AN INVESIGATION OF INTEGRATION AND PERFORMANCE ISSUES RELATED TO THE USE OF EXTENDED PAGE SIZES IN COMPUTATIONALLY INTENSIVE APPLICATIONS

A Thesis
Submitted to the Graduate Faculty
of the
North Dakota State University
of Agriculture and Applied Science

By

Matthew James Piehl

In Partial Fulfillment
for the Degree of
MASTER OF SCIENCE

Major Department:
Computer Science

November 2012

Fargo, North Dakota

North Dakota State University

Graduate School

Title

AN INVESTIGATION OF INTEGRATION AND PERFORMANCE ISSUES RELATED TO

THE USE OF EXTENDED PAGE SIZES IN COMPUTATIONALLY INTENSIVE APPLICATIONS

By

Matthew James Piehl

The Supervisory Committee certifies that this *disquisition* complies with North Dakota State University's regulations and meets the accepted standards for the degree of

MASTER OF SCIENCE

SUPERVISORY COMMITTEE:

Dr. William Perrizo

Co-Chair

Dr. Greg Wettstein

Co-Chair

Dr. Jun Kong

Dr. Ben Braaten

Approved by Department Chair:

11/08/12 Dr. Kenneth Magel

ABSTRACT

The combination of increasing fabrication density and corresponding decrease in price has resulted in the ability of commodity platforms to support large memory capacities. Processor designers have introduced support for extended hardware page sizes to assist operating systems with efficiently scaling to these memory capacities. This paper will explore integration strategies the designers of the Linux operating system have used to access this hardware support and the practical performance impact of using this support. This paper also provides a review of common strategies for adding support for this functionality at the application level. These strategies are applied to a sampling representative of common scientific applications to support a practical evaluation of the expected performance impact of extended page size support. An analysis of these results support a finding that a 5% performance improvement can be expected by adding support for extended page sizes to memory intensive scientific applications.

ACKNOWLEDGEMENTS

I would like to thank several people for their support during the development of this work. First I would like to thank Dr. Greg Wettstein for his endless encouragement, guidance, and advice. Without Dr. Wettstein, this work would have not been possible. I would also like to thank the other members of my committee Dr. William Perrizo, Dr. Jun Kong, and Dr. Ben Braaten for their valuable advice. Special thanks to Bryan Mesich of the NDSU Research Support Group for his valuable input and support throughout my time at NDSU. Lastly, I would like to thank my family for their endless encouragement.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF APPENDIX TABLES

LIST OF APPENDIX FIGURES

1. INTRODUCTION

This section provides relevant background discussion on hardware and software systems used to support operating system management of memory. It begins with a brief history of computers and memory followed by a discussion of concepts and technologies relevant to the Linux operating system.

1.1. Directly Addressed Memory

Beginning in 1969, various companies began production of minicomputers. These computers, such as the Data General Nova and the PDP-11 were used for general purpose computing in scientific, educational, and business applications. These 16-bit minicomputers only contained 64 kilobytes of addressable memory [7].

These systems all used what is known as directly addressed memory. Directly addressed memory refers to a system which statically assigns which sections of memory should be used for which purpose and a coordinated policy of access [8].

Directly addressed memory has many limitations. If a system has a limited amount of RAM available to the *Central Processing* Unit (CPU), it may not be enough to run all the programs a user may expect to run at once. For example, the operating system will require a fixed amount of memory and the remaining memory may not be enough to support simultaneous use of applications such as a web browser or a word processor. This would result in the operating system issuing an out of memory error or a refusal to execute an application whose memory requirement could not be fulfilled.

An additional limitation of directly addressed memory is a lack of protection. Memory used by the operating system could potentially be modified by user applications. A programming error in an application could result in total system failure. Limited protection mechanisms, such as segmentation, were the only means to ensure this could not happen.

1.2. Virtual Memory

To address the limitations of directly addressed memory, computer architects implemented the concept of *virtual memory*. Virtual memory is a collection of management techniques [9].

One of these techniques, known as *paging,* allows segments of memory that have not been used recently to be copied to secondary storage such as a hard disk. This will allow the memory that was occupied by an application to be released, thereby allowing that memory to be used to satisfy a request of another application. This process is completely transparent to the application.

Virtual memory also imposes isolation on processes (contexts of execution). Processes cannot address memory not assigned to their context which improves the reliability and security of the system. Virtual memory thus provides a mechanism for increasing the overall security of the system while also reducing the impact of application programming errors.

Figure 1.1 depicts how virtual address can be mapped to multiple storage mediums. It also details how processes are unaware of other processes use of physical memory. It should be noted that virtual memory assumes the memory required by a process will be broken into segments, the importance of these segments will be explained later.

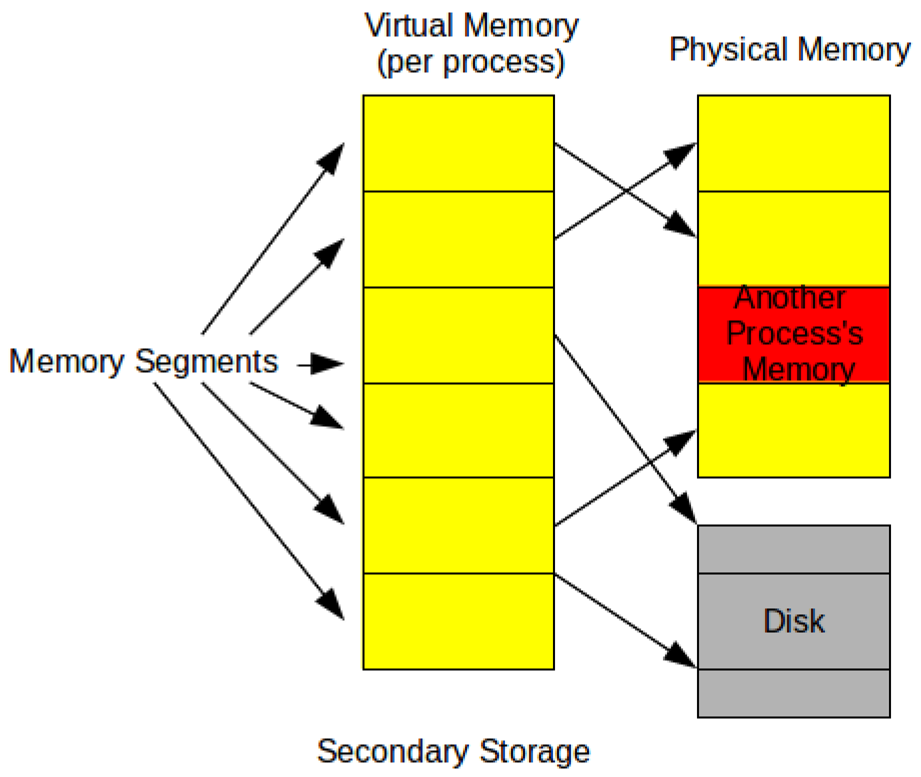


Figure 1.1. Virtual memory

The mapping of virtual memory to its physical location is the responsibility of the memory management unit (MMU) [10]. Since each process has its own independent virtual address space, the same virtual address from independent processes can map to different segments of physical memory. An example of this is provided in Figure 1.2 where the virtual memory address 0x886688 in processes one would result in a reference to physical memory location 0x123456. In process two, virtual memory address 0x668866 would reference physical memory location 0x987654.

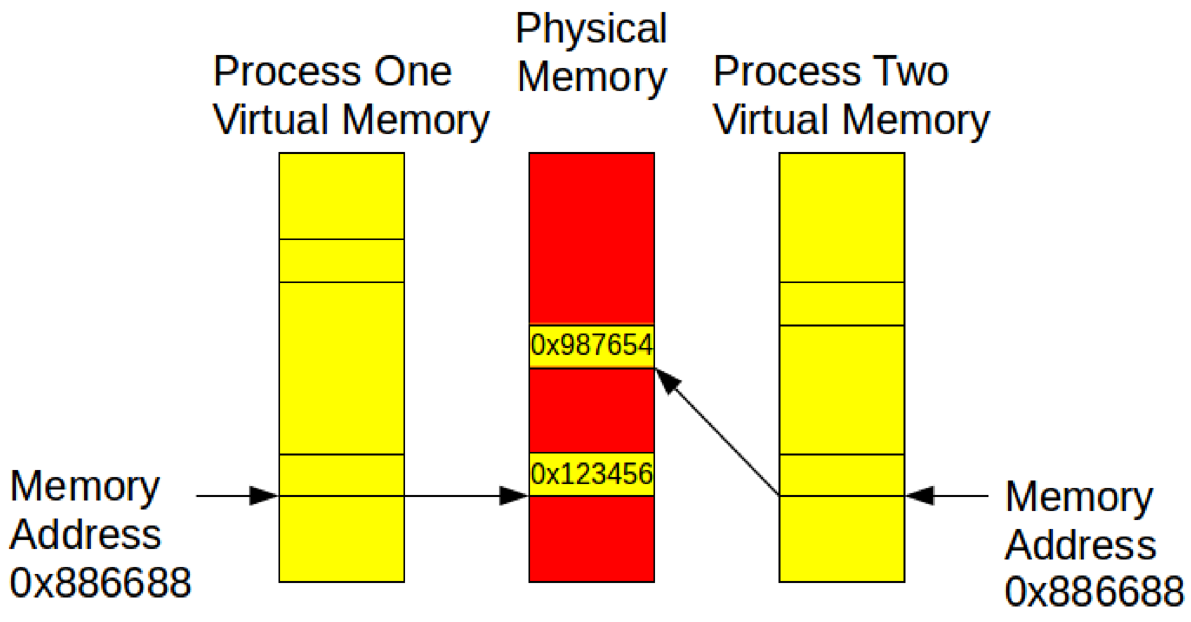


Figure 1.2. An example of independent process address space

The virtual to physical address translation capabilities of the MMU are also used to provide separate processes with mappings to common operating system data structures and shared libraries. Figure 1.3 demonstrates a common three-plus-one mapping strategy where three gigabytes of the virtual address space are used for application specific data with one gigabyte of virtual address space being used to map operating system data. The operating system mapping is common to all processes.

Virtual memory is also used to support optimization strategies such as *copy-on-write* (COW)[25]. With COW, when a process forks or creates a copy of itself, the new process points to the exact same memory as the original process until a write is issued. At this point, a private mapping of memory is created for use by the child process to prevent changes from being visible to other processes. The primary advantages of COW are twofold; the first being multiple copies of the identical memory segments do not need to be maintained with the second being the performance optimization of not needing to create a complete memory image of the parent for each child at the time of process creation.

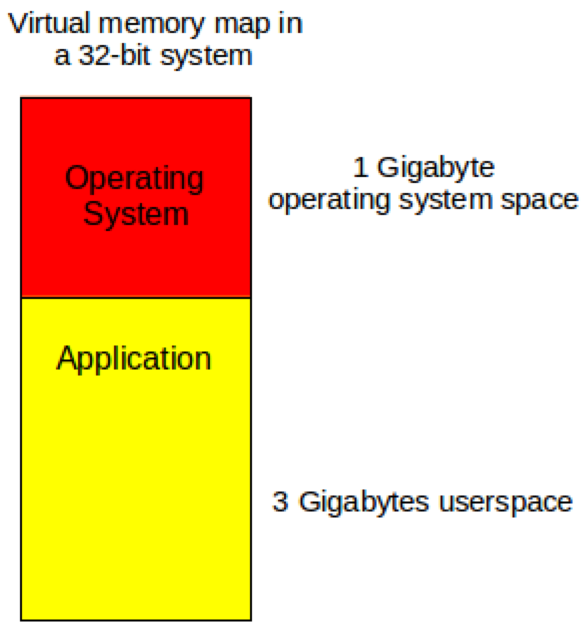

Virtual memory map in a 32-bit system
Operating System
Application
1 Gigabyte operating system space
3 Gigabytes userspace

Figure 1.3. A common three-plus-one mapping strategy

As first noted in Figure 1.1, virtual memory implementations partition the virtual address space into segments which are referred to as *pages*. Pages are blocks of contiguous virtual memory addresses. MMU's have historically used page sizes of 4096 bytes. These pages are the smallest unit of memory allocation performed by the operating system and are the unit in which data is transferred between main memory and secondary storage.

As noted in Figure 1.2, any available physical memory that has not been mapped to a process is capable of satisfying virtual mappings. As applications repeatedly execute, the physical memory map develops discontinuity. While not having a direct performance impact, this fragmentation process reduces the amount of contiguous physical memory which is available, this reduction has important implications with respect to the use and management of extended size pages.

Virtual memory allows a page which does not have a physical memory mapping to be referenced by an application. In the event a program attempts to reference a page not mapped in physical memory, an exception called a page fault is generated. Once a page fault is triggered, the operating system is notified and loads the required page from secondary storage.

The application referencing this page has no knowledge that a page fault occurred or of the underlying mechanism used to implement the fault. This transfer of pages between main memory and secondary storage is known as *paging* or *swapping*. It is the responsibility of the operating system to determine which pages are migrated to secondary storage in response to memory pressure.

As an application executes it will eventually reach a steady state with respect to memory which is being constantly referenced. Once this state is reached page faults, and in turn swapping, should be at a minimum. However, in the event where a program has a steady state which is too large to be supported directly in memory, a phenomenon known as *thrashing* occurs. This results in pages being constantly swapped between main memory and secondary storage which can result in an order of magnitude drop in system performance. This performance degradation results in the common practice of configuring systems with sufficient memory to minimize the need for swapping.

1.3. Caching

Table 1.1 summarizes the hierarchy of access times of various levels of memory in modern computer systems. As noted in the table, accessing memory from cache can result in a 3-6 fold improvement in access latency when compared to satisfying requests from main memory.

Modern processors implement caches in order to increase application performance by decreasing memory latency.

Table 1.1. Hierarchy of access times

Storage Level	Access Time (clock cycles)
Processor Registers	1
Processor Cache	2-3
Main Memory	12
Hard Disk	1,000,000+
Network	1,000,000+

In current architectures, up to three levels of cache are used. The cache is a high speed memory store which is populated by data from a referenced memory location in parallel to its return to the application. Caches hold a limited amount of data at any given time in *static random access memory* (SRAM) based storage.

Various levels of cache are considered to be 'closer' to the CPU with respect to access latency than main memory. The closest cache to the processor, the L1 cache, has very low latency. However, this cache is very small and is typically localized to a single processing core on the CPU. The L2 cache is slower but larger and typically is unified between two processing cores. The furthest from the CPU is the L3 cache. This cache is the slowest but can hold the most data and is unified between every processing core.

When data for a referenced memory location is located in cache, a condition known as a 'cache hit', the cached data is returned which reduces the time required to access the data. If the referenced address is not in the cache, a condition referenced to as a 'cache miss', the referenced value must be returned from main memory. The more requests which are served directly from cache, the better the application will perform.

1.4. Hardware Implementation of Virtual Memory

This paper will describe virtual memory management as implemented on the x86 architecture. The term x86 refers to a group of instruction set architectures (ISA) originating from the Intel 8086 CPU.

Released in 1978, the 8086 was a 16-bit extension of Intel's 8-bit based 8080 processor. The term x86 is derived from the fact that successors to the 8086 also had names ending in "86". Many additions and extensions have been added to the x86 instruction set architecture over the years while maintaining full backward compatibility. The term x86 became common after the introduction of the 80386 which implemented a 32-bit instruction set and addressing [11].

Prior to the 80386, the 80286 processor implemented *segmentation* as a method of memory of protection. With segmentation, memory is divided into partitions that are addressed with a single register (FS or GS) [12]. Partitions are of a fixed or variable size depending on the implementation and may also overlap depending on the segmentation model. Processors today still support segmentation as it is still minimally used by operating systems to support thread and CPU specific data. With the introduction of the 80386, which included a full featured MMU with a flat 32-bit address space, the use of memory segmentation was largely discontinued.

1.5. x86-32 vs. x86-64

Increases in memory subsystem sizes required a further extension to the x86 architecture. This extension is referred to as x86-64 and increases the native word size from 32 to 64 bits. This extension also provides a corresponding increase in virtual and physical address space which allows larger physical memory sizes to be implemented [4].

Given current hardware architectures, a 64-bit address space is so large that physical and virtual address spaces less than 64 bits are implemented. For example, the machine in this investigation implements a 38-bit physical address space and a 48-bit virtual address space. The 38-bit physical address space supports 256 gigabytes of physical memory while the 48-bit virtual address space supports a virtual mapping space of two trillion memory locations.

1.6. The x86 Memory Management Unit

As previously discussed, current processors use a memory management unit (MMU) to implement virtual memory. A MMU is a computer hardware component responsible for handling memory accesses requested by the CPU. Its responsibilities include translation of virtual to physical addresses, memory protection, and cache control.

As noted in the discussion on paging, modern x86 MMUs operate by dividing the virtual address space into pages. In order for the MMU to locate the page of physical memory being referenced, the MMU carries out a virtual to physical address translation process.

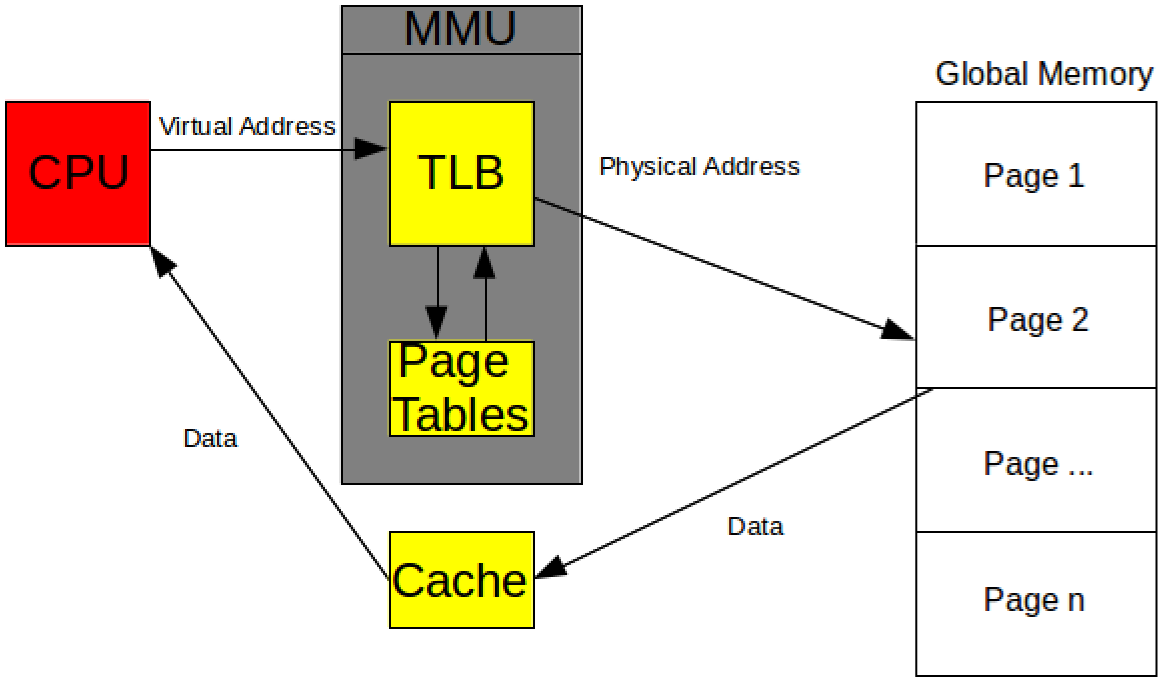


Figure 1.4. The x86 MMU

On the x86 MMU, virtual address translation takes place in three stages [6]. In the first stage, a *global page directory* (GPD) is located. The x86 architecture implements a global control register called the CR3 register, this register is used to store the physical address of the global page directory for the currently executing process. The global page directory is a 1024 element array of page table entries.

The address in the CR3 register is used as the base address of the array. The top 10 bits of the target virtual address are used to compute an offset into the GPD array which will contain a physical address to a *page directory entry* (PDE).

The PDE contains the physical address of a page table which is a 1024 element array, with each array element containing the physical address of a page. The second 10 bits of the target virtual address are used to compute an offset into the page table array to an element referred to as a *page table entry* (PTE). The remaining 12 bits of the virtual address are used to compute an offset to the final physical location of the data for the referenced virtual address from the given PTE. The desired data is returned to the application and the caching layers. Figure 1.5 provides an example of the virtual to physical translation process.

Consider the example where translation of the following virtual address is requested:

<u>1100110011</u> <u>0011001100</u> <u>110011001100</u>

The process specific address contained in the CR3 register is used to determine the location of the PGD. The first 10-bits of the virtual address are used as an index in the PGD to locate the PDE. In the example above, <u>1100110011</u> is used as the offset in the PGD to locate the PDE.

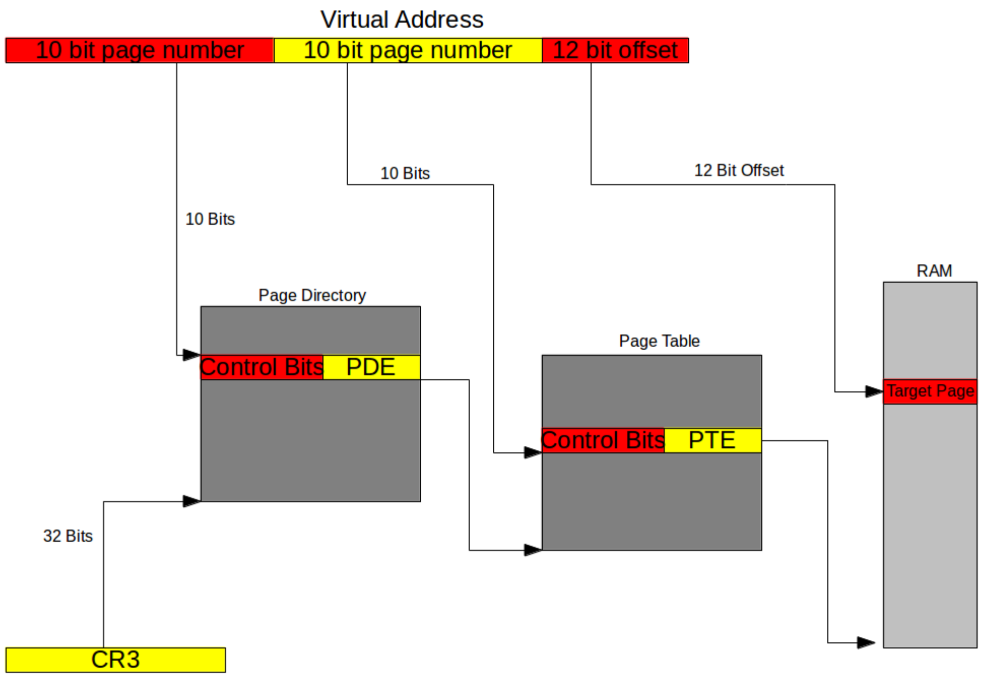


Figure 1.5. The x86 virtual address translation process

The second component of the target virtual address, <u>0011001100</u> represents the offset in the PDE to the address of the physical page. The final component of the virtual address, <u>110011001100</u> is applied as an offset to the address of the physical page.

A 12-bit offset is capable of enumerating 2^{12} or 4096 separate bytes which are found in a standard 4K page. The contents of the physical memory location at the specified offset are returned to fulfill the virtual address request.

As noted in our discussion of memory cache architectures there are significant latency penalties associated with accessing physical memory. The three memory accesses (GPD access, PDE access, page access) required for a 'page table walk', constitutes the physical memory latency required to support a virtual memory translation. This latency translates into tens of CPU cycles [13].

In order to reduce the memory latency imposed by virtual to physical address translation, an additional cache known as the *translation lookaside buffer* (TLB) is implemented [15]. The TLB uses an associative cache architecture to support the direct translation of a virtual address to a physical address. This direct translation eliminates the need for a 'page table walk' and its associated latency.

The cache tag or index is a set of virtual memory addresses. The value associated with the tag is a physical address which the virtual address resolves to. The result of the TLB cache hit is a physical address which is used to resolve the virtual address translation request. In the event of a TLB miss, the MMU must execute the virtual to physical address translation described above.

The effect of a TLB is to reduce the computational and latency costs associated with executing address translation and page table look-ups on every virtual address reference. The TLB exploits the common case of high locality of reference to reduce the memory latency impact of the virtual to physical address translation process.

If a virtual address translation is executed, the TLB must be updated. This update requires a decision to be made as to which current member of the cache must be evicted. The x86 TLB implements a *least recently used* (LRU) eviction strategy in hardware. With LRU, the address that was least recently referenced is discarded and is replaced with the current virtual address translation.

The advantage of this policy is that it has the best chance of releasing a translation that will not be used in the near future, thus reducing eviction pressure. On a standard x86 server machine, the TLB contains 1024 translation entries. Since the TLB is a very limited commodity, eviction pressure on the TLB has a significant impact on execution performance.

Similar to CPU caches, modern TLBs have multiple levels which contain different data. For example, with Intel's Nehalem architecture, a 64-entry L1 data-TLB (dTLB for short) is held 'closer' to the CPU. This TLB contains only data specific translations. An additional cache of related proximity is the instruction TLB or iTLB which is used to hold translations for instructions. Lastly, a unified L2 TLB is used to hold either data or instruction address translations [14].

If application data is to move in or out of memory, it does so through movement to or from a register. Movement from a register to memory is a data store. Movement from memory to a register is a load. These movements are mediated through several different dTLB and iTLB translations. There are performance metrics available for each of these translations.

The dTLB has several statistics of importance with regards to its influence on application performance. The first, a concept known as 'dTLB loads', is the number of translations for addresses associated with data loads. The second important statistic, known as ‘dTLB load-misses’, is instances when a data load address translation could not be resolved by the TLB.

Conversely, a second set of metrics, known as 'dTLB stores', is the number of address translations for data store instructions from CPU registers to main memory. Correspondingly, dTLB store-misses are the number of data store translations that could not be resolved by the TLB.

A final statistic, known as 'iTLB loads', represents the number of address translations needed to load processor instructions. 'iTLB load-misses' contains the number of times the processor instructions failed to be translated.

1.7. Address Space Expansion

Due to the inherent characteristics of modeling and simulation problems in high performance computing, applications often times possess a large memory footprint [1]. This has resulted in a situation where a 32-bit physical address space, supporting the previously mentioned three gigabytes of user based application memory, is insufficient. To remedy this limitation, system architects developed what is known as the *Physical Address Extension* (PAE) mode for x86 processors [16].

PAE mode implements physical address space sizes greater than four gigabytes as long as the underlying operating system supports it. The physical address space increase from 32-bits to 36-bits increases the maximum supported memory from 4 to 64 gigabytes. This extension requires a third level in the hardware page table hierarchy to support the increased physical address size [17].

As previously described, a traditional x86 processor uses a two-level page table with a four kilobyte page table directory with 1024 entries. Enabling PAE mode changes this implementation. Rather than four byte entries in the page table directory and the page tables, each table entry increases in size to eight byte entries (64-bits). The arrays used to implement the virtual to physical address translation remain four kilobytes in size which results in each array containing 512 entries rather than 1024 entries.

Since each page table contains only half as many entries as the original x86 design, an additional level must be added to the page table directory to compensate. As Figure 1.6 illustrates, the CR3 register now points to a *Page Directory Pointer Table* (PDPT) which contains references to four page directories. The four page directory references of the PDPT are sufficient to support the extended physical address space of 36-bits.

In PAE mode processes are still limited to three gigabytes of virtual address space. PAE mode thus enables a system to have a larger number of processes with full memory commitment given the conventional limitation on the amount of memory each process can reference

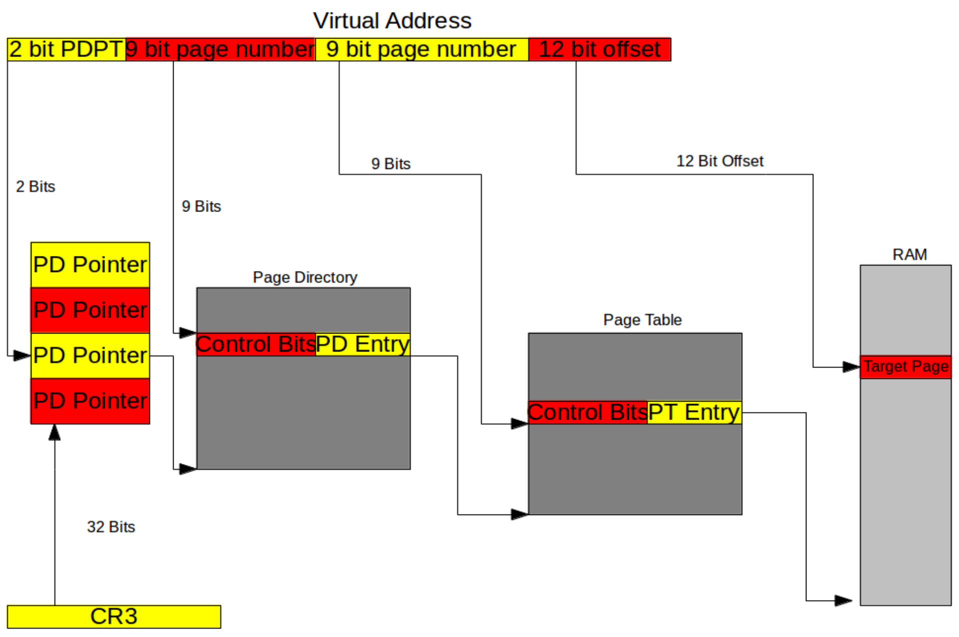


Figure 1.6. The x86 PAE virtual address translation process

With the introduction of x86-64, a further extension to x86 with PAE is introduced. x86_64 provides even larger virtual and physical address spaces than is possible with x86. As previously noted, current x86_64 processors support a physical address space of 48-bits which is 256 terabytes of physical memory.

In order to expand to a 64-bit logical address space, a superset of PAE mode called 'long mode' was introduced. Long mode contains the same support for 32-bit applications in addition to supporting an extended physical address space size.

Instead of utilizing the three level page table hierarchy present in PAE mode, the long mode implementation uses four levels of page tables. The PDPT from PAE mode is extended from 4 to 512 entries. In addition, a fourth level called the *Page-Map Level 4* (PML4) is added which contains 512 entries. Figure 1.7 details this architecture.

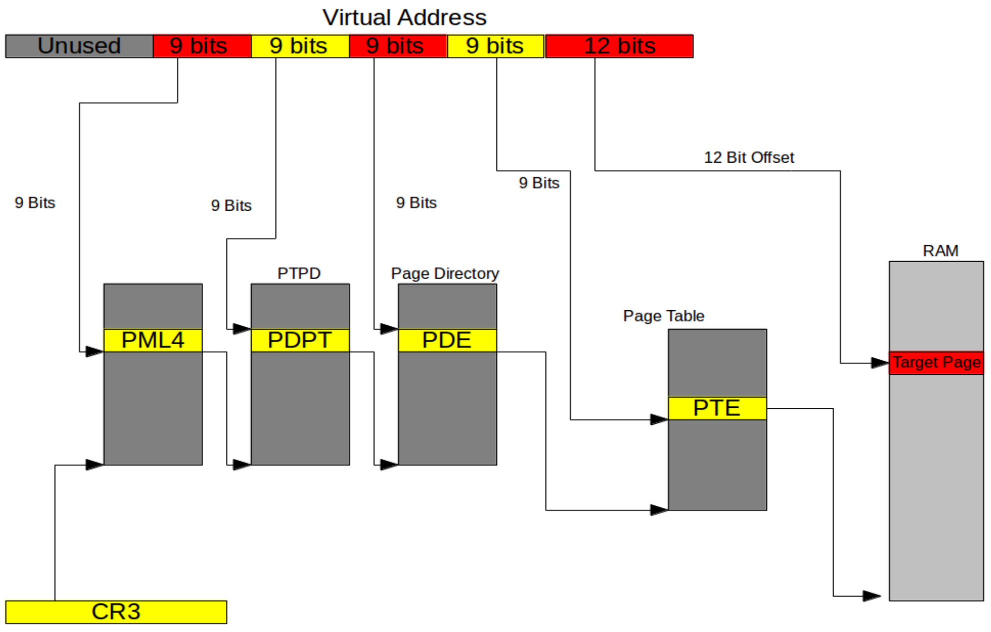


Figure 1.7. The x86-64 virtual address translation process

This page table hierarchy supports up to 48-bits of addressable physical memory. A complete mapping of 4 kilobyte pages on a 48-bit address space would provide the ability to reference 256 terabytes of physical memory. No known systems are currently able to utilize a 48-

bit physical address space. At the time of this writing the SGI UV2 shared memory system, which currently supports 64 terabytes of physical memory, is the largest memory system available [18].

1.8. Linux Virtual Memory Management Architecture

In the Linux Operating System [23], each process resides within its own contiguous virtual address space which translates to a discontinuous physical address space through platform specific hardware mapping systems. Each segment of the virtual address space is managed through a structure known as a V*irtual Memory Area* (VMA) which encapsulates information needed to define and manage a contiguous segment of virtual address space. The complete virtual memory map is represented by a linked list of VMAs.

The VMAs for a particular process can be viewed using two methods. The first method is by viewing the content of /proc/PID/maps. This is the native platform interface to VMA information. The second method is by using the 'pmap' command line tool on a process ID. Table 1.2 demonstrates the output of this command on a standard BASH shell process.

Table 1.2. Example pmap output

0000000000400000	712K	r-x--	/bin/bash
00000000006b2000	40K	rw---	/bin/bash
00000000006bc000	20K	rw---	[anon]
00000000008bb000	32K	rw---	/bin/bash
00000000008c3000	264K	rw---	[anon]

As displayed in Table 1.2, the VMAs for a process all have a size that is an exact multiple of a standard 4 kilobyte page size. At a minimum, a memory map for a process will contain application text, initialized / uninitialized data, the application stack, and any active memory mappings.

A VMA will be created when an application issues a mmap() system call to map memory into a process's address space. If the request can be satisfied, the operating system will grant the mmap() request by creating a new VMA segment large enough to represent the requested allocation.

Once a VMA is allocated by the system, appropriate permissions in the form of system flags are set for that segment of virtual memory. These permission flags will determine how data in the memory area is managed. VMA permission flags are independent from the permissions on each individual page within the memory area. Table 1.3 documents these VMA flags. They can also be found in <include/linux/mm.h>.

Table 1.3. VMA flags

VM_READ	Pages in this area can be read
VM_WRITE	Pages in this area can be written
VM_EXEC	Page in this area can be executed
VM_SHARED	Pages in this area are shared
VM_MAYREAD	Allow VM_READ to be turned off with mprotect()
VM_MAYWRITE	Allow VM_WRITE to be turned off with mprotect()
VM_GROWSDOWN	The VMA grows up
VM_GROWSUP	The VMA grows down

Table 1.3. VMA flags (continued)

VM_READ	Pages in this area can be read
VM_NOHUGEPAGE	Madvise() marked this VMA
VM_DENYWRITE	Deny write attempts to VMA
VM_EXECUTABLE	VMA maps executable file
VM_LOCKED	Pages in VMA are locked
VM_IO	VMA maps a devices I/O space
VM_SEQ_READ	Application will access data sequentially
VM_RAND_READ	Application will not benefit from clustered reads
VM_DONTCOPY	Do not copy VMA on fork()
VM_DONTEXPAND	Do not expand VMA with mremap()
VM_RESERVED	This area must not be swapped out
VM_NORESERVE	Suppress VM accounting
VM_HUGETLB	This VMA is a hugetlb
VM_HUGEPAGE	Madvise() marked this VMA

To manage VMAs, the Linux kernel uses a data structure known as the 'vm_area_struct'. Each vm_area_struct contains the start and end address of a segment of contiguous virtual memory. These segments of memory are non-overlapping and represent a set of virtual addresses generated by an application request to map virtual memory. The vm_area_struct is defined in <include/linux/mm_types.h>.

In order to access a page of memory, a series of data structures must be navigated. The structures referenced depend on the type of memory allocation request which is being requested.

If the application is requesting file backed memory, the value of the 'vm_file' pointer is used to locate a structure of type 'address_space'. The address_space structure contains a radix tree representation of the pages contained in that VMAs address space. Each node in the radix tree contains a page address and a bit field which represents the page's state. These bit fields indicate whether a page is clean, dirty, or locked.

If the application requests anonymous memory mapping, the 'vm_file' variable will be a set to NULL and a new variable called 'anon_vma' will point to a structure of type anon_vma.

Each physical page is represented with a corresponding page structure. This structure is used to keep track of the page's status. This structure is defined in <include/linux/mm_types.h>.

All of the information, including references to the VMAs required to manage the virtual address space of a process are encapsulated within a structure that is referred to as an mm_struct. Included in this structure is a pointer to the linked list of vm_area_structs which define the virtual address space of the process, a red-black tree containing references to the individual vm_area_structs, and a pointer reference to the processes PGD. This structure is found in <include/linux/mm_types.h>.

Inside the mm_struct several key data structures are critical to the operation of the virtual memory manager. The first is a red-black tree (defined in include/linux/rb.h) which contains the virtual start address for every VMA resident within the process. The red-black tree is used in the event rapid address lookups are required. These lookups are important for page fault handling when VMAs must be found quickly. The red-black tree implements O(log n) time complexity for these lookups. The tree is ordered in such a way that lower order addresses are found on the left side of the tree and higher order addresses are located on the right-hand side of the tree.

In order to service a mmap() system call requesting a new memory allocation, the linked list of VMAs is consulted. Within the linked list, VMAs are ordered in ascending virtual memory addresses. The linked list is traversed to determine whether a gap within the virtual address space can be found to satisfy the allocation request. In the event a gap is not found within the virtual address space to satisfy the request, the new VMA is inserted at the end of the list.

The mm_struct is referenced by another structure known as the task_struct. The task structure is responsible for encapsulating all data related to each process within the operating system. It formally declared in <include/linux/sched.h>.

During the process of a context switch the address of the PGD(which is referenced through the mm_struct) is loaded into the CR3 register. As noted in the description of the hardware implementation, the MMU will now have access to the physical memory implementation of the virtual memory map of the process.

In the event an entry for a physical page within the page table is not found, the page fault handling mechanism is invoked. At this point, the virtual page will indicate that the page is flagged as non-resident within page tables and a variety of scenarios may occur depending on the type of page fault.

The most common type of page fault is known as a minor page fault. The Linux page fault handler responds to a variety of events which result in a minor page faults. Table 1.4 details the different types of minor page faults.

Table 1.4. Types of minor page faults

1. Memory region is valid however the page frame is not allocated.
2. Memory region is not valid but is next to an expandable memory region (such as the stack).
3. Page is swapped out but present within swap cache.
4. Page is written to when marked as read-only. (COW page)

The most common type of minor page fault results from a page that is resident in memory at the time of fault generation however, the page does not contain a valid page table entry. In this event, the page fault handler needs to create a PTE which points to the requested page in memory and indicate to the operating system that the page's address is now loaded in the page tables.

This typical minor page fault case on the x86 architecture with Linux involves several steps.

1. The exception handler is alerted to a page fault in a valid memory region. The exception handler will proceed to invoke the architecture dependent function do_page_fault().
2. The do_page_fault() function will proceed to the architecture independent function handle_mm_fault(). This later function will allocate required page table entries if needed before passing control to the handle_pte_fault() function.
3. The handle_pte_fault() establishes a PTE for the requested page and updates the struct page accordingly.

There exists a second type of page fault, known as a major page fault. A major page fault occurs if the requested page is non-resident in physical memory but present in secondary storage. In order to handle this type of fault, the page fault handler must locate a page suitable for eviction, write it out to disk, and move the requested page into memory, followed by updating the page table entries appropriately. Major page faults are not the subject of this investigation as common HPC systems contains sufficient memory to handle the applications being used to avoid swapping.

Major faults are clearly more expensive than minor faults due to the added disk latency. However, minor page faults are not without computational cost. In the event of a minor page fault, an entry into the kernel is required, a requested page must be located, the requested page must be updated, and the page table(s) properly modified. All this must be accomplished while maintaining proper synchronization with other threads modifying and reading the page table tree.

The final step in the page fault procedure is the allocation of the physical memory needed to hold a page. Applications typically request memory in segments, often times these segments are multiple pages in size which the operating system attempts to place in physically contiguous memory. A specialized allocation algorithm is used to carry out physical memory allocation.

The algorithm used to implement physical memory allocation is known as the buddy allocator. This allocator is a two part scheme which combines power-of-two allocation with physical memory coalescing [19].

Allocation of physical memory is managed by segmenting sections of contiguous physical memory into blocks where each block contains a power-of-two number of pages. These blocks are placed in one of ten lists of varying sizes. The lists contain blocks that range in size from one page to blocks that are 2^{10} in size. This list and the number of list occupants can be viewed in '/proc/buddyinfo'. Table 1.5 provides an example output of this file.

Table 1.5. Contents of /proc/buddyinfo

DMA	1	1	1	0	2	1	1	0	1	1	3
DMA32	9	8	10	3	4	8	7	3	4	2	483
Normal	17	2	10	408	193	45	4	1	1	1	255

In order to allocate pages for a request, the allocator must check if the request can be satisfied by a free block in the smallest list possible. For example, if 64 (2^6) pages of contiguous physical memory are requested, the allocator will check the 2^6 list for a free block. If one exists, the allocator can pass the block to the requester. However, if the free block in the 2^6 list does not exist, the allocator will look in the 2^7 list for a free block. If a block is present within this list, the block is split and half is given to the requester while the second half is given to the 2^6 list.

In the event the 2^7 list is empty, the allocator proceeds to the 2^8 list where it further divides blocks to fill the previous lists and grant the allocation request. If no list contains a free block, a memory allocation error is reported.

Once the requester has finished using a block, the Kernel will attempt to coalesce free buddy blocks of size S to size 2S. In order to merge two blocks, several points must hold. Both buddy blocks must be the same size and both blocks must be adjacent to each other in physical memory.

1.9. Extended Page Sizes

As previously mentioned, standard four kilobyte pages impose a memory overhead of four kilobytes of physical memory per page table. This, coupled with the operating system data structure overhead required to manage the virtual memory, constitutes the physical memory overhead imposed by the virtual memory abstraction.

Consider the following example; a 16 gigabyte web server application handling 500 concurrent connections will be using 4,194,304 four kilobyte pages. The page table memory consumption to support these pages is 16 megabytes per process.

This results in a requirement to dedicate 8000 (16 megabytes * 500 processes) megabytes of memory to support the system page tables. Since the TLB can only hold a maximum of 1024 PTE translations, the eviction pressure of using standard sized pages on large memory configurations becomes apparent.

Table 1.6 summarizes VMA memory overhead as a function of physical memory size for four kilobyte pages across a spectrum of physical memory sizes.

Table 1.6. VMA memory overhead

	1GB	10GB	100GB	1000GB
4Kb Page Tables Required	256	2560	25,600	256,000
Total Page Table Entries	262,144	2,621,440	26,214,400	262,144,000

Processor architects have introduced extended size pages to address memory consumption and TLB cache pressure associated with standard page sizes. This architecture allows multiple page sizes to be used simultaneously due to the potential drawbacks and penalties imposed by using a single page size.

Table 1.7 illustrates various hardware architectures and the variety of page sizes available.

Table 1.7. Page sizes for various architectures

Architecture	Standard Page Size	Extended Page Size
x86	4Kb	4M and 2M in PAE mode
ia64	4Kb	4Kb, 8Kb, 64Kb, 256Kb, 1M, 4M, 16M, 256M
ppc64	4Kb	16M
sparc	8Kb	-
arm	4Kb	64Kb

Extended page sizes provide several benefits. The first of which results from the effects of decreasing the number of page table entries required to cover a processes memory map. Use of extended pages also reduces the amount of physical memory required by the previously described data structures.

Lastly, by utilizing extended pages, the overall amount of memory covered by entries within the TLB increases. For example on x86, a 1024 entry TLB using two megabyte extended page sizes would allow the TLB to translate two gigabytes of virtual memory. In contrast, the same size TLB would only allow four megabytes of virtual memory translations using four kilobyte pages.

The Linux operating system presents extended pages in two forms. The first form is a statically assigned pool of extended size pages allocated by a system administrator before an application is run. This pool is used to service extended page size allocation requests userspace applications. The second form is a transparent model where the operating system transparently allocates extended pages based on application demand for large physical memory segments during execution.

The statically assigned extended page size pool, referred to as 'hugetlbfs' within Linux, was the first attempt at offering extended size page support. 'hugetlbfs' or 'huge translation-lookaside buffer file system', first developed by Dr. Mel Gorman, is a collection of techniques to access extended page size mappings[24]. These techniques include shared memory, a pseudo RAM based file system, and anonymous mapping of memory backed by extended page sizes.

Before selecting an access method, support for extended pages has to be enabled within the Kernel. A system administrator must configure the extended size page pool which will be available for access. Various Linux Kernel configuration operations are available to configure this support.

The pseudo file /proc/meminfo contains information regarding the current number of the 'hugetlbfs' pages within the extended size page pool. This pseudo file also contains information regarding the free, reserved, surplus pages, and default extended page size. Table 1.8 provides an example of this file.

Table 1.8. Contents of /proc/meminfo

MemTotal:	16458212 kB
MemFree:	186656 kB
Buffers:	59344 kB
Cached:	15246344 kB
SwapCached:	0 kB
Active:	3730604 kB
Inactive:	11917232 kB
Active(anon):	305416 kB
Inactive(anon):	42080 kB
Active(file):	3425188 kB
Inactive(file):	11875152 kB
Unevictable:	32408 kB
Mlocked:	14012 kB
SwapTotal:	2097148 kB
SwapFree:	2097148 kB
Dirty:	0 kB
Writeback:	0 kB
AnonPages:	374652 kB

Table 1.8. Contents of /proc/meminfo (continued)

MemTotal:	16458212 kB
Mapped:	16576 kB
Shmem:	216 kB
Slab:	528820 kB
SReclaimable:	509328 kB
SUnreclaim:	19492 kB
KernelStack:	1696 kB
PageTables:	5420 kB
NFS_Unstable:	0 kB
Bounce:	0 kB
WritebackTmp:	0 kB
CommitLimit:	10326252 kB
Committed_AS:	2866592 kB
VmallocTotal:	34359738367 kB
VmallocUsed:	302460 kB
VmallocChunk:	34359431323 kB
AnonHugePages:	317440 kB
HugePages_Total:	10
HugePages_Free:	10
HugePages_Rsvd:	0
HugePages_Surp:	0
Hugepagesize:	2048 kB

Table 1.8. Contents of /proc/meminfo (continued)

MemTotal:	16458212 kB
DirectMap4k:	6144 kB
DirectMap2M:	16766976 kB

The following values within this file are the important parameters with regards to extended page sizes:

- HugePages_Total: Size of extended page pool.
- HugePages_Free: Number of unallocated extended pages
- HugePages_Rsvd: Extended pages reserved by application but have not been populated
- HugePages_Surp: Number of surplus extended pages currently overcommited
- Hugepagesize: Size of extended pages for this system

The number of available extended pages is set by writing values to the /proc/sys/vm/nr_hugepages pseudo file. This pseudo file indicates the number of persistent extended pages in the operating system extended page pool. Persistent extended pages are returned to the pool when a task has finished using them. A system administrator dynamically adds or removes extended pages from the pool by changing the value in the 'nr_hugepages' file.

Static allocation of extended pages has a number of limitations. Statically assigned extended pages are reserved within the operating system, as a result, the reserved memory may not be used for any other purpose. An architectural limitation of this model is that extended pages cannot be swapped to secondary storage should memory contention arise.

Extended page allocation also depends on the presence of contiguous physical memory. Without sufficient contiguous physical memory, an allocation request cannot proceed. The longer a system runs, the greater chance for memory fragmentation to develop. This fragmentation decreases the chance of a successful allocation.

Because of this fragmentation effect, system administrators should specify extended page pool size at system boot time when there is the greatest chance of allocating large segments of contiguous physical memory.

Once the system administrator has reserved an extended page pool, user applications may access extended page mappings using one of the following three methods:

1. A shared memory system call.
2. An anonymous memory map.
3. A pseudo filesystem.

The first method uses statically allocated extended pages accessed through a traditional shared memory system call. The shmget() system call is passed the SHM_HUGETLB flag to request that the memory allocation be managed using extended page size rather than a standard four kilobyte page.

Although simple to use, this method has a drawback in that it only supports the default extended page size for the system it is running on. This means architectures that support multiple extended page sizes, such as IA64 and PPC, are forced to a single page size.

Figure 1.8 is an example of using a shared memory system call backed by extended sized pages.

```
#include <stdlib.h>
#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <mman.h>

int shmid;
char *shmbuffer;
shmid = shmget(2, 1024*1024*128, SHM_HUGETLB | IPC_CREAT | SHM_R | SHM_W);
shmbuffer = shmat(shmid, 0, 0);
```

Figure 1.8. A shared memory code sample

The variable 'shmbuffer' now points to a 128 megabyte segment of memory which will be mapped by extended size pages.

The second method is the use of the “hugetlbfs” pseudo filesystem. This method requires a RAM-based filesystem be mounted prior to applications requesting allocations with extended page size backings.

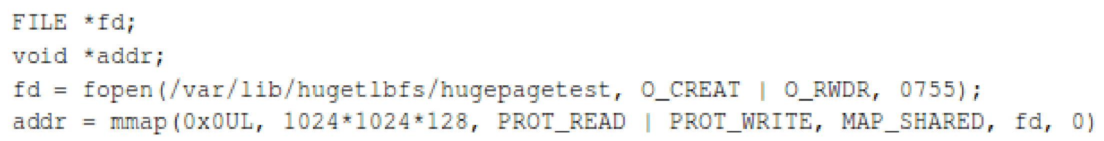
The size of the extended page which will be used for mappings is configured by a system administrator when the filesystem is first mounted. The extended page size selected during the mount must be supported by the underlying architecture. An example 'mount' command is provided below in Figure 1.9.

```
mount -t hugetlbfs none $HOME/fs/hugetlbsfs -o pagesize=16M
```

Figure 1.9. An example hugetlbfs mount command

The command in Figure 1.9 creates a RAM-based file system which uses an extended page size of 16 megabytes.

Once the filesystem has been mounted, the mmap() system call is used on a generic file-descriptor obtained by opening a file on the filesystem. Figure 1.10 provides an example:



```
FILE *fd;
void *addr;
fd = fopen(/var/lib/hugetlbfs/hugepagetest, O_CREAT | O_RWDR, 0755);
addr = mmap(0x0UL, 1024*1024*128, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0)
```

Figure 1.10. An example mmap() binding to a pseudo filesystem

Memory referenced by the 'addr' variable will be mapped with the extended page size specified to the pseudo filesystem.

The total size of all files mapped on the filesystem cannot exceed the number of extended pages allocated in the extended page pool.

The final method is an anonymous memory mapping obtained with the mmap() system call. Anonymous memory mappings are not backed by any file. As of Kernel version 2.6.38, the flags MAP_ANONYMOUS and MAP_HUGETLB can be passed to mmap() to receive a memory allocation managed by extended page sizes. This strategy provides a simpler method to handle general memory allocation without the special requirements imposed by the prior methods. Figure 1.11 provides an example mmap() extended page allocation.

```
char *mmapbuffer;

      mmapbuffer = mmap(NULL, 1024*1024*128, PROT_READ | PROT_WRITE,
            MAP_PRIVATE | MAP_ANONYMOUS | MAP_HUGETLB, 0, 0);
```

Figure 1.11. An example anonymous mmap()

The variable 'mmapbuffer' references 128 megabytes of virtual memory managed by extended size pages.

Many applications simply need a single memory buffer backed by extended page sizes. In order to provide an API for this functionality, libhugetlbfs was developed by Dr. Mel Gorman. [39] Libhugetlbfs provides the get_hugepage_region() and get_huge_pages() which implement this functionality.

The get_huge_pages() function is primarily used for the development of custom allocation schemes and is not a suitable replacement for malloc(). The size parameter of this function is required to be a multiple of the default extended page size.

The get_hugepage_region() function is used by applications that want to allocate large segments of memory which are not precise multiples of the extended page size. This function can also fall back to using small pages if needed making it a suitable replacement for malloc() in most cases. Figure 1.12 provides an example allocation using this method.

```
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>
#include <hugetlbfs.h>

#define LENGTH ((2097153))

#define ADDR (void *)(0x0UL)
#define SHMAT_FLAGS (0)

int main(void)
{
  size_t dsize;

  unsigned long i;
  char *shmaddr;

  shmaddr = get_hugepage_region(LENGTH, GHR_DEFAULT);

  printf("Starting the writes:\n");
  for (i = 0; i < LENGTH; i++) {
    shmaddr[i] = (char)(i);
  }
  printf("\n");

  free_hugepage_region(shmaddr);

  return 0;
}
```

Figure 1.12. An example libhugetlbfs allocation

There are several challenges with associated with the use of statically assigned extended pages. One of the most fundamental challenges is that support is not transparent to the application. The application programmer must introduce support for extended page size mappings within the application. This presents a challenge to application programmers who are not necessarily experienced in the implementation of extended page size backings. Additional application development time is required to integrate the appropriate APIs to request extended page size mappings.

In addition to these issues, support from system administration staff is required to properly configure support for extended size pages at the operating system level. There also must be an agreement between administrators and developers with respect to the amount of memory assigned to the extended page pool.

Transparent huge page support has been introduced in recent Linux kernels to address the deficiency of statically assigned extended pages. *Transparent Huge Pages* (THPs) are implemented by allocating extended size pages mappings whenever possible to any application that requests a memory allocation greater than or equal to the default system extended page size.

The THP system has a number of advantages over the static allocation system. The primary advantage arises from the fact that no changes are needed at the application level to gain the benefits of extended page size allocations.

An additional benefit is support for coalescence of standard size pages to extended size pages. This functionality is implemented through a Kernel monitoring thread called 'khugepaged'. This thread periodically attempts to consolidate groups of contiguous standard sized pages into a smaller number of extended size pages.

The final advantage is support for migrating (swapping) memory backed by THPs to secondary storage. To achieve this functionality, transparent pages are split into corresponding smaller, four kilobyte pages which are then handled by the traditional swapping mechanism.

An extension to the madvise() API allows application programmers to advise the operating system that certain memory allocations would be well suited to be supported by extended size mappings. The hinting is accomplished by passing the “MADV_HUGEPAGE” flag to the madvise() system call. The opposite hint is also available. The "MADV_NOHUGEPAGE" is used to specify to the Kernel to not attempt extended size mappings for a memory allocation requests.

Due to these advantages, current development efforts are focused on improvements to the transparent model. In spite of these improvements, some disadvantages remain.

One of the first disadvantages with the current THP implementation is that the VMM is only able to handle a single extended page size of two megabytes. This leads to a limitation on architectures which support multiple page sizes or which don't have support for two megabyte pages.

The realloc() standard library call (which invokes the mremap() system call) must have special handling with transparent huge pages. When a segment of memory backed by transparent huge pages is reallocated, the operating system splits the extended pages into four kilobyte pages in order to process the reallocation request. Once complete, it is the responsibility of the 'khugepaged' monitoring daemon to coalesce the newly reallocated memory segment back into extended pages.

Lastly, unlike statically allocated extended pages, transparent huge pages offer no guarantees that the application will have access to extended size mappings since the memory is not explicitly reserved.

1.10. Objectives

The objective of this work is to analyze the previously described extended paging methods in simulation environments which have significant memory requirements. The specific areas of investigation are as follows:

- Understand and analyze the impact of varying memory allocation architectures used by different applications
- Determine the effectiveness of application integration strategies with respect to page fault rate, dTLB performance, and system time with a collection of non-synthetic benchmarks.
- Determine where performance advantages are gained within the VMM when using extended page sizes.
- Investigate performance anomalies and situations where extended size pages are not appropriate.

2. RELATED WORK

There is a large body of work across multiple hardware and operating system architectures to address the problem of TLB eviction pressure. These are characterized by various strategies which seek to leverage increased TLB scope through the use of extended hardware managed page sizes. The following discussion offers a high level overview of the use and performance of extended page sizes to increase application and operating system performance.

The effectiveness of static extended page sizes measured in [36] demonstrates a variety of test cases in which several common benchmarking applications are used. These benchmarking applications suggest an overall improvement in application run time when using with statically allocated extended pages. In most of these test cases, the synthetic benchmarks had a performance improvement of between 5 and 10 percent.

In [29], comparison tests were run on another set of synthetic benchmarking applications against both transparent and statically allocated extended pages. In this case, it was discovered that the transparent paging method was slightly less effective compared to the static method. It was noted in this study that the comparison could not be considered an exact 'like-by-like' study as the 'hugetlbfs' backed memory segments were allocated using shared memory while the transparent segments were allocated using anonymous memory mappings.

In [28], a custom benchmarking application is used to determine the effectiveness of statically allocated extended page mappings. This benchmarking application used a random read-write access pattern to determine the effect of a TLB miss. It was found that extended pages didn't always outperform standard pages, especially at reduced memory levels.

In [30] the effect of operating system “noise” which in this report is in the form of TLB misses, interrupts, and asynchronous events. This paper uses a custom microkernel versus several different Linux kernels to better understand how operating system noise affects a system. It was noted that the impact on TLB eviction pressure can be greatly affected just by application code alone.

In [31] an analysis of TLB miss-rates is conducted. In this paper, the TLB is treated as a bottleneck which is alleviated by increasing page size and potentially supporting two page sizes. A SPARC system was used which is programmed to support a single 32 kilobyte page size or two page sizes, a 4 kilobyte (standard) and 32 kilobyte (extended). It was found that the single 32 kilobyte page had a 60% increase in average working set size and significantly improved TLB statistics compared to a single four kilobyte page model. The mixed page size model had a 10% increase in average working set size with almost no improvements to TLB pressure.

The reference cited in [32] analyzes the benefits of using extended page sizes on *Open Multiprocessing* (OpenMP) applications. In this paper, a custom OpenMP application able to support extended page sizes was developed. Results were gathered from the application using Oprofile and a 25% improvement in execution time was seen in some cases.

In [33] a vendor specific microkernel as an alternative to Linux in a high-performance computing center is introduced to combat the impact of TLB misses. A “big memory” design is proposed which uses extremely large extended pages available on the PowerPC architecture in an attempt to create a fully transparent TLB-miss-free environment. Single compute node benchmarks showed a 0.03%-0.2% improvement compared to a Linux Kernel. When run on a 1024 node cluster using a benchmarking application, a 0.1% to 0.7% improvement was noted.

In [35], decreasing communication overhead in parallel applications through extended page sizes is evaluated. By using appropriate data placement strategies, the author was able to utilize a transparent approach to extended page sizes to decrease memory registration costs and improve network bandwidth.

The author points out how communication latency can vary depending on how data is placed and organized in memory. By using extended pages and proper in-memory data placement, more than a 10% improvement in communication performance resulted with applications using RDMA over Infiniband.

In [34] the authors detail the difficulties of a transparent design due to architectural limitations and operating system overhead cost. The overall cost of the transparent model of extended pages access is brought into question and instead, an explicit static method of allocation is proposed. The explicit method saw an average 2% to 10% improvement on x86-64 and a 4%-15% improvement on PPC64 when using standard memory benchmarking applications.

The inference from [29], [33], [34], and [35] suggests that simply increasing the hardware page size is not enough to guarantee an application performance increase. Factors such as hardware architecture, number of processing cores, degree of parallelism, amount of memory, page size, and paging method all need to be taken into account when developing a solution which supports maximum application performance.

3. METHODS

For the purpose of this investigation we will be observing how the use of extended page sizes influence the performance of a variety of applications found in high performance computing which have significant physical memory requirements. These applications include HPL, GAMESS, and a specialized vertical data mining application.

3.1. System Overview

The following evaluations were carried out on a 1U server. The application x86info[26] was used to obtain the following system characteristics:

<u>CPU:</u>

- Dual-quad core 2.66 Ghz Processor; E5430 Xeon

<u>Cache Information:</u>

- L1 Instruction cache: 32KB, 8-way associative. 64 byte line size.
- L1 Data cache: 32KB, 8-way associative. 64 byte line size.
- L2 cache: 6MB, 24-way set associative, 64-byte line size.

<u>TLB Information:</u>

- Instruction TLB: 4x 4MB page entries, or 8x 2MB pages entries, 4-way associative
- Instruction TLB: 4K pages, 4-way associative, 128 entries.
- Data TLB: 4K pages, 4-way associative, 256 entries.

- L1 Data TLB: 4KB pages, 4-way set associative, 16 entries

Address Sizes:

- 38 bits physical, 48 bits virtual

Main Memory:

- 16 gigabytes

Operating System:

- Linux kernel version 3.4.7

3.2. Performance Monitoring

The *Performance Monitoring Unit* (PMU) [20] of Intel based CPUs contain a collection of registers that count the number of hardware events generated by the execution of an application. These events include cache-misses, total instructions executed, context-switches, page faults, and much more. In addition to this, the PMU is used to measure where applications spend the majority of their time executing. The PMU is a common feature on all modern x86_64 processors and IA-64 based processors.

The Linux Perf subsystem provides an abstraction interface to interact with the PMU to gather event data. This interface is used gather application events by process, thread, and function from the PMU.

A userspace tool called 'perf' is used to query the Perf subsystem for application event information. Figure 3.1 outlines the interactions of these tools and interfaces.

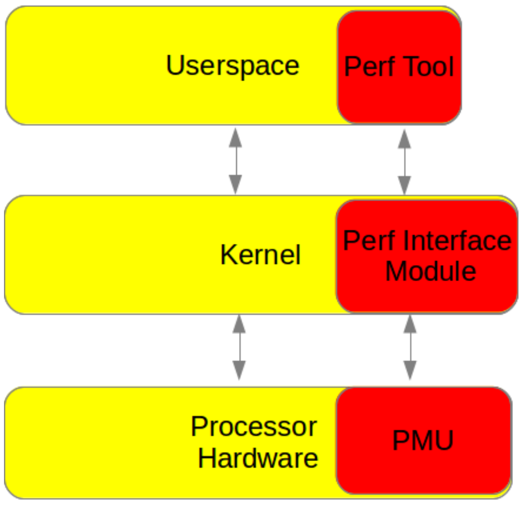
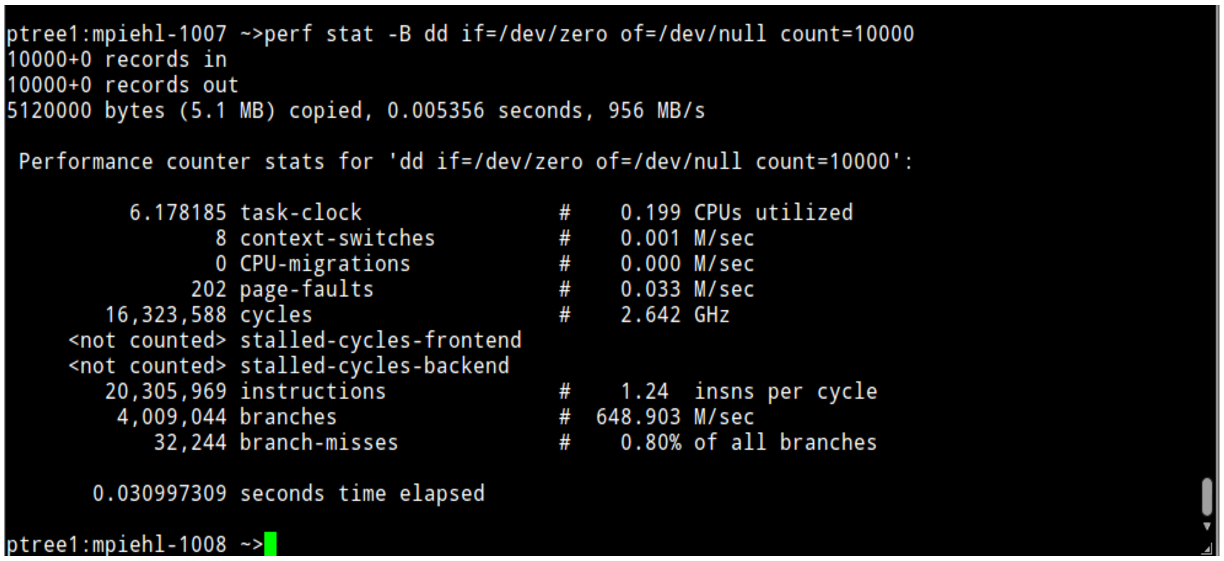


Figure 3.1. Perf tool interfaces

Figure 3.2 provides example output from the 'perf stat' command on a simple UNIX 'dd' command. With 'perf stat', hardware events are aggregated during program execution from the PMU registers and presented through standard output once the application has finished executing.



```
ptree1:mpiehl-1007 ~>perf stat -B dd if=/dev/zero of=/dev/null count=10000
10000+0 records in
10000+0 records out
5120000 bytes (5.1 MB) copied, 0.005356 seconds, 956 MB/s

 Performance counter stats for 'dd if=/dev/zero of=/dev/null count=10000':

          6.178185 task-clock                #    0.199 CPUs utilized
                 8 context-switches          #    0.001 M/sec
                 0 CPU-migrations            #    0.000 M/sec
               202 page-faults               #    0.033 M/sec
        16,323,588 cycles                    #    2.642 GHz
     <not counted> stalled-cycles-frontend
     <not counted> stalled-cycles-backend
        20,305,969 instructions              #    1.24  insns per cycle
         4,009,044 branches                  #  648.903 M/sec
            32,244 branch-misses             #    0.80% of all branches

       0.030997309 seconds time elapsed

ptree1:mpiehl-1008 ~>
```

Figure 3.2. Example perf stat command

Table 3.1 summarizes common commands used with perf to gather, record, and view various system events.

Table 3.1. Perf tool commands

Command	**Description**
stat	Display event counts
record	Record events for later reporting
report	Breakdown events by process, function, thread
annotate	Annotate source code with event counts
top	View live(current) event counts

System events provided by this interface, in combination with system time, provided the means of benchmarking the applications being studied. The primary event of interest to this investigation is the number of minor page faults and TLB misses an application generates during execution.

The page fault metric is used to determine to what extent page faults occur when using varying page sizes and methods (static vs. transparent). The overall execution time will be used to determine whether or not the use of extended page sizes, through a reduction in page faults and TLB misses, has a user impact on application performance.

3.3. High Performance LINPACK

The High Performance LINPACK (HPL) is a portable, highly scalable linear algebra application package used as the standard test to measure the execution rate of distributed high performance computing clusters [1]. Benchmark results from this application are used as the standard for the TOP500 ranking [21], which is a ranking of the 500 fastest super computers in the world.

HPL measures cluster performance in floating point operations per second (FLOPS). This unit of measure is commonly interrupted as TeraFLOPS or GigaFlops (TFLOPS/GFLOPS), where GFLOP/s and TFLOP/s are a measure of the number of billion or trillion floating point operations a computer system can execute in one second. The floating point operations used as a metric consist of 64-bit multiplications and additions.

As of June 2012, Sequoia, a super computer used by the Department of Energy located at Lawrence Livermore National Laboratory (LLNL) holds the top ranking on this list at 16,324,751 GFlops (16.3 PetaFlops).

3.4. Theoretical Peak Performance

The theoretical rate at which a computer can execute an application is not based on actual performance measurements. Theoretical performance of a computer is a generalized metric to determine the peak throughput rate of execution in FLOPS. A computers performance will not exceed this theoretical limit thus establishing an upper-bound.

The theoretical peak performance is computed by determining the number of floating point operations that are completed in a single clock cycle on the machine. For example, the theoretical execution rate for the system being using for this investigation is as follows:

- Quad core Intel Xeon CPU at 2.66 GHz can complete 4 floating point operations per core per cycle
- 4 Flop/s * 2.66 GHz * 8 cores = 85.12 Gflop/s

The actual measured performance of the platform is a complex metric with a large number of factors involved. These include algorithmic efficiency, problem size, memory usage, memory speed, disk speed, programming language, compiler, operating system, etc. The results presented in this benchmarking should not be used as a measure of total system performance but instead as a reference for evaluation.

Table A.1 details a description of input parameters used for this application. The 'Ns' value is adjusted between tests in order to exercise a variety of memory profiles (5Gb, 10Gb, 15Gb). All experimental runs were conducted using HPL version 2.0.

Figure A.2 contains the changes made to the HPL source code to implement the use of static extended pages in the application. The following changes were placed at line 241 in src/panel/HPL_pdpanel_init.c in the HPL source tree.

When using mmap() with extended page sizes, the allocation request must lie on an extended page boundary. To support this, a conditional was placed in the source code to ensure allocation requests greater than two megabytes are aligned on extended page sizes. If the allocation request is less than two megabytes, a standard malloc() is used with regular page sizes.

3.5. General Atomic and Molecular Electronic Structure System

The General Atomic and Molecular Electronic Structure System (GAMESS) is a freely available computational chemistry application developed by Mark Gordon and researchers at Iowa State University [2]. GAMESS is used to model the electronic structure of atoms and molecules. GAMESS shares the common characteristics of other computational chemistry applications, which include large memory requirements, significant disk IO, and long execution time.

This investigation will use a memory intense simulation using a Restricted Hartree-Fock wavefunction enhanced with Moller-Plesset Perturbation Method (MP2) corrections on the Buckminsterfullerene (C60) molecule. An augmented triple-zeta (aug-cc-pVTZ) basis set was used to maximize the amount of physical memory required for the simulation. This simulation was created with the help of [37]. The input file is located in Figure A.1.

Buckminsterfullerene is a spherical molecule entirely carbon based. The cage structure of the sphere is made up of 20 hexagons and 12 pentagons [22]. Figure 3.3 contains a representation of the molecule drawn using the modeling language Avogadro [27].

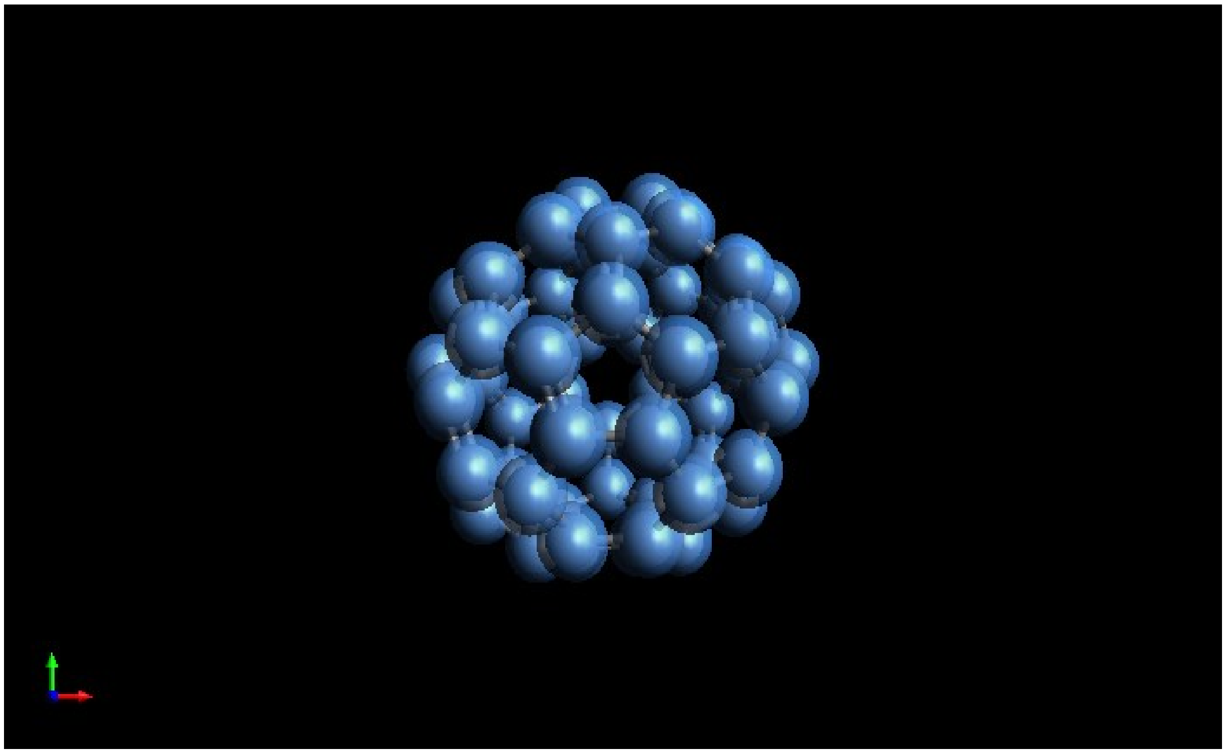


Figure 3.3. Buckminsterfullerene image

Simulations such as this one are common in computational chemistry and can take days or weeks to complete depending on the level of theory used. The described simulation uses 6.3

gigabytes of physical memory and requires approximately 20 hours of computational time to complete.

For this simulation, GAMESS version 5-22-09 was used. Appendix A contains the input file for this computation. It should be noted that before being run in GAMESS, the geometry of the molecule was optimized in Avogadro.

The GAMESS application is primarily written in Fortran. However, the memory allocation routines are implemented in C. In order to properly include support for statically allocated extended pages, the malloc() standard library call was replaced with a mmap() system call. A conditional is in place to ensure the allocation request is properly aligned.

Lastly, a single free element at the end of the array is used to hold the length of the array in order to properly free the array using the munmap() function. In order to facilitate these changes, the file 'zunix.c' was modified as detailed in Figure A.3 and A.4.

3.6. Vertical Data-mining Application (PTrees)

Developed by Dr. Perrizo and researchers at *North Dakota State University* (NDSU), PTrees (Predicate trees) implement methods for prediction and analysis using vertically structured data. These methods are being developed to address challenges involved in the analysis of massively large data sets. These methods are based upon vertical structuring of data combined with compression of these structures into hierarchical arrangements which provide the opportunity for accelerated analysis.

These methods were applied to the Netflix data mining competition. This competition focused on developing improvements to improve the accuracy of the NetFlix movie recommendation system. A monetary prize of $1,000,000 was offered as an incentive to develop a solution. Netflix provided research contestants with 7 years of past rating data. This data include 100,480,507 user ratings on 17,770 movies from 480,189 random Netflix customers.

The prediction system developed for the NetFlix problem was based on vertically structured methods. The analytical methods are based on a library which implements primary PTree operations which consist of binary and unary boolean operations. The libPTree library is used to implement the operational primitives on the vertically structured NetFlix data. [38] The prediction application and its associated library are used to measure the effect of extended page sizes.

The vertical data-mining application algorithm functions by implementing item-based collaborative filtering. This algorithm attempts to predict a rating a NetFlix user would give to a particular movie. NetFlix provides past rating data in the form of training data which is provided in a single large text file. This training data is compressed into a binary vertical PTree structure and saved to disk.

Once the application begins, 6.8 gigabytes of data represented in vertically structured form by PTrees, are transferred from disk to main memory. Next, a movie similarity matrix is constructed which contains a predetermined number of the most similar movie-to-movie relationships. A prediction based on a particular users' favorability toward a movie is then made on the basis of the user’s past movie preferences against the similarity matrix. For the purpose of this study, 622,453 NetFlix user-ratings of 100 different movies are predicted.

In order to generate the similarity matrix and to determine the favorability of hundreds of thousands of users’ likeness of a specific movie, the vertically structured data must be repeatedly analyzed. This introduces a computational constraint which is influenced by memory access efficiency and latency.

The current libPtree code is written in C++ with C standard library and system calls. Figure A.6 details the code changes needed to support statically allocated extended pages.

The code changes for this application were the most extensive compared to the previous applications. With this application, the memory allocation architecture had to be rewritten to properly accommodate extended page sizes. The original architecture requested hundreds of thousands of 2,224 byte memory allocations which each belonged to its own class object. While these allocations added up to a large memory footprint (6.5 gigabytes), each individual allocation was not large enough to be handled with extended page sizes without a significant degree of internal fragmentation.

In order to resolve this issue, the thousands of memory allocations were replaced with a single large memory allocation which was capable of containing the entire 6.5 gigabyte PTree dataset. Once the dataset was fread() into the buffer, each PTrees data object's tree array variable referenced the position in the large array where the object specific PTree was located. This allowed the new large buffer to be allocated with an anonymous mmap() backed by extended size pages.

4. EXPERIMENTAL RESULTS AND ANALYSIS

In this section we will observe how the previously described scientific applications perform using a variety of integration methods on a collection of memory intensive simulations. For completeness, a large number of system statistics were gathered using the PMU software including dTLB loads/stores, the number of page faults, system, userspace, and execution times. In order to eliminate timing transients and ensure correctness, each performance assessment for each paging method was executed three times. The results of these runs are provided in tables 4.1 to 4.9.

4.1. HPL Results

The first assessment involved evaluating HPL using a standard four kilobyte pages to establish a baseline performance metric. This metric will serve as a control to compare the effectiveness of the various integration methods used.

Each test run of the HPL application was executed using eight processes over a steadily increasing physical memory size. The “N Size” represents the size of the matrix used during decomposition, as its size increases the overall physical memory footprint occupied by the application increases as well.

The average results of these baseline runs are summarized in Table 4.1. In each set of application executions, the memory utilized increased at an exponential rate which translated into an exponential increase in page fault rate. At the final N size increase, a 26.6% increase in physical memory usage resulted in a 29.5% increase in the number of page faults.

TLB pressure also climbed significantly as the problem size and physical memory requirements increased. dTLB load-misses from N size 35,000 to 40,000 increased by 47.9%. In addition, dTLB store-misses, for the same N size range increased by 32.7%. System time averaged 13.16 seconds on an N size of 40,000. The partitioning of this time will be discussed later.

Table 4.1. Experimental results: HPL with four kilobyte pages

N Size	**5000**	**15000**	**25000**	**35000**	**40000**
Memory(MB)	1550	2843	5602	9371	11871
Page Faults	77,482	517,389	1,344,853	2,570,603	3,329,412
dTLB-loads	29,338,925,503	552,941,215,983	2,369,933,621,884	6,321,361,399,214	9,275,877,679,481
4Kb dTLB-load-misses	10,070,349	137,363,529	548,013,720	1,379,489,146	2,012,378,206
dTLB-stores	5,466,989,725	53,975,664,885	177,150,206,806	420,025,762,269	562,893,505,586
dTLB-store-misses	1,764,952	21,392,034	86,769,290	234,834,897	310,261,395
iTLB-loads	103,430,912,398	2,119,589,275,510	9,271,921,132,159	24,902,318,701,857	36,747,531,771,252
iTLB-load-misses	148,910	256,825	464,079	751,061	905,398
	4.58	53.10	221.42	586.45	849.05
4Kb time elapsed(seconds)	0.69	0.20	0.17	0.91	0.91
	22.75	412.60	1,752.82	4,659.06	6,755.52
Userspace Time	0.42	1.64	1.31	7.83	6.03
	0.42	2.27	5.75	11.00	12.97
4Kb system time	0.03	0.01	0.08	0.08	1.25
	35.35	46.81	48.72	50.19	51.55
Gflops	0.02	0.01	0.00	0.00	0.00

Table 4.2 details the average results for three experimental runs using a two megabyte extended page size with transparent page support. As in the previous test, the application was run with the same input file and problem size configuration to ensure equivalent system memory usage.

The use of extended size pages resulted in a page fault rate increase of 3.3% between N size of 35,000 to 40,000. In contrast to the basline run, use of extended page sizes results in a decrease in the number of page faults. In addition, there was a 12.75% decrease in dTLB load-misses for this problem size compared to the previous value of 47.9%.

All of the transparent cases resulted in a decrease in the overall system time for each application run. These decreases are attributed to two primary factors which will be discussed in the next section.

Table 4.2. Experimental results: HPL with transparent extended pages

N Size	**5000**	**15000**	**25000**	**35000**	**40000**
Memory(MB)	1550	2843	5602	9371	11871
Page Faults	29,505	47,117	50,019	66,509	70,338
dTLB-loads	29,998,551,448	556,897,454,669	2,366,798,239,424	6,287,224,745,673	9,268,143,000,469
2MB-transparent dTLB-load-misses	3,914,936	35,634,631	82,526,129	216,379,969	262,130,058
dTLB-stores	5,673,718,419	55,249,394,548	175,363,789,264	404,804,783,815	559,060,582,342
dTLB-store-misses	617,450	2,521,421	4,857,158	10,043,933	11,891,715
iTLB-loads	104,135,253,088	2,125,360,229,064	9,261,007,931,350	24,819,864,190,443	36,723,262,098,116
iTLB-load-misses	147,482	250,552	432,477	739,778	904,972
	4.40	52.97	219.76	577.73	840.33
2MB-transparent time elapsed(seconds)	0.49	0.55	0.08	0.65	0.76
	23.09	412.90	1,743.73	4,599.21	6,696.41
Userspace Time	0.22	0.62	2.38	0.58	2.67
	0.32	1.27	2.92	5.75	7.24
2MB-transparent system time	0.02	0.11	0.01	0.23	0.32
	35.77	48.06	50.82	52.01	53.07
Gflops	0.06	0.25	0.03	0.30	0.16

Table 4.3 contains the average HPL run results using a statically allocated extended page integration method. As the table shows, there was a uniform slight improvement when compared to the transparent model.

Due to the application's memory allocation architecture, slightly more memory was utilized when the static integration method was used. This was the result of the slightly larger mmap() allocations required to ensure each allocation was an exact multiple of the extended page size. This is discussed further in the next section.

Between N sizes 35,000 and 40,000, there was a 2% increase in the number of application minor page faults. This further improves on the previous transparent rate of 3.3% and the base case of 29.5%.

There was a 13.8% decrease in dTLB load-misses between this run and the baseline for the N size range of 35,000 to 40,000. The system time on most static runs out performed the transparent runs by a very small margin, often times by as a little a tenth of a second.

Table 4.3. Experimental results: HPL with static extended pages

N Size	**5000**	**15000**	**25000**	**35000**	**40000**
Memory(MB)	1560	3002	5954	9787	12684
Page Faults	25,570	42,775	46,539	63,062	64,105
dTLB-loads	30,620,237,829	562,507,215,821	2,376,868,313,078	6,304,754,865,088	9,379,134,313,867
2MB-static dTLB-load-misses	3,629,084	32,685,939	87,201,887	217,948,706	264,054,549
dTLB-stores	6,029,825,074	58,099,578,607	181,152,852,327	415,249,163,176	601,675,648,588
dTLB-store-misses	638,074	2,069,082	4,598,664	9,429,038	11,619,247
iTLB-loads	105,982,585,985	2,138,410,918,872	9,285,334,294,886	24,864,231,987,083	37,021,508,431,113
iTLB-load-misses	133,974	246,424	429,558	683,696	863,801
	4.56	52.41	218.00	572.38	823.38
2MB-static time elapsed(seconds)	0.49	0.55	0.08	0.65	0.76
	23.80	413.37	1,742.87	4,585.33	6,692.15
Userspace Time	0.22	0.62	2.38	0.58	2.67
	0.34	1.23	2.86	5.48	6.91
2MB-static-extended system time	0.02	0.11	0.01	0.23	0.32
	34.77	48.22	51.09	52.68	53.71
Gflops	0.06	0.25	0.03	0.30	0.16

With this application it is clear that using extended page sizes improves application performance. Figures 4.1 to 4.3 contain graphical representations of important system statistics to compare the three paging strategies used with HPL.

Figure 4.1 contains the dTLB-load-misses comparison between each paging method, Figure 4.2 contains a comparison of the average overall run time, Figure 4.3 contains the average system time for each performance assessment.

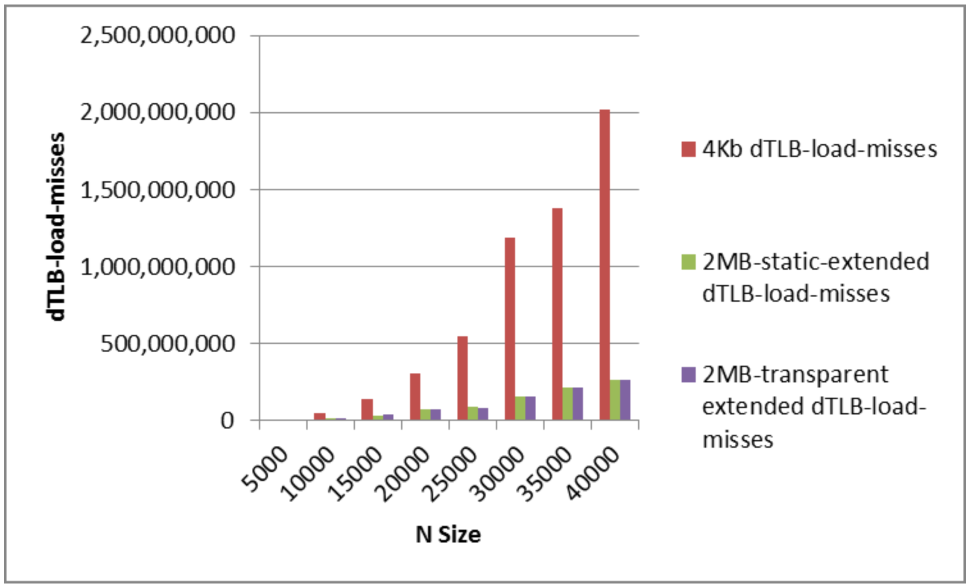


Figure 4.1. Experimental results: Average dTLB load-misses in HPL

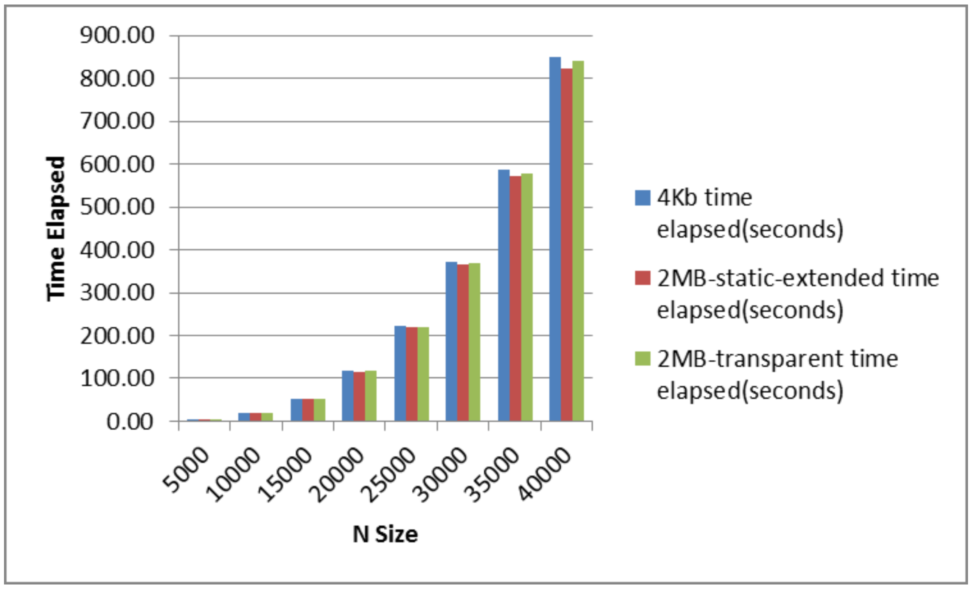


Figure 4.2. Experimental results: Average execution time in HPL

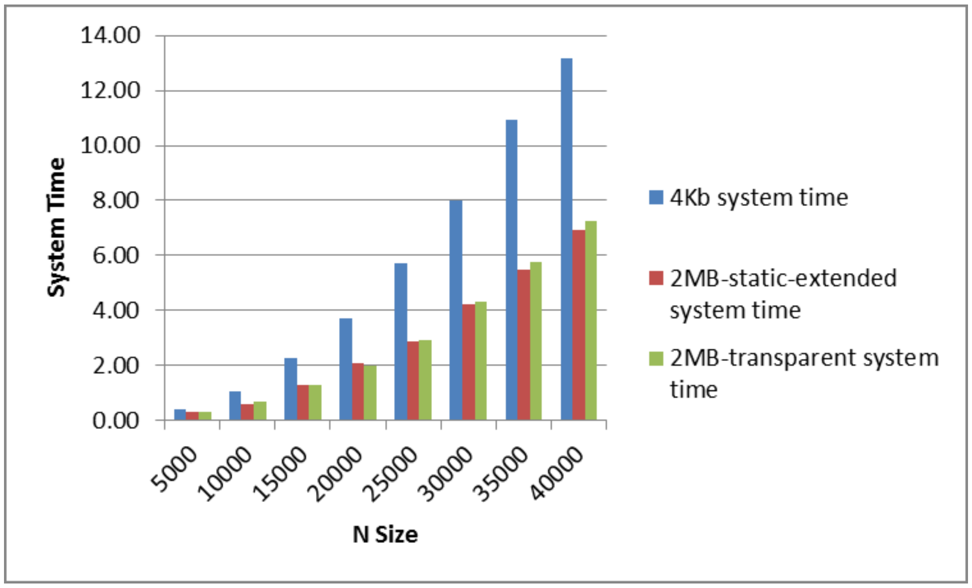


Figure 4.3. Experimental results: HPL with static extended pages

4.2. GAMESS Results

The testing and analysis of GAMESS was approached in a similar manner to the HPL application. The evaluation of the performance of each paging strategy was assessed by a model simulation described by the input file Table A.1. This input model required 6.5 gigabytes of memory which was allocated at the beginning of each simulation. Table 4.4 contains the average performance statistics of three GAMESS runs using a standard page size of four kilobytes.

Table 4.4. Experimental results: GAMESS with four kilobyte pages

Memory(GB)	6.5
Page Faults	9,327,727
dTLB-loads	16,099,434,770,249
4Kb dTLB-load-misses	191,875,297,119
dTLB-stores	4,898,981,277,659
dTLB-store-misses	271,448,901,747
iTLB-loads	45,786,055,686,513
iTLB-load-misses	3,321,999
	75,992.81
4Kb time elapsed(seconds)	393.50
	74,700.08
Userspace Time	340.39
	1,333.72
4Kb system time	6.64

Tables 4.5 details average experimental run statistics using a two megabyte extended page size with transparent page support.

The use of transparent extended size pages with GAMESS resulted in an average page fault rate of 392,758 which was a 95% decrease compared to the four kilobyte benchmark. Similar to the HPL results, use of extended page sizes resulted in a decrease in the number of page faults compared to 4 kilobyte pages. In addition, there was an average of 18,911,053,951 dTLB-misses per run which was a 9.8% decrease compared to the baseline results.

The transparent paging method used an average of 645.61 seconds of system time per run. This was a 48.5% improvement in performance time when compared to the four kilobyte baseline value.

Table 4.5. Experimental results: GAMESS with transparent extended pages

Memory(GB)	6.5
Page Faults	392,758
dTLB-loads	15,778,947,563,803
2MB-transparent dTLB-load-misses	20,244,387,285
dTLB-stores	4,854,633,911,900
dTLB-store-misses	14,869,695,598
iTLB-loads	45,479,347,843,756
iTLB-load-misses	2,762,298
	75,004.52
2MB-transparent time elapsed(seconds)	554.46
	75,325.57
Userspace Time	1,400.09
	645.61
2MB-transparent system time	5.04

Table 4.6 presents the average results for three runs using a statically allocated extended page model. Similar to the transparent results, page fault rate decreased by 96% compared to the benchmark value. dTLB-load-misses also decreased at a rate similar to the transparent results.

It should be noted that the static allocation model resulted in a decrease in average system time when compared to the transparent method. This is likely due to reduced demands on the operating system from not having to allocate resources to page coalescences. The GAMESS memory allocation architecture of using a single buffer makes the use of static page allocation particularly efficient.

Table 4.6. Experimental results: GAMESS with static extended pages

Memory(GB)	6.5
Page Faults	387,015
dTLB-loads	15,617,020,475,385
2MB-static dTLB-load-misses	20,520,389,478
dTLB-stores	4,847,999,877,867
dTLB-store-misses	12,592,415,096
iTLB-loads	46,137,872,466,241
iTLB-load-misses	2,767,822
	74,913.94
2MB-static time elapsed(seconds)	71.47
	75,375.89
Userspace Time	236.92
	632.71
2MB-static system time	1.49

Figure 4.4 to 4.6 contain bar graph representations of the above results. Figure 4.4 is the dTLB-load-misses for each set of three runs using the different paging methods, Figure 4.5 is overall execution time for each paging method, and Figure 4.6 is the graphical representation of the system time for each run.

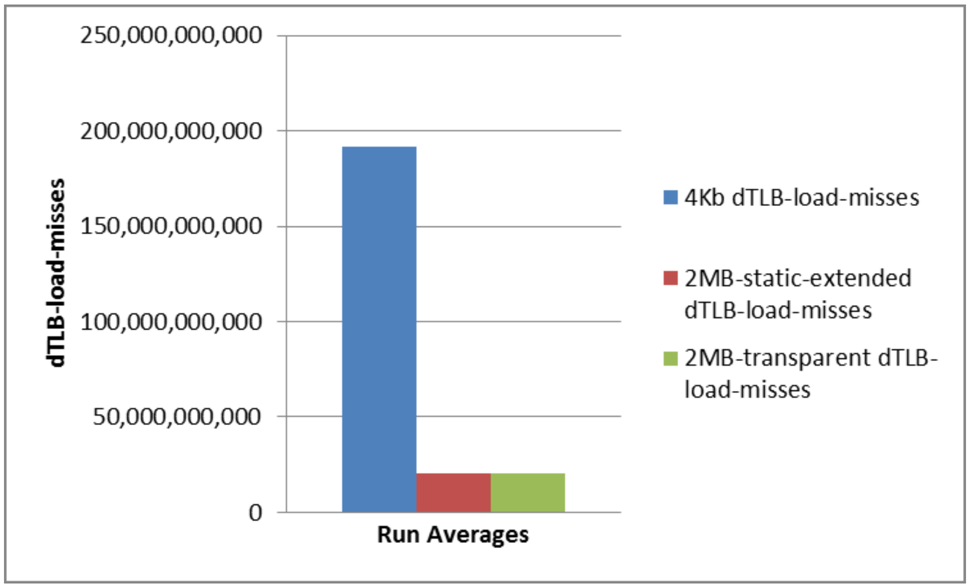


Figure 4.4. Experimental results: Average dTLB load-misses in GAMESS

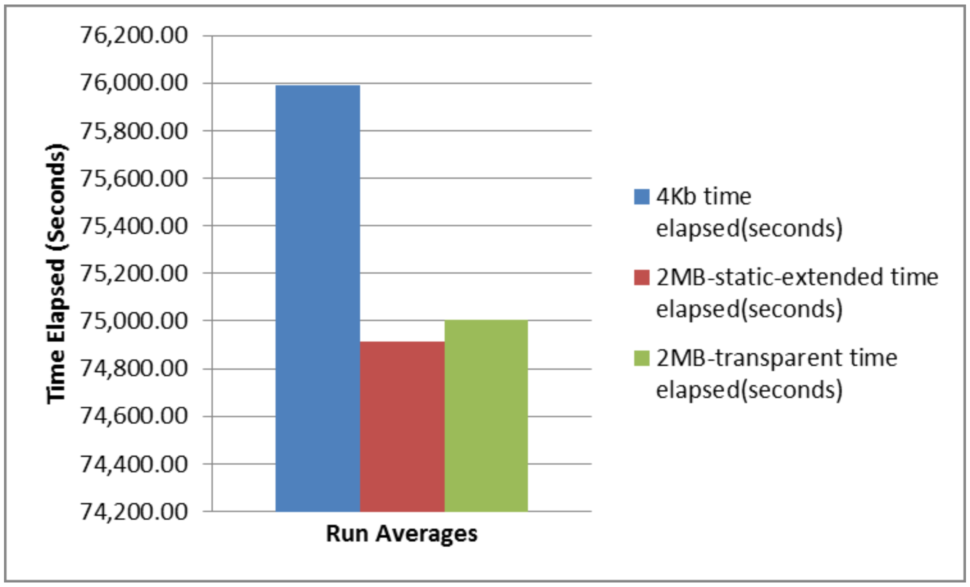


Figure 4.5. Experimental results: Average execution time in GAMESS

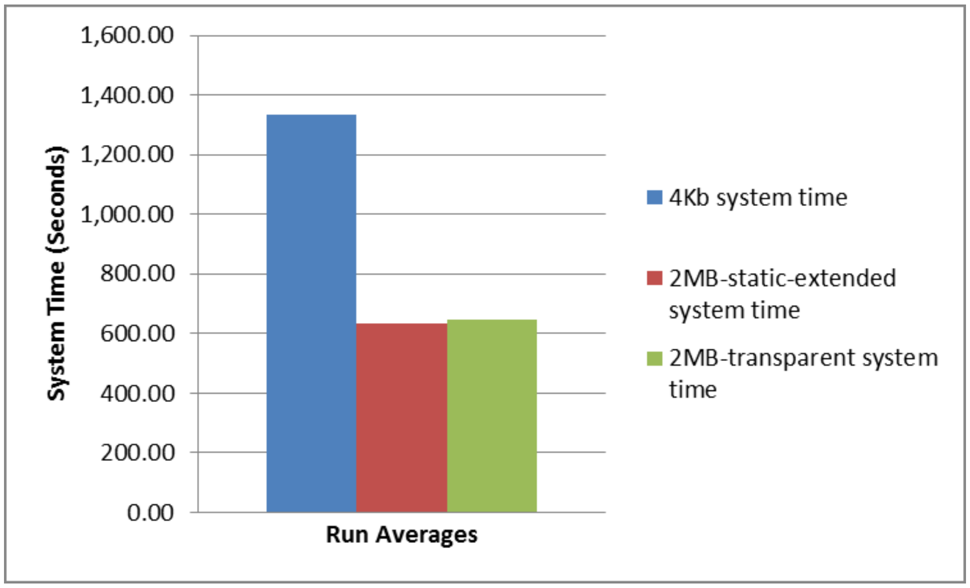


Figure 4.6. Experimental results: GAMESS with static extended pages

4.3. Vertical Data Mining Results

Table 4.7 presents results for average baseline baseline run of a vertical data-mining application. To remove the application cost of the initial load of the data into memory from hard disk, the application was first run once on system prior to initial benchmarking. This ensures that the page cache was populated with data prior to test runs. Testing was conducted in a manner similar to the previous applications.

Table 4.7. Experimental results: PTrees with four kilobyte pages

Memory(GB)	6.3
Page Faults	613,359,804
dTLB-loads	49,207,872,777,818
4Kb dTLB-load-misses	132,083,167,159
dTLB-stores	15,109,495,668,408
dTLB-store-misses	246,000,108,765
iTLB-loads	142,702,658,622,792
iTLB-load-misses	10,145,979
	249,569.00
4Kb time elapsed(seconds)	125.12
	1,818,112.37
Userspace Time	4230.21
	33,824.93
4Kb system time	51.77

Table 4.8 contains the average performance results when the application is run using a transparent paging architecture. Compared to the standard four kilobyte paging method, the transparent model had a 51.9% decrease in the number of dTLB-load-misses. In addition, the page fault rate decreased by 96%. Lastly, the overall average system time of 16,391.02 seconds was a 51.54% decrease compared to the baseline performance assessment.

Table 4.8. Experimental results: PTrees with transparent extended pages

Memory(GB)	6.3
Page Faults	24,087,478
dTLB-loads	50,082,340,605,514
2MB-static dTLB-load-misses	63,443,297,806
dTLB-stores	15,406,330,382,482
dTLB-store-misses	50,582,972,225
iTLB-loads	144,338,243,301,581
iTLB-load-misses	8,703,010
	244,459.37
2MB-static time elapsed(seconds)	98.34
	1,722,617.16
Userspace Time	2142.43
	16,391.02
2MB-static system time	43.76

Table 4.9 contains the average performance assessment on the vertical data-mining application when paired a with statically allocated extended page size architecture. Similar to the transparent performance assessment, page faults and dTLB-misses were markedly less than the benchmark assessment. In every case the static runs also slightly out preformed the average system time of transparent assessment by 3.4%.

Table 4.9. Experimental results: PTrees with static extended pages

Memory(GB)	6.3
Page Faults	22,185,634
dTLB-loads	49,854,668,541,845
2MB-transparent dTLB-load-misses	54,897,558,631
dTLB-stores	15,385,416,855,174
dTLB-store-misses	47,856,945,201
iTLB-loads	141,584,004,521,877
iTLB-load-misses	7,854,215
	241,857.65
2MB-transparent time elapsed(seconds)	153.11
	1,702,485.75
Userspace Time	1539.54
	15,241.42
2MB-transparent system time	53.02

Significant programming effort was required to implement the statically allocated integration but the performance benefits exceeded both the transparent and standard integration strategies. Figures 4.7 to 4.9 contain graphical representations of important system statistics used to compare the three paging strategies used with the vertical data-mining application. Figure 4.7 contains the dTLB-load-misses comparison between each paging method, Figure 4.8 contains a comparison of the average overall run time and Figure 4.9 contains the average system time for each performance assessment.

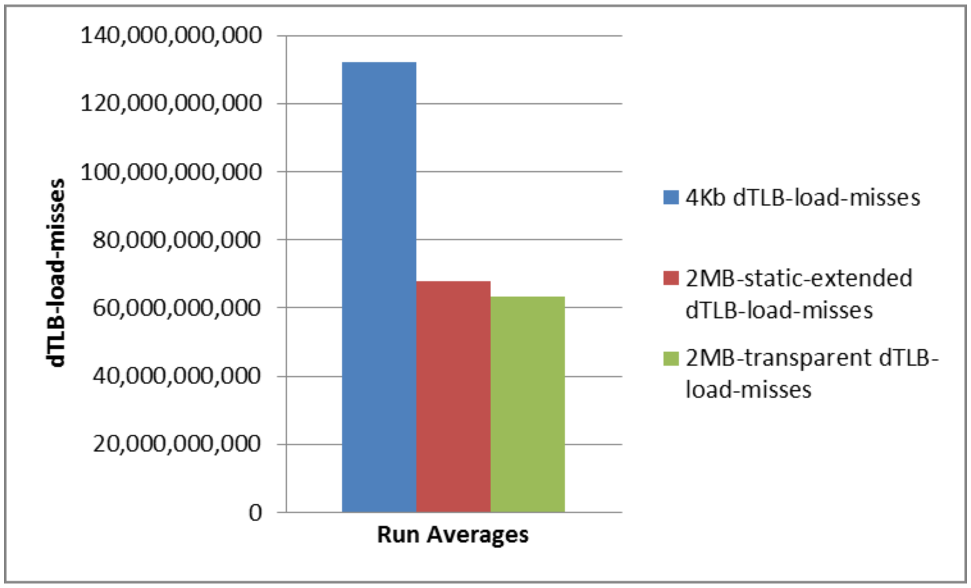


Figure 4.7. Experimental results: Average dTLB load-misses with PTrees

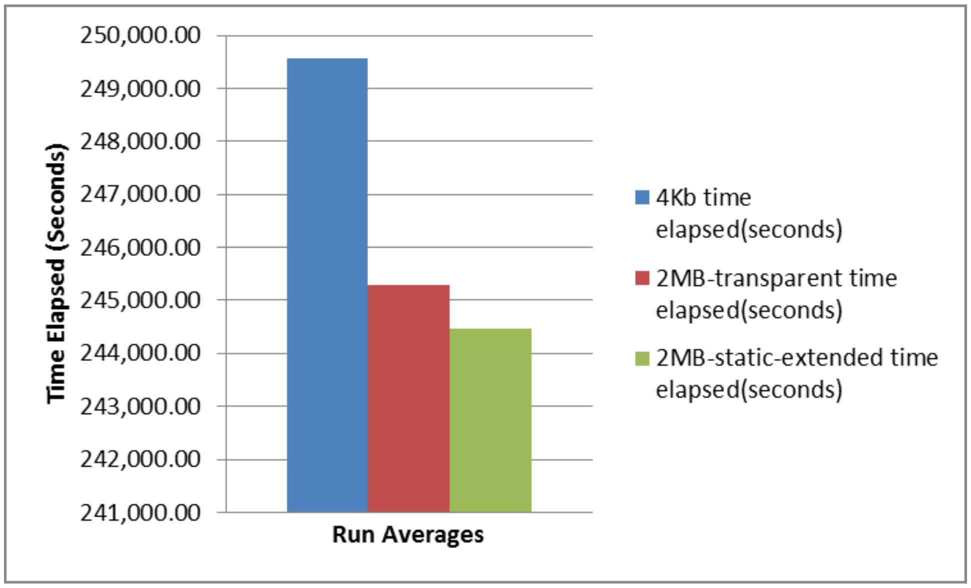


Figure 4.8. Experimental results: Average execution time with PTrees

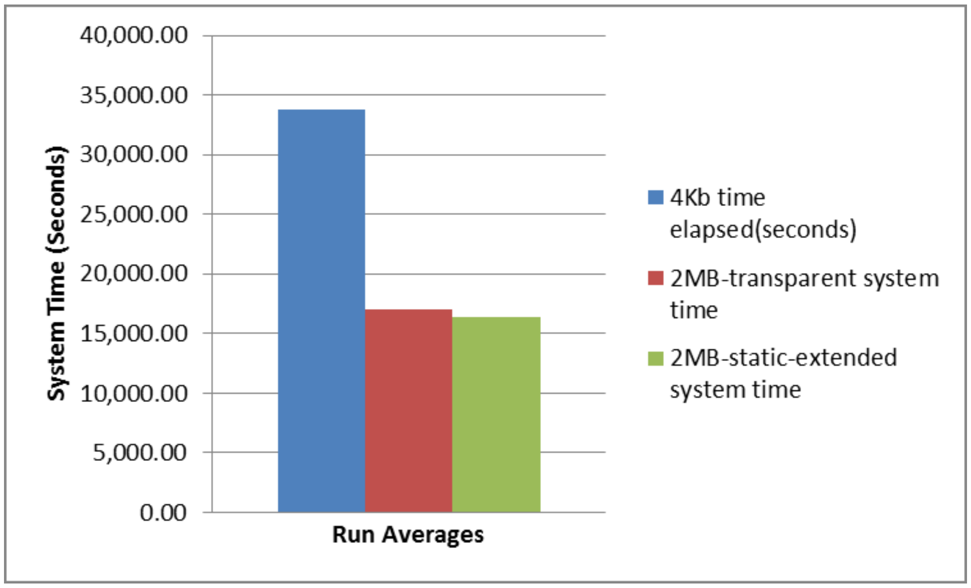


Figure 4.9. Experimental results: Average system time with PTrees

5. DISCUSSION

Before considering further analysis and discussion about the results of this work it is important to consider issues which influence the strategy used to integrate support for extended size pages. While HPL, GAMESS, and the NetFlix code are all memory intensive applications, important architectural differences exist with respect to how application memory is allocated.

The HPL application uses a memory allocation pattern characterized by small-but-frequent allocation requests. Each application memory allocation request is typically between three and twelve megabytes in size. Depending on the problem size, the application generates hundreds to thousands of such requests rapid and parallel succession. When configured for a large problem size, these requests translate into a total memory consumption in the range of tens of gigabytes.

In GAMESS, memory is requested in single large segment. When configured within the methods section, GAMESS requires approximately six gigabytes of application memory. These six gigabytes of memory are allocated shortly after the start of the application in a single large buffer request.

In the original architecture, the vertical data mining application exhibited a memory allocation pattern which involves hundreds of thousands of 2,224 byte allocation requests. This allocation architecture was not suitable to extended page sizes.

In the alternative architecture which implemented, the conversion to a single large buffer request resulted in a architecture similar to that employed by GAMESS which made the static allocation strategy optimum for both applications.

These differences in application allocation behavior, while not obvious, play an important role during the analysis of the effectiveness of each paging scheme. When considering integration of extended page size support, it is not only important for the application to utilize large memory allocations. An equally important consideration is how the application requests memory segments as the type of allocation requests react much differently to different paging methods.

The three unique styles of memory allocation exposed by these applications resulted in several challenges during the course of this work. One of these challenges was a performance anomaly related to the memory allocation scheme of the HPL application when utilizing statically allocated extended pages. Due to the previously mentioned memory allocation characteristics of the HPL application, use of the libhugetlbfs API, resulted in a performance regression.

This is in contrast to the expected observation that use of extended size pages in high memory utilization environments would result in improved performance. When implementing extended size page support using libhugetlbfs, the get_hugepage_region() library call was used to replace the malloc() system call. An example of this call is provided in the introduction.

The use of this integration method resulted in a performance time regression. While this performance regression was noted, page faults and TLB pressure were improved from the standard page size.

An analysis of this regression indicated the HPL 'small-but-frequent' allocation pattern was the source of the regression. The applications memory allocation pattern resulted in memory requests which were not multiples of an extended page size. This resulted from non-extended page size aligned memory request. For simplicity this is referred to as a “two megabyte plus one byte mapping”. Consider the following allocation request:

allocation_request = (3*2*1024*1024) + X | where X is less than 2*1024*1024

(3 pages)

An allocation request such as this grants four pages however the total memory required is three pages plus one byte. This partial allocation requires operating system overhead to zero fill the remaining 2,097,151 bytes of the final page. The page zeroing cost generated enough operating system overhead to nullify the performance advantages of the use of an extended page size. This effect is not experienced with applications which make a small number of very large requests.

This issue can be resolved by allocating only on extended page size boundaries within libhugetlbfs. Rather than requesting the following integer value:

X = memory_request | where X is 0 < X < 2*1024*1024

Compute the offset to the next complete extended page and add the offset to the previous X value to complete the next page during allocation.

allocation_request = total_request + (total_request MOD extended_page_size)

By doing this, the operating system impact of the additional page clear can be avoided and the statically allocated extended page sizes are capable of outperforming the standard page size. Internal fragmentation will still exist on the page

6. CONCLUSION

In this paper, a group of extended page size application integration methods were introduced and analyzed using a selection of common scientific applications. The results and discussion suggest that using extended page sizes, when appropriate, can result in significant changes in application performance.

The results also demonstrate that depending on the type and size of the allocation requests, extended size pages can produce performance regressions. This suggests that integrating extended size page support requires an analysis of the applications memory allocation behavior.

Transparent huge pages offer an "ease of use" advantage which reduces development and integration costs needed to obtain performance improvements. Static allocation of extended pages produces additional performance improvements when compared to transparent huge pages but at the cost of additional management and implementation time. The static allocation model carries an additional advantage of guaranteeing deterministic performance and benchmarking benefits.

Several avenues exist for potential further research. While the dTLB was closely analyzed with regards to application performance, further work could done to determine how effective the use of extended page sizes would be in reducing iTLB costs.

In addition, some applications which issue memory reallocation requests currently do not have operating system support to handle this function with statically allocated extended pages. While transparent huge pages support memory reallocation requests, it is at the expense of degrading the memory allocation to standard size page backings with subsequent operating system cost to recoalesce the reallocated memory to extended page mappings.

Finally, the impact on application integration cost and performance of an object magazine architecture would be of interest. An object magazine system would involve language support to preallocate segments of memory backed by extended pages which would serve as an allocation pool for user requested objects and allocations.

7. REFERENCES

[1] Innovative Computing Laboratory, High Performance Linpack,

http://www.netlib.org/benchmark/hpl, June 2012.

[2] Gordon Research Group ISU, GAMESS Software Package,

http://www.msg.ameslab.gov/gamess/download.html, June 2012.

[3] Netflix, NetFlix Prize, http://www.netflixprize.com/index, Retreived July 2012

[4] Intel Corporation, "Intel 64 and IA-32 Combined Architectures Software Developer Manuals," June 2012.

[5] DataSURG, P-Tree Application Programming Interface Documentation,

http://midas.cs.ndsu.nodak.edu/~datasurg/ptree, November 2005.

[6] M. Gorman, "Understanding The Linux Virtual Memory Manager," Prentice Hall, pp. 36-42, 2004.

[7] P. Ceruzzi, "A History of Modern Computing," MIT Press, pp. 238. 2003.

[8] I. Englander, "The Architecture of Computer Hardware and Systems Software 3rd Edition," Wiley, 2003.

[9] R. Bryant, D. O'Hallaron, "Computer Systems A Programmer's Perspective,"

Prentice Hall, pp. 690-764, 2003.

[10] R. Bryant, D. O'Hallaron, "Computer Systems A Programmer's Perspective,"

Prentice Hall, pp. 694, 2003.

[11] Intel Corporation, "Intel 80386 Reference Programmer's Manual,"

http://css.csail.mit.edu/6.858/2012/readings/i386/toc.htm, 2012.

[12] Intel Corporation, "80286 Microprocessor With Memory Management and Protection,"

http://datasheets.chipdb.org/Intel/x86/286/datashts, July 2012

[13] T. Barr, A. Cox, S. Rixner, “Translation Caching: Skip, Don't Walk (The Page Table),” ISCA, 2010.

[14] D. Kanter, “Inside Nehalem: Intel's Future Processor and System,” Retrieved September 2012 from http://www.realworldtech.com/nehalem/4, April 2nd, 2008

[15] R. Bryant, D. O'Hallaron, “Computer Systems A Programmer's Perspective,” Prentice Hall, pp. 707-715, 2003.

[16] MSDN, “Operating Systems and PAE Support,” http://msdn.microsoft.com/en-us/library/windows/hardware/gg487512.aspx, July 14th, 2006

[17] J. Corbet, “Four-level Page Tables,” http://lwn.net/Articles/106177, October 12th, 2004

[18] SGI Press Release, “Announcing the New SGI UV: The Big Brain Computer,” http://www.sgi.com/company_info/newsroom/press_releases/2012/june/uv.html, June 18th, 2012.

[19] D. Knuth, “The Art of Computer Programming 1 (2nd Edition),” Addison-Wesley, pp. 435-455, 1973.

[20] Intel, “Performance Monitoring Unit Sharing Guide (White Paper),” Intel Corporation, http://software.intel.com/file/30388, 2010.

[21] Top500, Top 500 List, http://www.top500.org, 2012.

[22] H. Kroto, J. Heath, S. O'Brien, R. Curl, R. Smalley, “C60: Buckminsterfullerene,” Nature Volume 318, pp. 162-163, 1985.

[23] The Linux Kernel Archives, http://www.kernel.org, 2012.

[24] M. Gorman, “Using the Direct Hugepage Allocation API with STREAM,” http://www.csn.ul.ie/~mel/docs/stream-api/, April 29th, 2009.

[25] W. Stallings, “Operating Systems: Internals and Design Principles (5th Edition),” Prentice Hall, pp. 644, 2006.

[26] x86info, http://codemonkey.org.uk/projects/x86info, 2012.

[27] Avogadro, http://avogadro.openmolecules.net/wiki/Main_Page, 2012

[28] K. Yoshii, “Regular Memory v.s. HugeTLBFS”, Argonne National Laboratory Technical Report, http://www.mcs.anl.gov/~kazutomo/hugepage/x86laptop.html, 2006.

[29] M. Gorman, “Transparent Hugepage Support #33,” Technical Report, http://lwn.net/Articles/423590, December 20th, 2010.

[30] P. Beckman, K. Iskra, K. Yoshii, S. Coghlan, A. Nataraj, “Benchmarking the Effects of Operating System Interference on Extreme-Scale Parallel Machines,” Cluster Computing, v.11, pp. 3-6, 2008.

[31] M. Talluri, S. Kong, M. D. Hill, D. A. Patterson, “Tradeoffs in Supporting Two Page Sizes,” In Proceedings of the 19th Annual International Symposium on Computer Architecture, 1992.

[32] R. Noronha, “Improving Scalability of OpenMP Applications on Multi-core Systems Using Large Page Support,” Parallel and Distributed Processing Symposium, IPDPS, pp. 1-8, 2007.

[33] K. Yoshii, K. Iskra, P. C. Broekema, H. Naik, P. Beckman, “Characterizing the Performance of Big Memory on Blue Gene Linux”, Technical Report, Argonne National Laboratory, 2009.

[34] M. Gorman, P. Healy, “Performance Characteristics of Explicit Superpage Support,” Sixth Annual Workshop on the Interaction between Operating Systems and Computer Architecture (WIOSCA), 2010.

[35] R. Rex, F. Mietke, W. Rehm, et al. “Improving Communication Performance on Inifiniband by Using Efficient Data Placement,” In Proceedings of Cluster 06, Barcelona, Spain, 2006.

[36] M. Gorman, “Benchmarking with Huge Pages”, Technical Report, http://lwn.net/Articles/378641, October 19th 2012.

[37] J. Jensen, “Molecular Modeling Basics,” CRC Press, 2010.

[38] T. Lu, W. Perrizo, Y. Wang, G. Wettstein, “Extensino study on item-based P-Tree Collaborative Filtering Algorithm for Netflix Prize,” ISCA, pp. 149-154, 2010.

[39] libhugetlbfs, http://libhugetlbfs.sourceforge.net.

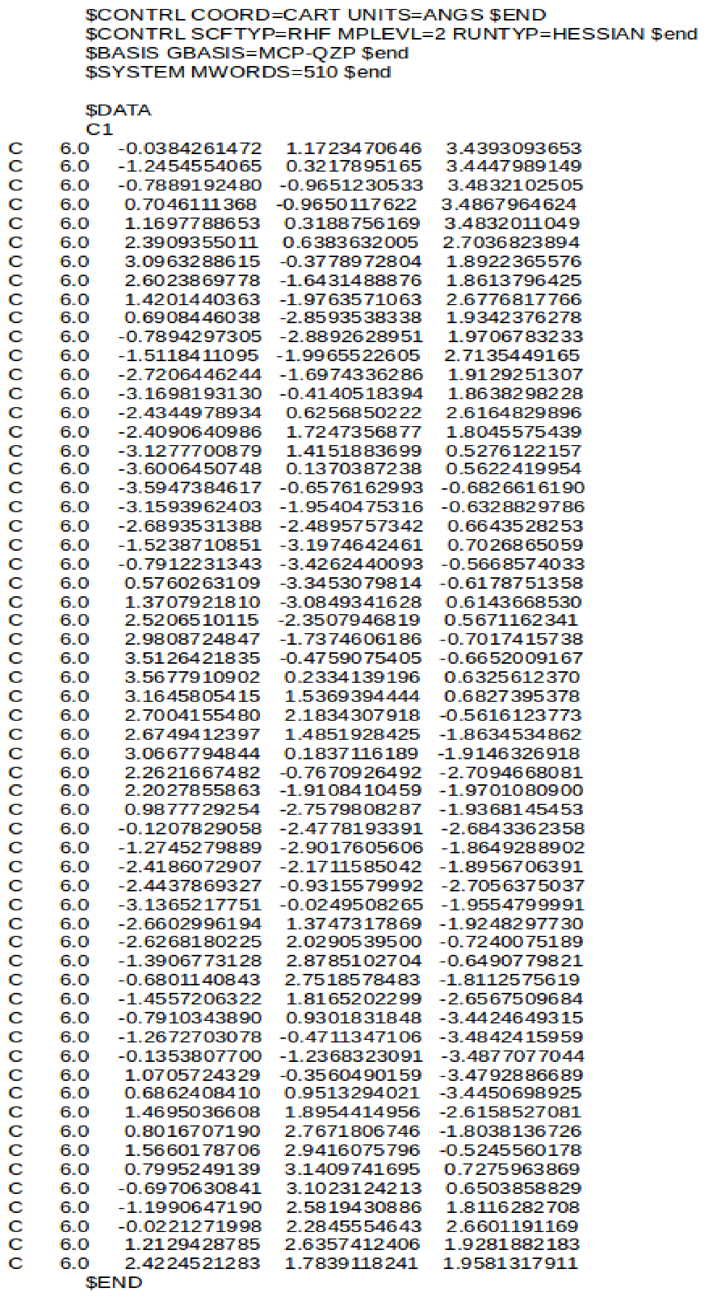
8. APPENDIX

Table A.1. HPL input file

HPL.out	# output file name (if any)
6	# device out (6=stdout,7=stderr,file)
1	# of problems sizes (N)
10000	# Ns
1	# of NBs
128	# NBs
0	PMAP process mapping (0=Row-,1=Column-major)
1	# of process grids (P x Q)
4	# Ps
2	# Qs
16	# threshold
1	# of panel fact
2	# PFACTs (0=left, 1=Crout, 2=Right)
1	# of recursive stopping criterium
4	# NBMINs (>= 1)
1	# of panels in recursion
2	# NDIVs

Table A.1. HPL input file (continued)

HPL.out	# output file name (if any)
1	# of recursive panel fact.
1	# RFACTs (0=left, 1=Crout, 2=Right)
1	# of broadcast
1	BCASTs (0=1rg,1=1rM,2=2rg,3=2rM,4=Lng,5=LnM)
1	# of lookahead depth
1	# DEPTHs (>=0)
2	# SWAP (0=bin-exch,1=long,2=mix)
64	# swapping threshold
0	# L1 in (0=transposed,1=no-transposed) form
0	# U in (0=transposed,1=no-transposed) form
1	# Equilibration (0=no,1=yes)
8	# memory alignment in double (> 0)



```
          $CONTRL COORD=CART UNITS=ANGS $END
          $CONTRL SCFTYP=RHF MPLEVL=2 RUNTYP=HESSIAN $end
          $BASIS GBASIS=MCP-QZP $end
          $SYSTEM MWORDS=510 $end

          $DATA
          C1
C     6.0    -0.0384261472   1.1723470646   3.4393093653
C     6.0    -1.2454554065   0.3217895165   3.4447989149
C     6.0    -0.7889192480  -0.9651230533   3.4832102505
C     6.0     0.7046111368  -0.9650117622   3.4867964624
C     6.0     1.1697788653   0.3188756169   3.4832011049
C     6.0     2.3909355011   0.6383632005   2.7036823894
C     6.0     3.0963288615  -0.3778972804   1.8922365576
C     6.0     2.6023869778  -1.6431488876   1.8613796425
C     6.0     1.4201440363  -1.9763571063   2.6776817766
C     6.0     0.6908446038  -2.8593538338   1.9342376278
C     6.0    -0.7894297305  -2.8892628951   1.9706783233
C     6.0    -1.5118411095  -1.9965522605   2.7135449165
C     6.0    -2.7206446244  -1.6974336286   1.9129251307
C     6.0    -3.1698193130  -0.4140518394   1.8638298228
C     6.0    -2.4344978934   0.6256850222   2.6164829896
C     6.0    -2.4090640986   1.7247356877   1.8045575439
C     6.0    -3.1277700879   1.4151883699   0.5276122157
C     6.0    -3.6006450748   0.1370387238   0.5622419954
C     6.0    -3.5947384617  -0.6576162993  -0.6826616190
C     6.0    -3.1593962403  -1.9540475316  -0.6328829786
C     6.0    -2.6893531388  -2.4895757342   0.6643528253
C     6.0    -1.5238710851  -3.1974642461   0.7026865059
C     6.0    -0.7912231343  -3.4262440093  -0.5668574033
C     6.0     0.5760263109  -3.3453079814  -0.6178751358
C     6.0     1.3707921810  -3.0849341628   0.6143668530
C     6.0     2.5206510115  -2.3507946819   0.5671162341
C     6.0     2.9808724847  -1.7374606186  -0.7017415738
C     6.0     3.5126421835  -0.4759075405  -0.6652009167
C     6.0     3.5677910902   0.2334139196   0.6325612370
C     6.0     3.1645805415   1.5369394444   0.6827395378
C     6.0     2.7004155480   2.1834307918  -0.5616123773
C     6.0     2.6749412397   1.4851928425  -1.8634534862
C     6.0     3.0667794844   0.1837116189  -1.9146326918
C     6.0     2.2621667482  -0.7670926492  -2.7094668081
C     6.0     2.2027855863  -1.9108410459  -1.9701080900
C     6.0     0.9877729254  -2.7579808287  -1.9368145453
C     6.0    -0.1207829058  -2.4778193391  -2.6843362358
C     6.0    -1.2745279889  -2.9017605606  -1.8649288902
C     6.0    -2.4186072907  -2.1711585042  -1.8956706391
C     6.0    -2.4437869327  -0.9315579992  -2.7056375037
C     6.0    -3.1365217751  -0.0249508265  -1.9554799991
C     6.0    -2.6602996194   1.3747317869  -1.9248297730
C     6.0    -2.6268180225   2.0290539500  -0.7240075189
C     6.0    -1.3906773128   2.8785102704  -0.6490779821
C     6.0    -0.6801140843   2.7518578483  -1.8112575619
C     6.0    -1.4557206322   1.8165202299  -2.6567509684
C     6.0    -0.7910343890   0.9301831848  -3.4424649315
C     6.0    -1.2672703078  -0.4711347106  -3.4842415959
C     6.0    -0.1353807700  -1.2368323091  -3.4877077044
C     6.0     1.0705724329  -0.3560490159  -3.4792886689
C     6.0     0.6862408410   0.9513294021  -3.4450698925
C     6.0     1.4695036608   1.8954414956  -2.6158527081
C     6.0     0.8016707190   2.7671806746  -1.8038136726
C     6.0     1.5660178706   2.9416075796  -0.5245560178
C     6.0     0.7995249139   3.1409741695   0.7275963869
C     6.0    -0.6970630841   3.1023124213   0.6503858829
C     6.0    -1.1990647190   2.5819430886   1.8116282708
C     6.0    -0.0221271998   2.2845554643   2.6601191169
C     6.0     1.2129428785   2.6357412406   1.9281882183
C     6.0     2.4224521283   1.7839118241   1.9581317911
          $END
```

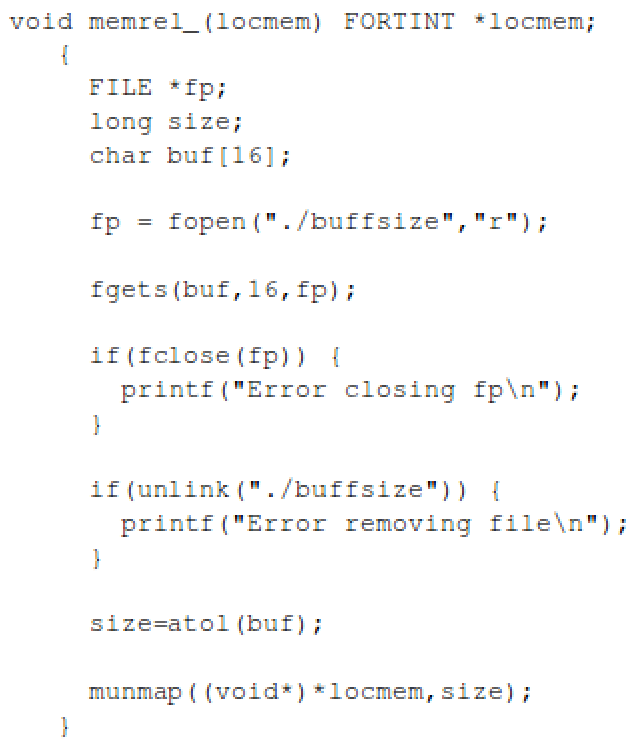
Figure A.1. GAMESS input file

```
/* BEGIN MPIEHL */
#include <stdio.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <unistd.h>
#include <sys/mman.h>
#include <fcntl.h>

#ifndef MAP_HUGETLB
#define MAP_HUGETLB 0x40000 /* arch specific */
#endif

#define HugePage (1024*1024*2)

size_t dsize = (size_t)(lwork) * sizeof( double );

size_t remainder;

if(dsize >= HugePage) {

    remainder = dsize % HugePage;

    dsize += HugePage - remainder;

    PANEL->WORK = mmap(NULL, dsize, PROT_READ | PROT_WRITE,
            MAP_PRIVATE | MAP_ANONYMOUS | MAP_HUGETLB, 0, 0);
    PANEL->dsize = dsize;
}
else {

  if(!(PANEL->WORK = (void *)malloc((size_t)(lwork)*sizeof(double))))
      {
         HPL_pabort( __LINE__, "HPL_pdpanel_init",
                     "Memory allocation failed" );
      }
}
/* END MPIEHL */
```

Figure A.2. HPL allocation code sample

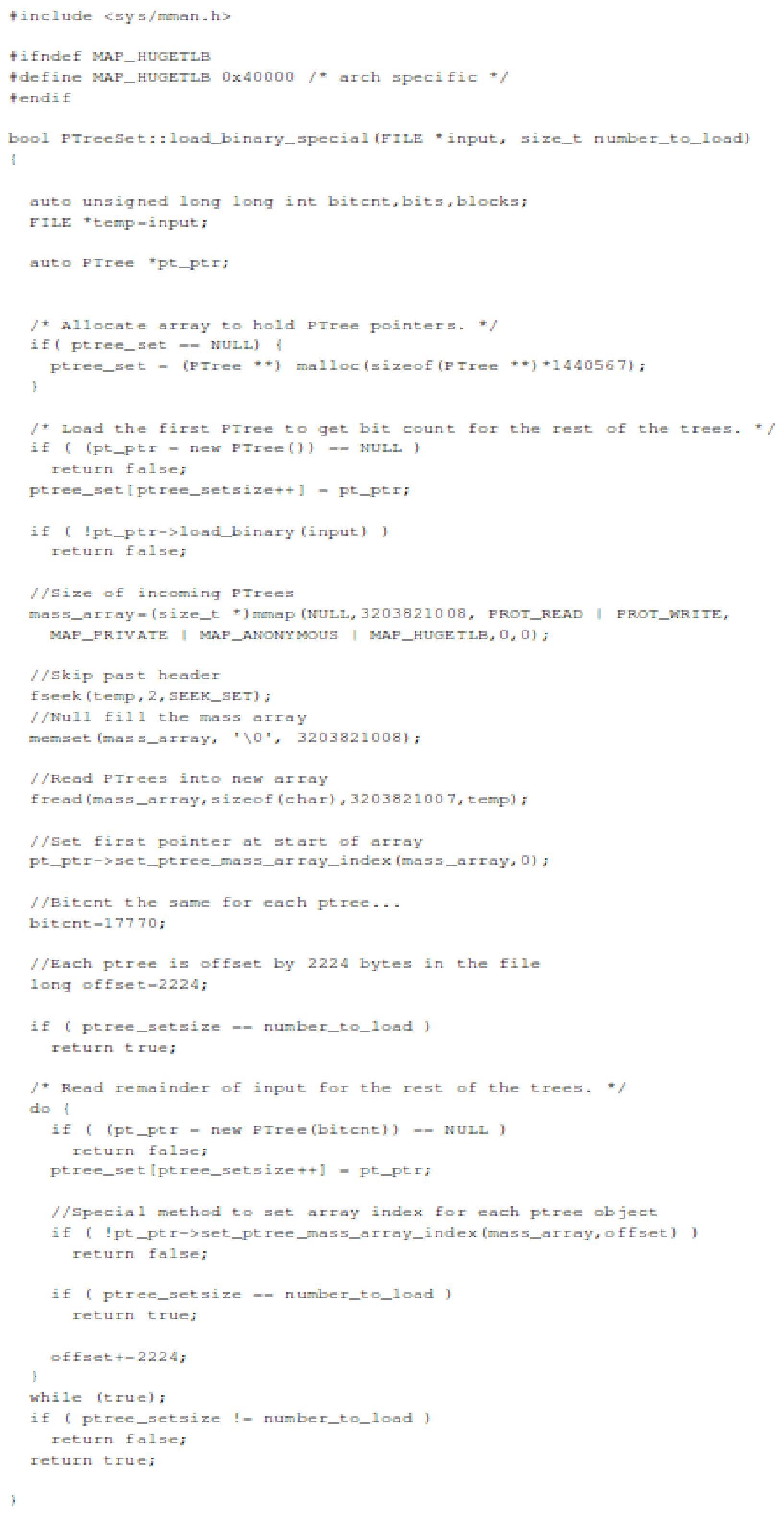
```
#ifdef LINUX64

#include <stdlib.h>
#include <stdio.h>
#include <malloc.h>
#include <sys/mman.h>

#define FORTINT long
#define HugePage (1024*1024*2)

#ifndef MAP_HUGETLB
#define MAP_HUGETLB 0x40000 /*x86 specific HUGETLB flag */
#endif

FORTINT memget_(nwords) FORTINT *nwords;
   {
     size_t nbytes;
     size_t remainder;
     FILE *fp;

     nbytes = (*nwords+2)*8;

     remainder = nbytes % HugePage;

     nbytes+=remainder;

     fp = fopen("./buffsize","w");

     fprintf(fp,"%d",nbytes);

     if(fclose(fp)) {
       printf("Error closing fp\n");
     }

     return (FORTINT) mmap(NULL,nbytes,PROT_READ | PROT_WRITE,
            MAP_PRIVATE | MAP_ANONYMOUS | MAP_HUGETLB, 0, 0);
   }
```

Figure A.3. GAMESS allocation code sample



```
void memrel_(locmem) FORTINT *locmem;
   {
     FILE *fp;
     long size;
     char buf[16];

     fp = fopen("./buffsize","r");

     fgets(buf,16,fp);

     if(fclose(fp)) {
       printf("Error closing fp\n");
     }

     if(unlink("./buffsize")) {
       printf("Error removing file\n");
     }

     size=atol(buf);

     munmap((void*)*locmem,size);
   }
```

Figure A.4. GAMESS free code sample



```
#include <sys/mman.h>

#ifndef MAP_HUGETLB
#define MAP_HUGETLB 0x40000 /* arch specific */
#endif

bool PTreeSet::load_binary_special(FILE *input, size_t number_to_load)
{

  auto unsigned long long int bitcnt,bits,blocks;
  FILE *temp=input;

  auto PTree *pt_ptr;


  /* Allocate array to hold PTree pointers. */
  if( ptree_set == NULL) {
    ptree_set = (PTree **) malloc(sizeof(PTree **)*1440567);
  }

  /* Load the first PTree to get bit count for the rest of the trees. */
  if ( (pt_ptr = new PTree()) == NULL )
    return false;
  ptree_set[ptree_setsize++] = pt_ptr;

  if ( !pt_ptr->load_binary(input) )
    return false;

  //Size of incoming PTrees
  mass_array=(size_t *)mmap(NULL,3203821008, PROT_READ | PROT_WRITE,
    MAP_PRIVATE | MAP_ANONYMOUS | MAP_HUGETLB,0,0);

  //Skip past header
  fseek(temp,2,SEEK_SET);
  //Null fill the mass array
  memset(mass_array, '\0', 3203821008);

  //Read PTrees into new array
  fread(mass_array,sizeof(char),3203821007,temp);

  //Set first pointer at start of array
  pt_ptr->set_ptree_mass_array_index(mass_array,0);

  //Bitcnt the same for each ptree...
  bitcnt=17770;

  //Each ptree is offset by 2224 bytes in the file
  long offset=2224;

  if ( ptree_setsize == number_to_load )
    return true;

  /* Read remainder of input for the rest of the trees. */
  do {
    if ( (pt_ptr = new PTree(bitcnt)) == NULL )
      return false;
    ptree_set[ptree_setsize++] = pt_ptr;

    //Special method to set array index for each ptree object
    if ( !pt_ptr->set_ptree_mass_array_index(mass_array,offset) )
      return false;

    if ( ptree_setsize == number_to_load )
      return true;

    offset+=2224;
  }
  while (true);
  if ( ptree_setsize != number_to_load )
    return false;
  return true;

}
```

Figure A.5. PTree allocation code sample