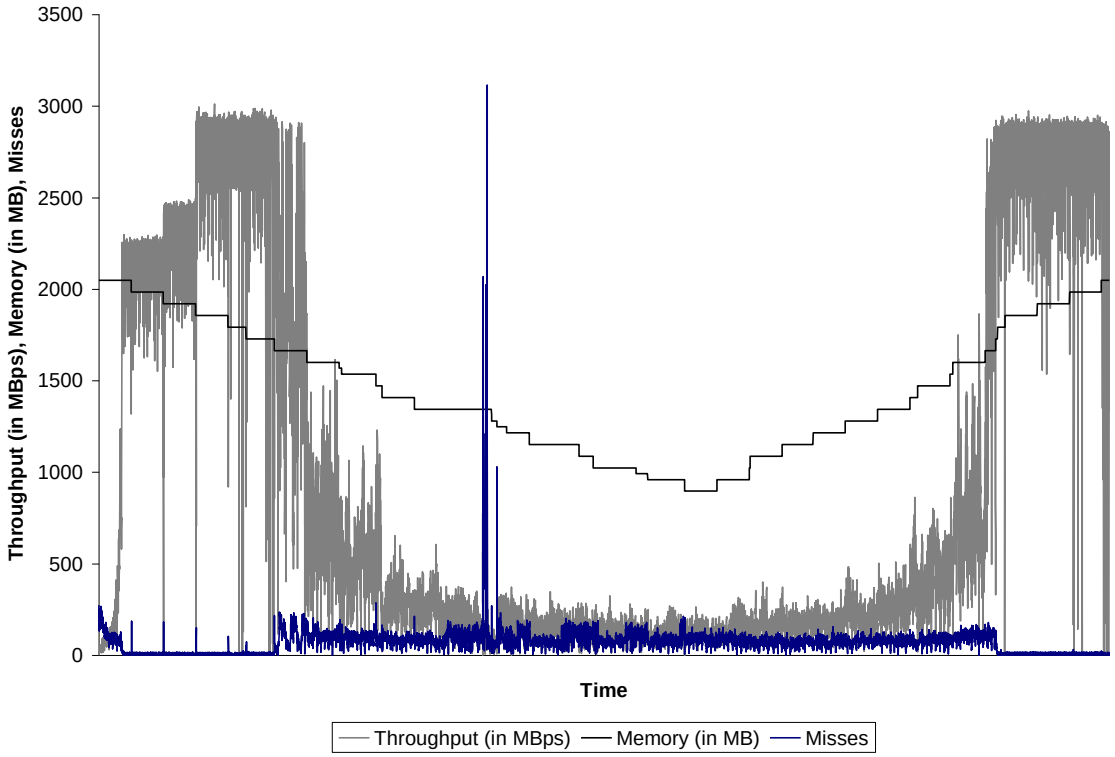

Figure 15: Throughput achieved and misses recorded during the FS test (biased mode, multiple threads) at various memory levels

As seen from the graph, the performance was better than the previous cases, even at low memory levels. Like before there was difficulty turning off the banks as ten threads were accessing the files.

4.2.4. SPEC JBB 2005

The objective of this experiment was to study the effects of the size of the JVM heap on the applications running on it. We chose the JVM heap size, as opposed to the physical memory size, because we assume an administrator would not run a JVM with a heap size larger than the physical memory size available.

Objects dynamically allocated by Java applications are allocated on the JVM heap. The heap contains objects that can be referenced and objects that cannot be referenced. The objects on the heap that can be referenced comprise the live heap. The

objects on the heap that cannot be referenced anymore are garbage. The space allocated for these objects has to be reclaimed periodically because the size of the live heap along with the space occupied by the dead objects cannot exceed the maximum specified size of the entire JVM heap. The space occupied by dead objects is reclaimed automatically by the garbage collector, relieving the programmer from the burden of having to free the objects explicitly in the code. The job of the garbage collector is to identify the objects that cannot be referenced by the program anymore and to free the space used by those objects.

Applications cannot make progress during the time garbage is being collected. Hence garbage collection affects the performance of the applications. The naïve approach to collecting garbage is to identify all live objects and reclaim objects that cannot be referenced from the Java application. This approach takes a long time and hence the performance of the application degrades. In order to efficiently reclaim garbage, today's garbage collectors use the technique known as generational collection. Objects are grouped into different generations or memory pools, based on their age and garbage is collected from each pool. Objects are moved between generations based on their age. More information about such garbage collectors can be found in [20].

The frequency at which GC has to occur depends on the memory requirements and the maximum size to which the JVM heap can grow. When the size of the heap available to the JVM is higher than the memory requirements of the applications, the garbage does not have to be collected frequently. When the size of the live heap is close to the memory requirements of the application, the performance is expected to degrade [21]. Thus it is expected that as the size of the heap is increased, the performance of the applications increases.

Spec JBB is a benchmark designed to measure the performance of server side Java applications. Among other metrics it measures the performance of the JVM (Java Virtual Machine), JIT (Just-In-Time) compiler and garbage collection. We were primarily interested in the garbage collection aspects of the benchmark. Though the performance

metrics reported by it do not solely depend on the garbage collection, GC is a factor that has a major influence on the measured metrics.

Spec JBB emulates a three-tier client server architecture in which clients are the customers placing orders at the warehouses. The clients are represented by threads. The middle tier is where the business logic is implemented. The benchmark does not make use of an external database to store the warehouse data. Instead, it uses hash maps and trees to store the data pertaining to the warehouses. The following diagram, adapted from [22], shows the architecture of SPEC JBB.

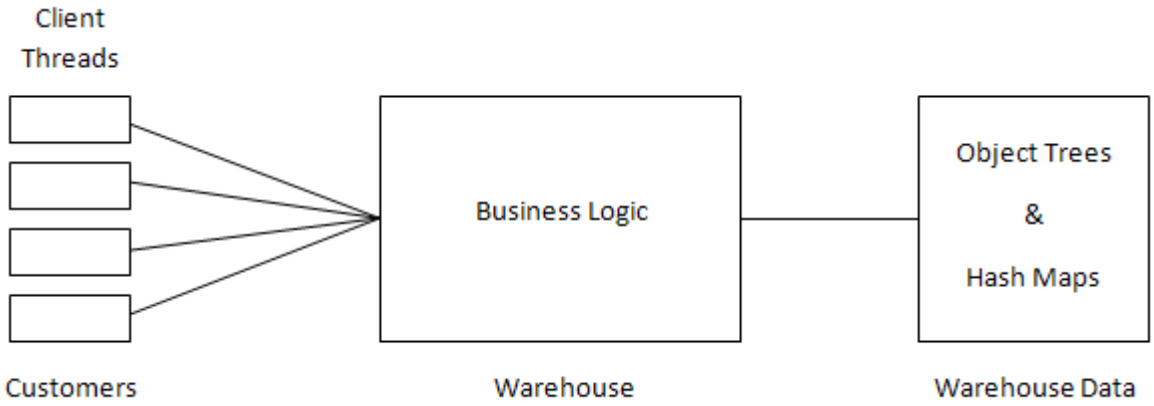

Figure 16: SPEC JBB Architecture

Spec JBB resides in the memory and does not perform any disk I/O operations. Hence it is reasonable to assume that memory pressure is a major factor affecting its performance.

The benchmark measures the number of transactions completed per second as well as the number of transactions completed per second per JVM when more than one instance of JVM is used at the same time. Since our experiments were performed with only one JVM, we only consider the first metric.

Unlike the previous experiments, physical memory was not turned off for this experiment. As the goal of the experiment was to determine the impact the size of the heap had on the performance of the applications, the size of the heap was varied and not

the physical memory available. The benchmark was run on the modified kernel booted with 2048 MB of memory and the maximum size to which the heap could grow was varied. In each run of the experiment, the initial size of the heap was set to be equal to the maximum allowable heap size for the run.

Each run of the experiment lasted about 240 seconds. 12 warehouses were simulated in every run. Each warehouse's data requires about 25 MB of memory, which comes from the heap. Thus the live heap size was about 12 * 25 = 300 MB. The allowed maximum heap size was varied between 320 MB and 1920 MB.

The following graph shows how the performance of the benchmark varies for different heap sizes.

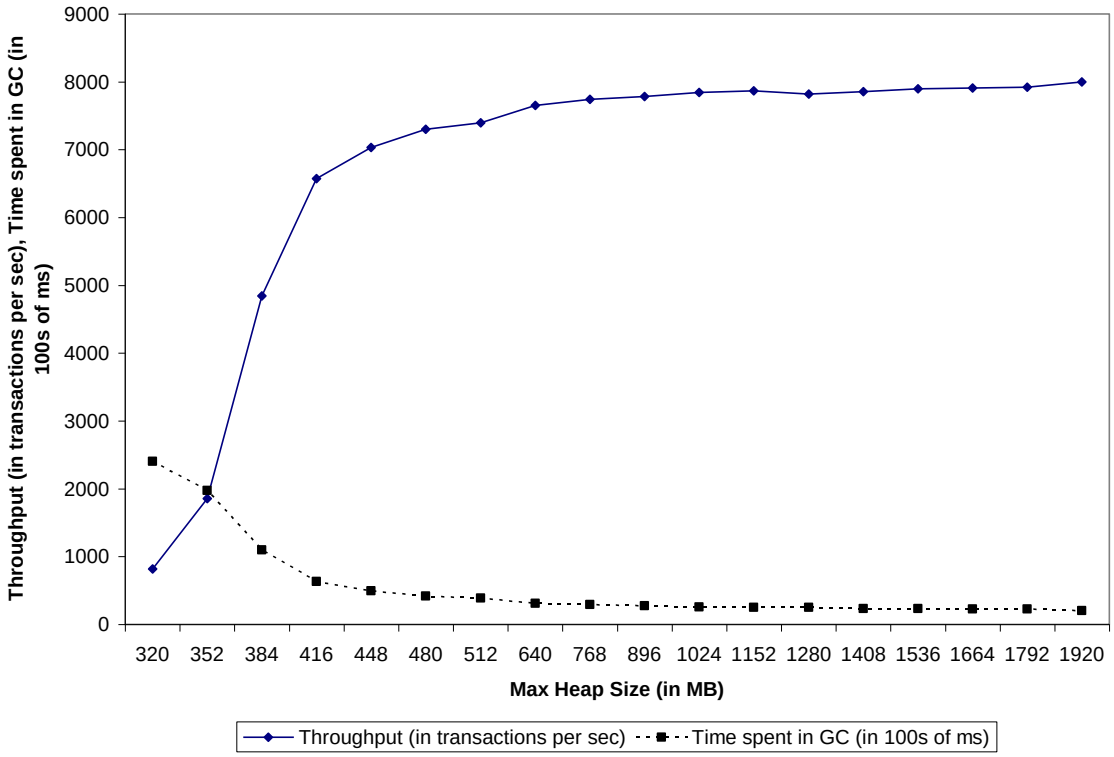

Figure 17: Time spent in GC and the throughput for SPEC JBB 2005 at various heap sizes

The size of the live heap was the same in all the runs as the number of warehouses was not varied between the runs. In the initial runs, with the maximum size of the heap

just above the base requirement of 300 MB, the garbage collector is invoked frequently to reclaim garbage. Thus out of the 240 seconds of simulation, a substantial amount is spent on garbage collection, thereby reducing the time available for the actual simulation thereby resulting in a low throughput. But as the size of the heap grew larger and the demand remained the same, the pressure on the memory was lesser, thereby leading to less frequent garbage collection and more time for the simulation, resulting in higher throughput.

As can be seen from the graph, the increase in the heap size leads to higher throughput initially. But as the heap size is increased beyond what is required, the increase in throughput diminishes. This is because garbage collection involves marking the live heap and reclaiming those parts of the heap that cannot be referenced. Since the size of the live heap is the same irrespective of the size of the heap, the time taken to mark the live heap is the same. Thus it can be seen that by sizing the heap appropriately, performance can be maximized.

4.2.5. Predictions of Memory Requirements

The experiments we have performed could be broadly classified into the following 3 categories:

1. Applications that make extensive use of non-disk backed data
2. File system intensive workloads
3. Applications that manage memory on their own

Based on the results we have obtained, we see that for each of these categories the memory requirements can be predicted. Using these results, controllers similar to the one used in [23] can be developed which make decisions about when to turn memory off or on based on the requirements of the applications so that the degradation in the performance is minimal.

Applications that do not use disk backed data perform well when the memory available for use is close to the size of their working sets. As the available memory decreases, swapping activity increases, and thus the performance decreases. If the

available free memory is more than the size of the data set, memory can be reclaimed without much degradation in the performance of such applications.

For applications making extensive use of disk backed data, the performance depends on the amount of memory available for caching the data read from the disk. The results of the experiments illustrate the benefits of caching and show that the performance depends to a very large degree on the relationship of cache size to data size. Thus memory should be sized based on the desired performance.

In the case of applications such as JVMs that manage memory on their own, the application performance is affected by the size of the heap. As long as there is enough physical memory so that the heap itself need not be paged, the performance is solely dependent on the size of the heap. Performance of the applications increases with an increase in the size of the JVM heap [24]. But increasing the size of the heap beyond a point is not accompanied by a corresponding increase in the performance. Thus by optimally sizing the heap the memory, performance can be maximized at the least expense of physical memory.

5. Related Work

Most of the work related to our work has been targeted at power conservation and support for hot pluggable memory rather than investigating the suitability of the existing memory management techniques for such purposes and study of the application performance under memory pressure on such infrastructures.

5.1. Power Conservation

Huang et al have designed and implemented Power Aware Virtual Memory [25] and as the name indicates its goal is to reduce the power consumed by the memory modules by putting some of the memory nodes to lower power operating levels. This work is primarily applicable to RDRAM [26], but could be used with other memory technologies also. All the nodes that have page frames used by a process *i* are called the active nodes of the process *i*. Power is conserved by putting into low power operating modes all nodes other than the active nodes of the currently running process. DLL aggregation and page migration techniques are used to reduce the number of active nodes of a process.

The nodes being put into low power modes can have pages allocated on them and hence the only constraint they had to satisfy is that there should not be any pages used by the current process on the low power mode nodes. Hence their work did not have to deal with the detecting the type of pages or page migration..

Lebeck et al [27, 28] analyze the effect of static and dynamic memory controller policies to transition memory into low power states. Delaluz et al [29] describe a compiler based approach that inserts power state transition instructions into the compiled code. Delaluz et al [30] have designed a scheduler based approach for power management.

5.2. Support for Hot Pluggable Memory

Tolentino et al have developed a system similar to ours to conserve power by turning off memory at run time [23]. This work is also based on logically dividing the available memory into many banks. They classify pages that are easy to be migrated as Easy (E) pages and those pages that are hard to migrate as Hard (H) pages. The H pages are redirected to the lower banks that are assumed to be turned on all the time. They do not handle the migration of as many different kinds of pages as we do. They continuously monitor the memory demands and keep online only the required number of banks to meet the demand and the others are turned off. They have designed a user level daemon that computes the memory requirements based on certain parameters and it makes onlining or offlining decisions. Our work is different from theirs as their system assumes to always have more memory than what is needed at any point of time in order to match the performance on a machine where no memory is turned off.

The Linux developer community has been working on integrating support for hot pluggable memory into the production kernel. The main objective of the work is to provide runtime memory module addition and removal capabilities [7, 8]. This feature benefits machines running for a long time, such as servers, as reboots are avoided.

6. Conclusions and Future Work

The results of the experiments show that our infrastructure indeed reduces and increases the physical memory available for use. As future work, the modifications made to the buddy allocator that require it to be aware of the bank the allocation request is being satisfied from, could be made more efficient. This increase in efficiency could be achieved by using a more efficient implementation for when the allocator has to distinguish between low banks and non-low banks to make sure that the allocation is satisfied from the appropriate bank. Secondly, some of these problems related to reclamation activity could be fixed with the use of 64 bit architectures. These architectures do not have ZONE_HIGH and hence the distinctions between pages are reduced thus leading to lesser overhead when allocating pages.

During the experiments we have not been able to turn off memory for some workloads, because the banks being turned off contained pages that are either frequently used or being used for a long time. Our infrastructure could be modified to keep track of the usage of those pages that prevent these banks from being turned off and capturing them when they are not being used. Another possible extension to our work is to add support for discontinuous *mem_map* as mentioned in [7] and [8], so that adding more memory than what was present at the boot time could be made possible.

Though we have not experimented with many different kinds of applications, the results obtained for the types we have considered show that the results can be used to predict the memory demands of the applications. The results of the experiments can be used to develop application specific monitors that track the memory requirements of an application and make decisions about turning memory off or on.

As we can see from the results of the experiments, the design of the existing memory managers allows them to be modified to support the ability to vary the memory at runtime. But the requirement to have a one-to-one mapping between the kernel address space and the physical memory makes it hard to migrate certain kinds of pages

used by the kernel. Thus the banks containing those non-migratable pages cannot be turned off.

The ability to support the runtime variation of memory was not one of the design goals of the current design. Hence the performance obtained on a kernel modified to support run time variation of memory was not as good as the performance obtained on an unmodified kernel booted with an equal amount of memory. An important reason for the degradation in performance is the partitioning of memory into ZONE_NORMAL and ZONE_HIGH due to shortage of virtual addresses on the 32 bit architecture. Linux on 64 bit architectures is expected to be better suited for the purposes of varying the memory at run time.

7. References

[1] R. R. Ricardo Bianchini, "Power and Energy Management for Server Systems," *IEEE Computer*, vol. 37, pp. 68-74, 2004.

[2] K. R. C. Lefurgy, F. Rawson, W.Felter, M. Kistler, T. Keller, "Energy Management for Commercial Servers," *IEEE Computer*, vol. 36, pp. 39-48, 2003.

[3] C. S. E. a. A. R. L. X. Fan, "Memory controller policies for DRAM power management," in *International Symposium on Low Power Electronics and Design (ISLPED)*, 2001.

[4] X. F. A.R. Lebeck, H. Zeng, C. Ellis, "Power Aware Page Allocation," in *ASPLOS-IX*, 2000.

[5] B. D. Paul Barham, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, Andrew Warfield, "Xen and the Art of Virtualization," in *ACM Symposium on Operating System Principles*, 2003.

[6] C. A. Waldspurger, "Memory Resource Management in VMware ESX Server," in *USENIX Association Symposium on Opearting System Desgin and Implementation*, 2002, pp. 181 - 194.

[7] M. K. Dave Hansen, Brad Christiansen and Matt Tolentino, "Hotplug Memory and the Linux VM," in *Linux Symposium*, Ottawa, Ontario, Canada, 2004, pp. 287 - 294.

[8] D. H. Joel Schopp, Mike Kravetz, Hirokazu Takahashi, Iwamoto Toshihiro, Yasunori Goto, Kamezawa Hiroyuki, Matt Tolentino and Bob Picco, "Hotplug Memory Redux," in *Linux Symposium*, Ottawa, Ontario, Canada, 2005, pp. 151 - 174.

[9] P. V. Jon Hass, "Fully-buffered DIMM Technology Moves Enterprise Platforms to the Next Level," in *Technology@Intel Magazine*. vol. 3, 2005.

[10] D. P. B. a. M. Cesati, *Understanding the Linux Kernel*: O'Reilly, 2005.

[11] M. Gorman, *Understanding the Linux Virtual Memory Manager*. NJ: Prentice Hall, 2004.

[12] K. C. Knowlton, "A fast storage allocator," *Communications of the ACM*, vol. 8, pp. 623 - 624, 1965.

[13] D. Knuth, *The Art of Computer Programming, Fundamental Algorithms* vol. 1: Addison-Wesley, 1968.

[14] J. Bonwick, "The Slab Allocator: An Object-Caching Kernel Memory Allocator," in *USENIX Summer*, 1994.

[15] P. J. Denning, "The working set model for program behavior," *Communications of the ACM*, vol. 11, pp. 323 - 333, 1968.

[16] R. W. C. a. J. L. Hennessy, "WSCLOCK—a simple and effective algorithm for virtual memory management," in *ACM Symposium on Operating Systems Principles*, pp. 87 - 95.

[17] F. J. Corbato, "A Paging Experiment with the Multics System," 1968.

[18] C. A. R. Hoare, "Quicksort."

[19] R. C. W. A. Petitet, J. Dongarra and A. Cleary, "HPL - A Portable Implementation of the High-Performance Linpack Benchmark for Distributed-Memory Computers."

[20] S. Microsystems, "Tuning Garbage Collection with the 5.0 Java[tm] Virtual Machine."
[21] Y. F. a. E. D. B. Matthew Hertz, "Garbage Collection Without Paging," in *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2005.
[22] SPEC, " SPECjbb2005."
[23] J. T. a. K. W. C. Matt Tolentino, "Memory-MISER: A performance-constrained runtime system for power-scalable clusters."
[24] M. H. a. E. Berger, "Quantifying the Performance of Garbage Collection vs. Explicit Memory Management," in *ACM SIGPLAN conference on Object Oriented Programming, Systems, Languages, and Applications*, San Diego, 2005.
[25] P. P. a. K. G. S. Hai Huang, "Design and Implementation of Power-aware Virtual Memory," in *Usenix 2003 Annual Techinical Conference*, 2003.
[26] "RDRAM," Rambus Inc, 1999.
[27] C. S. E. a. A. R. L. Xiaobo Fan, "Memory controller policies for DRAM power management," in *International Symposium on Low Power Electronics and Design (ISLPED)*, 2001.
[28] X. F. Alvin R.Lebeck, Heng Zeng, Carla Ellis, "Power Aware Page Allocation," in *Architectural Support for Programming Languages and Operating Systems*, 2000.
[29] M. K. V. Delaluz, N. Vijaykrishnan, A. Sivasubramaniam and M. J. Irwin, "DRAM Energy Management Using Software and Hardware Directed Power Mode Control," in *International Symposium on High Performance Computer Architecture*, 2001.
[30] A. S. V. Delaluz, M. Kandemir, N. Vijaykrishnan and M. J. Irwin, "SchedulerBased DRAM Energy Management," in *Conference on Design automation*, 2002.

Appendix A

Some of the new /proc interfaces added and their purposes are mentioned below:

- vmtrack_enabled: When written to, it turns the daemon on/off. When read, reports the current status of the daemon
- vmtrack_status: This read-only file reports the on/off status of the banks
- bank_pages: This read-only file reports the number of free pages on each bank
- low_bank_mark: This is also a read-only file and reports the bank up to which the __GFP_LOWBANK allocations have taken up
- cache_stats: This read-only file reports the number of page cache hits and misses since the system was started up