Memory Management using Dynamic Memory Switching

CS499 – Project Report

by

Sharad Chole and Sanchay Harneja
Y3101, Y3307

Under the guidance of

Prof. Rajat Moona

Department of Computer Science and Engineering
Indian Institute of Technology Kanpur
May 2007

Certificate

This is to certify that the work contained in this report entitled "*Memory Management using Dynamic Memory Switching*", by Sharad Chole(Y3101) and Sanchay Harneja (Y3307), has been carried out under my supervision and that this work has not been submitted elsewhere for a degree.

May, 2007.

[illegible]

(Dr. Rajat Moona)
Department of Computer Science and Engineering,
Indian Institute of Technology, Kanpur.

Abstract

Memory can be efficiently utilized if the memory demands can be analyzed at run-time and satisfied using dynamic memory switching. To provide a base for our implementation we first do static analysis of system's performance versus amount of available memory. We also study the fragmentation of memory and usage of memory in certain systems. Then we implement a kernel thread to dynamically control availability of memory depending on system's need. Further our implementation provides support for reducing power consumed by memory. We also give an approach to calculate the most suitable amount of memory for a given system given its performance requirements.

1. INTRODUCTION

This project studies the behavior of a given system (a system here is defined as: a fixed platform, OS, and applications) as regards to its memory requirements. Specifically, given some performance measure, we try to utilize memory efficiently. Performance measures can be defined in various ways, for example, either time taken to execute the program or benchmark, or it could be throughput in terms of number of bytes read, or it could as simple as the total number of page faults. We define our performance measure to be the page-cache hit ratio and power consumed by memory.

1.1 Motivation

We know that low memory hampers the performance of any system, but intuitively, memory above a certain point doesn't increase the performance of the system by much. We want to test this intuition and understand the relationship between performance degradation versus savings, power.

"If by giving extra memory to a process, the improvement in performance is marginal, it is better to power down the extra memory to conserve energy. Optimizing energy consumption of computing components has become important not only for mobile, wireless and embedded devices due to short battery life, but also for high-end systems to reduce electricity bills."[8]

It has been observed by previous studies [9, 10, 11, 12] that the memory subsystem is one of the dominant consumers of the overall system energy. "Recent measurements from real server systems show that memory consumes as much as 50% more power than processors."[13] To address this problem, many modern memory modules support multiple low-power operating modes to conserve energy [14].

1.2 Work done

In the first semester we did static analysis of the memory usage on Linux kernel. By static analysis we mean, changing the total memory available to the system at the boot time, and then plotting performance versus the total memory. We ran several benchmarks and also made bitmaps of the whole memory to aid to our understanding of the memory usage.

As this static analysis is impractical for systems (i.e. we cannot reboot system to change memory while application is running), we want the system to be able to decide for itself its most suitable amount of memory.

In the second semester we have implemented dynamic memory switching for Linux kernel (specifically 2.6.15). First of all we divide the whole memory into banks (contiguous segments of same size). Then using a page migration algorithm we can migrate pages from one bank to another. A kernel thread runs, which monitors the total memory usage of the system, if it finds reclaimable pages (PageCached and free) over a certain limit, it migrates pages such that a whole bank becomes free, and then it turns this bank off. This switched off bank is invisible to the system (i.e. this memory cannot be used for any purpose). If the system needs memory beyond what it already has, the kernel thread turns the switched off banks on, and this bank can be used as usual like free memory. On a longer run this thread can plot the number of page faults (or page cache hit ratio) versus the amount of memory available (number of banks which are on) from which we can extract the most suitable amount of memory for this system.

2 Static Analysis

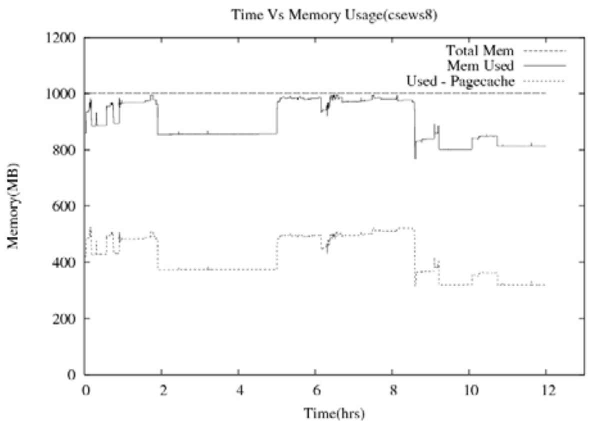
In this section we try to analyze systems behavior by statically observing the system and if necessary using reboot to change system memory.

2.1 Memory Usage Patterns

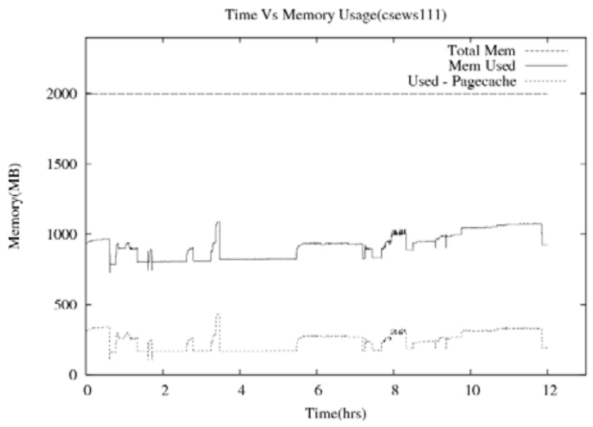
Before starting to implement dynamic memory we wanted to get an idea of how a system uses its memory, for that we studied the meminfo and vmstat files in /proc directory. These files give several parameters like total memory, free memory, buffers, cached, active, inactive, page faults etc of the system. We chose to monitor a few of these:

MemTotal:	Total usable ram (i.e. Total RAM - few reserved bits - the kernel binary code)
MemFree:	Total free ram (The sum of LowFree + HighFree)
Buffers:	Mem used by BufferCache (Relatively temporary storage for raw disk blocks)
Cached:	Cache for files read from the disk
SwapCached:	Mem used by swap cache
PageCached:	Buffers+Cached+SwapCached (This PageCached is reclaimable)
Pgfault:	Faults (major+minor)

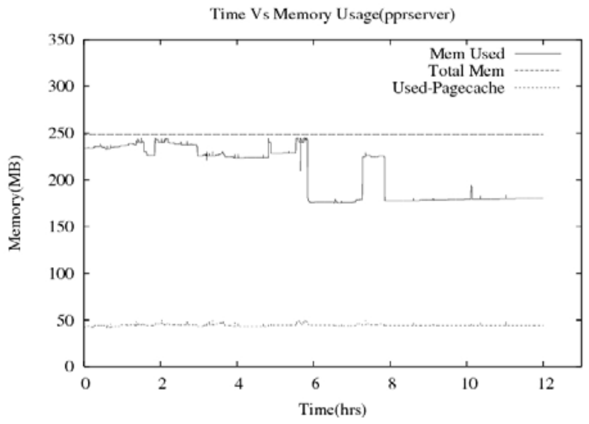
We ran a program for 12 hours on few of the select systems, recording meminfo and vmstat every 5 secs. We then plotted the above parameters against time, and also calculated their average. These graphs and table give us an indication as how the systems use up their memory.



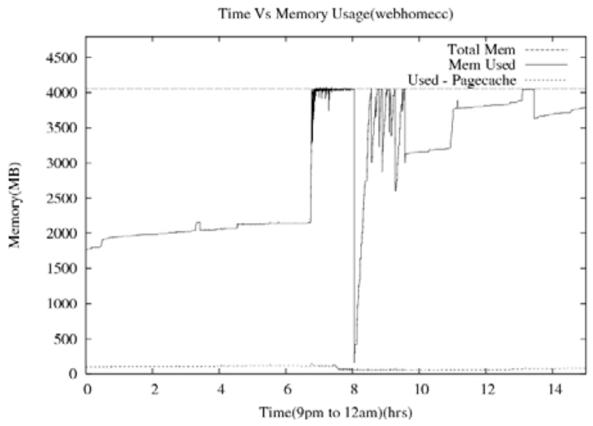
a) Time versus Memory Usage



b) Time versus Memory Usage



c) Time versus Memory Usage



d) Time versus Memory Usage

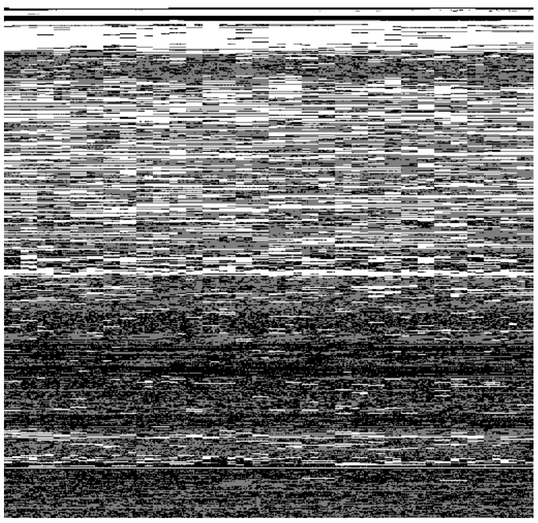
TABLE 1: This table gives the average of the above parameters over a 12 hour period (All figures are approximated to the nearest MB)

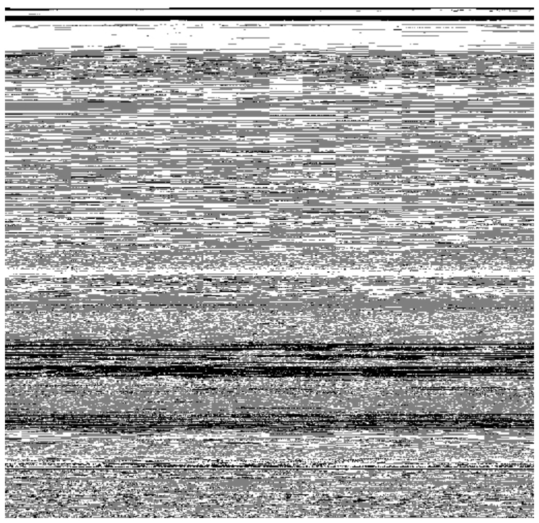
Computer Name	Total Mem	Free Mem	Used Mem	PageCached	Used-Pagecached
cseserver1	4056	2800	1256	1090	166
csews100	1999	791	1208	891	317
csews102	1999	1525	474	221	253
csews103	1999	1215	784	548	236
csews106	1999	694	1305	995	310
csews111	1999	1080	919	674	245
csews112	1999	129	1870	1358	512
csews12	1001	30	971	465	506
csews14	1001	234	767	432	335
csews15	1001	62	939	542	397
csews16	1001	114	887	527	360

2.2 Memory Bitmaps

Memory bitmap is bitmap of memory taken per page, i.e. a bit in the bitmap shows status of a particular page in physical memory.

The bitmap image is a grayscale image with one pixel per page stored in row major format. This is constructed by traversing the /proc entry and assigning the following color values:
Free=White, PageCache=Grey, Slab or Reserved or Non-Free=Black
Following are bitmaps of a machine with 1 GB ram at different times.



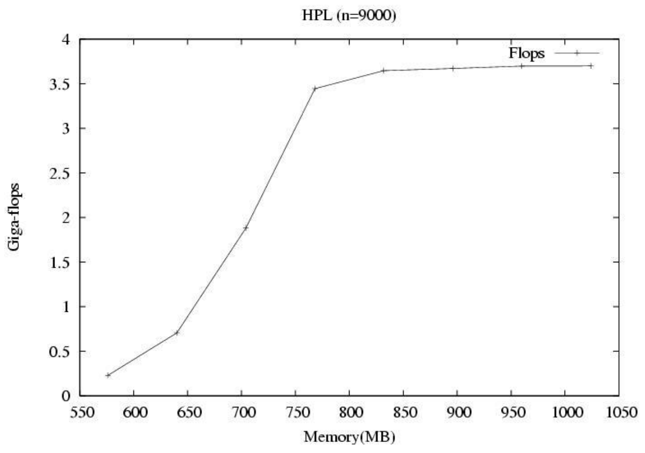


2.3 Effect of changing memory on performance

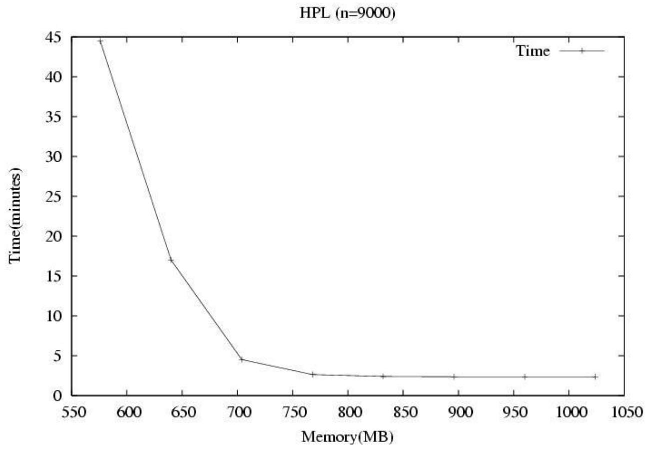
To see the effect of varying physical memory on the performance we ran some standard and synthetic benchmarks like HPL, TPCC, Quick Sort, Java Garbage Collection, Kernel Compilation etc on specific machines. We changed the amount of physical memory by giving a boot-time option in the grub boot-loader and ran benchmarks each time. Some of the benchmarks and there performance statistics are given below.

2.3.1 HPL Benchmark

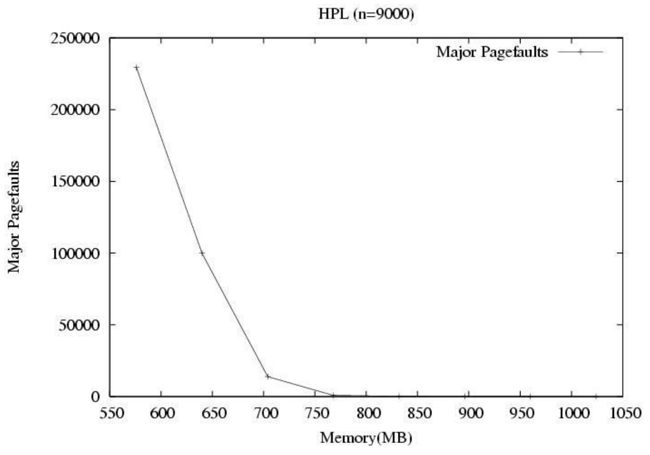
HPL is a software package that solves a (random) dense linear system in double precision (64 bits) arithmetic on distributed-memory computers. It solves a linear system of order n: A x = b.
Performance measure: Floating operations per second.
Input: A fixed value for the rank of the linear equation, we chose it to be 9000(n = 9000).
Platform: P4 2.8 GHz, 1 GB RAM, Kernel 2.6.5-1.358, Fedora Core 2.
Following graphs give us Performance Measures versus Memory. Here we can see relation between performance and pagefaults.



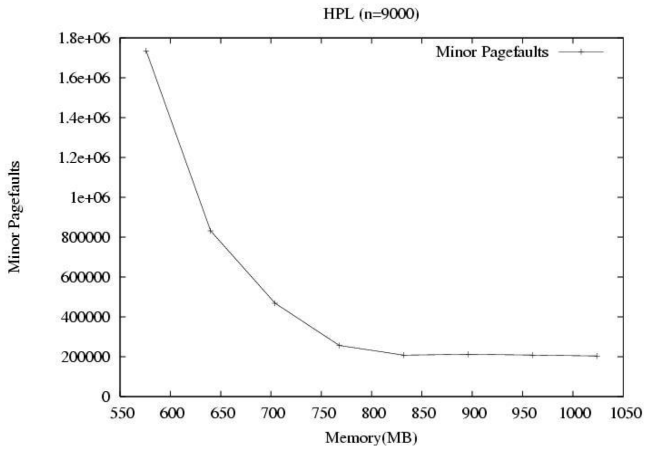
a) Giga Floating point operations versus Memory



b) Time (minutes) versus Memory



c) Major Pagefaults versus Memory



d) Minor Pagefaults versus Memory

2.3.2 TPCC Benchmark

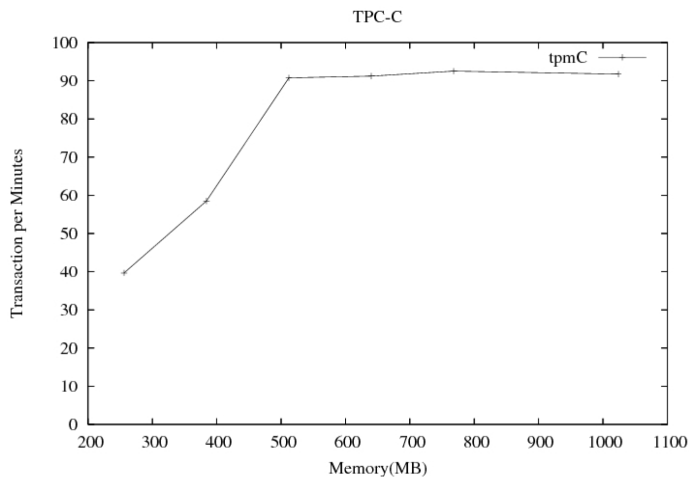
The TPC-C benchmark is a well-know benchmark used to measure the performance of high-end systems. TPC-C simulates the execution of a set of distributed, on-line transactions (OLTP), for a period between two and eight hours.

Performance measure: TPC-C transactions per minute (tpmC).

Input: 8 warehouses with 10 terminals per warehouse, measurement interval of 2 hours.

Platform: AMD 64 1.8ghz, 1 GB ram, 2.6.15 kernel, FC5

Following graph was obtained after plotting tpmC versus amount of memory.



2.3.3 Quick-Sort

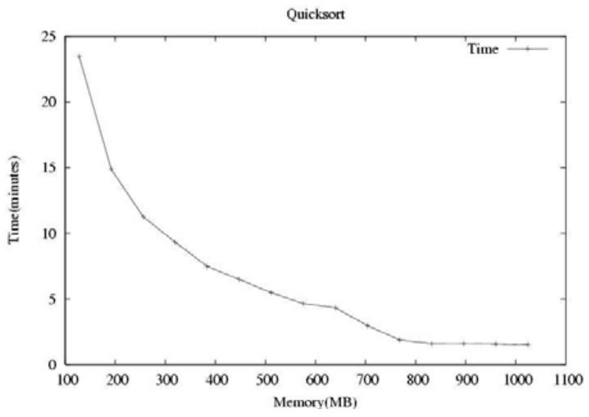
The qsort() function is an implementation of C.A.R. Hoare's quick sort algorithm, a variant of partition-exchange sorting. It has O(nlogn) average performance; this implementation uses median selection to avoid the O(n^2) worst-case performance of quick sort

Performance measure: The time to sort.

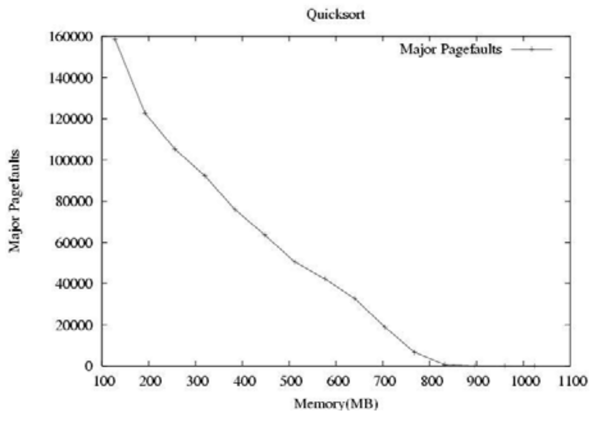
Input: We used integer array of size ~750MB to perform quick sort.

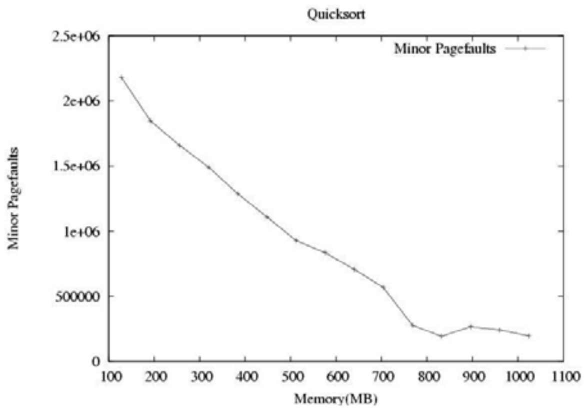
Platform: P4 2.8 GHz, 1 GB RAM, Kernel 2.6.5-1.358, Fedora Core 2

Following graphs give us Performance Measures versus Memory:



a) Time (minutes) versus Memory





b) Major Pagefaults versus Memory **c)** Minor Pagefaults versus Memory

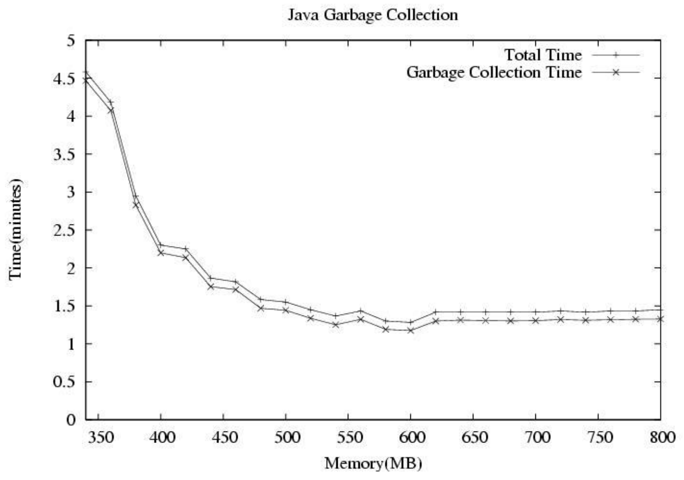
2.3.4 Java Garbage Collection

This is a synthetic benchmark. In this we take an array of linked list of some size. In a single iteration, we dump the linked list in every array index and make a new linked list at that index. This is repeated for a few iterations. This results in garbage at each step which has to be collected by the Java Garbage Collection.

Performance measure: Time for execution (garbage collection).

Platform: P4 2.8 GHz, 1 GB RAM, Kernel 2.6.5-1.358, Fedora Core 2

Following graph give us Time versus Memory:



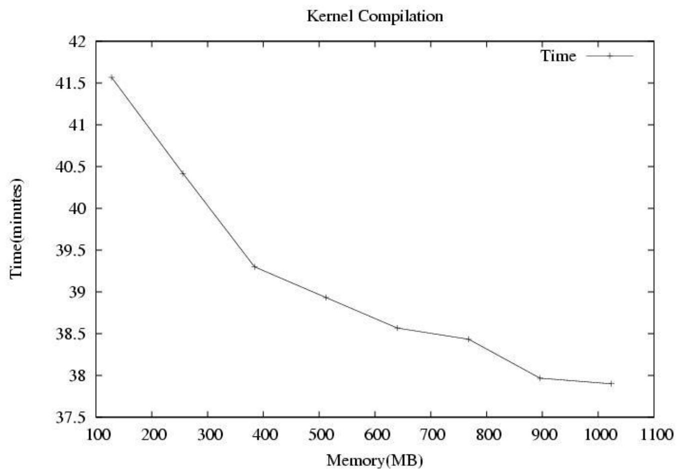
a)Time execution (garbage collection) versus Memory

2.3.5 Kernel Compilation

This is also a synthetic test. We measure the time taken to compile the linux kernel.

Platform: Pentium D 1.8 GHz, 2 GB ram, 2.6.15 kernel, Fedora Core 5

Following graph give us Compilation Time versus Memory



a) Compilation Time versus Memory

3 Implementing Dynamic Memory Switching

The following steps constitute our whole process of implementing dynamic memory switching in a linux kernel.

1. Introducing a new kernel zone, allocating kernel pages in it and making it dynamic.
2. Implementing page migration.
3. Creating banks & implementing bank turning off and on.
4. Making a kernel daemon which automatically turns banks on/off.

3.1 Introducing a new kernel zone

Slab allocations are contiguous in memory (size more than page-size). So we cannot migrate slab pages using page migration based on single page. As we cannot reallocate slab objects we need to allocate them into some region of memory which we will never turn off. A simple solution lies in creating a new zone for allocation of kernel-pages (kernel-page is pages allocated using GFP_KERNEL flag).

a. GFP-flags are used to allocate pages in potentially best region of memory. There are flags defined for each zone of memory. We introduced a new flag called __GFP_KERNEL corresponding to the new zone.
b. To initialize the zone we need to define (or calculate) its size. It should be defined equal to maximum size of slab. We defined it to be 112 MB. So now memory is fragmented into following zones

1.	DMA	0-16 MB
2.	Kernel	16-128 MB
3.	Normal	128-896 MB
4.	Highmem	896-4 GB

c. Now to allocate all kernel-pages in zone_kernel we add our zone flag to GFP_KERNEL. Now when a page is allocated using GFP_KERNEL flag it is situated either in zone_dma or zone_kernel i.e. below 128 MB limit. As all the non-migrating pages are below 128MB we can turn off memory above 128MB. Zone_kernel can be made specific to only kernel-pages i.e. changes can be made so that only kernel-pages are allocated in zone_kernel.

3.2 Implementing page migration

The migration function takes pointers to source page and destination page. Implementation is based on migration algorithms for following cases.

1. When page is pointed by one or more page table entries we need to change all of them to point to destination page. We use reverse mapping to find page table entries form page pointers.
2. When page is in swap-cache or page-cache we delete it from cache and add destination page in respective cache.
3. When page is in buffer-cache, we remove it from corresponding buffer_heads and add destination page into it.
4. When page is in slab-cache or is in writeback or is locked we do nothing.

We copy relevant page flags and other fields to destination page. Last step is to copy the source page data into the destination page. Note that the algorithm should be such that it either does it all or does nothing otherwise we will end up with two copies of same page.

3.3 Creating banks and implementing bank_on, bank_off

We divide memory into contiguous segments of same size. We call this a bank. Each bank is a struct which contains information about whether bank is on or off, bank range and number of free pages in it. Captured pages are those pages which can’t be allocated to anyone on request. So when the number of free pages goes to zero, bank is turned off.

Banks are stored in an array per node. For captured pages we use captured_area defined per zone. Captured_area is used to capture pages using buddy system (exactly similar to buddy allocator for free pages). This puts constraint on bank size to be multiple of 4MB. This implementation makes it easy to exchange captured and free pages.

We have written two system calls, one for turn off and other for turn on. Both calls take bank as argument.

a. Turn off:
    1. Capture all free pages from buddy which are in the bank.

2. Hold zone lock and isolate all LRU pages (i.e. isolate pages in active list and inactive list that belongs to bank.)
3. Try to migrate pages. If migration was successful, capture pages.
4. Calculate number of captured pages in this bank. If it is equal to bank size, return success.

b. Turn on:
For each captured page in bank, free it.

3.4 Kernel daemon

To implement dynamic memory switching in a linux kernel we have written a kernel daemon. We call it bmd (Bank Management Daemon). Its major functionality is to keep track of system's need and availability of memory and take action according to them. Actions are

1. If system needs more memory than available then turn part of memory on.
2. If availability is more than system's need then turn part of memory off.
3. If need and availability are compatible with each other then do nothing.

To understand systems memory requirement daemon needs to keep track of free memory and reclaimable memory. And as it can increase memory available to system we have to consider its effect and use in page reclamation algorithm. We implemented turning part of memory on and off using banks, so change in amount of memory will always be a multiple of bank size. Working of daemon can be explained in three steps.

1. Memory requirement:

Reclaimable pages are potential free pages as they can be reclaimed very easily and used as free pages (note that other pages in LRU lists can also be reclaimed but they will most likely cause disk write due to swapout). Reclaimable pages consist of PageCached pages (i.e. cached pages + buffer cache pages + swap cache pages). Even though all PageCached pages are potential free pages, it is not good to reclaim all PageCached pages as it might lead to a large PageCache miss ratio. Watermarks for allowable PageCached size can be defined using performance requirement of user and type of applications running on system. Similarly we have to define watermarks for free pages as too many free pages are not good for power and too few free pages are not good for performance.

As PageCached pages are potential free pages, watermark for them are decided while considering them free pages. i.e. watermark is decided for number of PageCached pages (nr_cache)+ number of free pages (nr_free). These watermarks are called CACHE_FREE_HIGH and CACHE_FREE_LOW. Similarly two watermarks for number of free pages are FREE_HIGH and FREE_LOW. Now we define actions for daemon as follows.

Action 1: If(nr_free + nr_cache < CACHE_FREE_LOW && nr_free < FREE_LOW)
Turn smallest indexed bank on.

Action 2: If(nr_free + nr_cache > CACHE_FREE_HIGH || nr_free > FREE_HIGH)
Turn largest indexed bank off.

Condition for action 1 can be explained as: after turning bank on neither free pages nor potential free pages are more than certain limit. Condition for action 2 is either free pages or potential free pages are more than certain limit. So, allowable PageCached size is bounded by CACHE_FREE_HIGH.

Note that to avoid clashing, difference between pair of watermarks should be greater than bank size i.e. FREE_HIGH should be greater than FREE_LOW + bank size. Otherwise as soon as we turn bank on (action 1) condition for turning bank off (action 2) is satisfied and vise versa.

2. Page reclamation algorithm:

The daemon which implements page reclamation is called kswapd. As LRU lists are stored zonewise, kswapd also works per zone as it needs to scan LRU lists. But the all data-structures for implementing bmd are stored per node. So we need to somehow communicate changes done by bmd to kswapd.

As the job of kswapd is to increase number of free pages in zone (above certain limit) it should call action 1 from our daemon if the condition is satisfied. Now action 2 needs free pages and this need may wake up kswapd. Again if condition for action 1 is satisfied kswapd will call action 1. Now we are simultaneously turning bank on and off. To avoid this race condition we have to use locks per node.

3. Other Parameters:

a. **Timeout** is time after which bmd will be rescheduled. It affects both performance and power. If timeout is too small bmd will be rescheduled much frequently. This will certainly increase load on system. It may also cause to turning memory on and off too frequently. As both this actions are quite heavy this will certainly decrease system’s performance. If timeout is too large effect of change in system’s memory requirement on power will be very late. Timeout should be decided considering how frequently system’s memory requirement changes and how much cpu-time a turning bank off operation needs.

b. **Bank_size** is size of a bank. If it is too small them it may cause too may bank turn on and off. If it is too large then precision to which we manage power will decrease. So it should be determined considering smallest block size on RAM that can be turned off and the frequency of turning bank off. (The Rambus rdram has memory with bank_size of 128MB, 32MB, 16MB [2,8])

3.5 Power Considerations

Rambus RDRAM differs from the rest by allowing a finer-grained unit of control in power management, i.e. it has bank architecture and has support to change power to individual banks.

Following tables give specifications of Rambus RDRAM:

Power State/Transition	Power	Time	Active Components
Active	300mW	-	Refresh, clock, row, col decoder
Standby	180mW	-	Refresh, clock, row decoder
Nap	30mW	-	Refresh, clock
Powerdown	3mW	-	Refresh
Standby To Active	240mW	+6ns	
Nap To Active	160mW	+60ns	
Powerdown To Active	150mW	+6000ns	

Switching a bank off means that a bank is transitions from Active to Nap, similarly bank on means transition from Nap to Active.

An Example:

Now an example of how much power we can save. Consider all 2 GB machines in table 1. Average value of Used – PageCached is 310 MB. Now if we set watermark CACHE_FREE_HIGH = 200MB, we get bound on PageCached equal to 200MB (this will decrease some performance due to less cache hits). So, average total use of memory is bounded by 510 MB. This means on an average 3/4th of memory is switched off. So, total power consumed by memory will be 33% of original (considering nap and active modes instead of total on and off). 2GB RDRAM consumes approximately 16 W. Out of which we are saving 10 W. A general desktop pc uses approximately 150 W. Hence we may save approximately 7% of systems power.

Note, here we are not claiming that we can save 7% of systems power in our cse lab. This claim would require much more data (probably duration of months) collected over different seasons on majority of cse machines. Again limiting value for PageCache chosen here is random. To get exact limit we need to have understanding of behavior of PageCache hit ratio depending on PageCache size and our expectation for performance.

4. FUTURE WORK

1. As slab pages can only be allocated below zone_kernel we need to keep some extra space so that in future all requests are satisfied. A better way to do this will be to make the boundary between zone_normal and zone_kernel dynamic.
2. In page migration algorithm when a page is in only in cache (and its not dirty) and memory is almost full we should just delete it, instead of copying somewhere else as it would evoke page reclamation daemon and may cause a disk write. We also need to consider TLB-flushes for better implementation.
3. Writing a /proc entry for changing the bank_on, bank_off watermarks.
4. Finding PageCache hit ratio in the kernel.

5. Writing a user level program which changes watermarks and scans PageCache hit ratio.
6. Plotting PageCache hit ratio versus watermarks, active memory versus time. This active memory versus time graph gives us the most suitable amount of memory of the system.
7. Extensive testing for the suitable values of various parameters.

5. CONCLUSION

These conclusions follow from our static analysis:

1. Some of the memory in a typical system is used by the PageCached (which is reclaimable, i.e. even if it were not there the system will continue to function, although with a performance degradation), and relatively lesser amount of the total memory is being used by the processes. So there is scope that we should be able to reduce the total amount of memory without affecting the performance by much.

2. Using memory bitmaps we found that the free pages are scattered around in the physical memory. Thus we have to migrate the free pages using some ‘page migration algorithm’ so that they become contiguous in the physical memory, to be able to switch (that bank) on/off.

3. By running some benchmark tests we found that after a certain limit, increasing the size of physical memory does not improve the performance by a significant amount. The loss in performance when the memory is low can be accredited to the increase in major and minor page faults. Thus if the work to be done on a system in known from advance, we may assign that system suitable amount of memory according to its need.

From dynamic memory switching we may conclude that using our implementation, for a given system, given sufficient time we can get suitable amount of memory for the system, as governed by the user’s requirements and the system applications. Also our implementation may be used to reduce the power consumption of the memory subsystem given the hardware support [14].

From the active memory versus time graph (here active memory means the total number of banks that are in on state at a given time) the user may deduce the most suitable amount of memory as he deems fit. Of course a user may change the watermarks if he feels that the PageCache hit ratio is less, thus affecting active memory.

6. REFRENCES

1. Pratap Ramamurthy, Ramanathan Palaniappan: Performance Directed Energy Management using BOS, proceedings of SLOW'05.
   http://www.cs.wisc.edu/~remzi/Classes/736/Spring2005/Projects/Pratap-Ram/bos.pdf
2. Hai Huang, Padmanabhan Pillai and Kang G. Shin: Design and implementation of Power – Aware Virtual Memory, proceedings of USENIX 2003 Annual Technical Conference.
   http://www.usenix.org/events/usenix03/tech/full_papers/huang/huang_html/index.html
3. Mel Gorman, Understanding the Linux Virtual Memory Manager.
   http://www.skynet.ie/~mel/projects/vm/guide/pdf/understand.pdf
4. HPL Benchmark
   http://www.netlib.org/benchmark/hpl/
5. TPCC Benchmark
   http://www.infor.uva.es/~diego/tpcc-uva.html
   http://www.tpc.org/tpcc/detail.asp
6. Cross-Referencing Linux – a Linux kernel code browser.
   http://lxr.linux.no/blurb.html
7. Bruno Diniz, Dorgival Guedes, Wagner Meira Jr., Ricoardo Bianchini: Limiting the power consumption of main memory. Proceedings of the International Symposium on Computer Architecture (ISCA), June 2007.
   http://www.cs.rutgers.edu/~ricardob/papers/isca07.pdf
8. Pin Zhou, Vivek Pandey, Jagadeesan Sundaresan, Anand Raghuraman, Yuanyuan Zhou and Sanjeev Kumar: Dynamic Tracking of Page Miss Ratio Curve for Memory Management. Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2004, Boston, MA, USA, October 7-13, 2004.
   carmen.cs.uiuc.edu/paper/ASPLOS04-Zhou.pdf
9. F. Catthoor, S.Wuytack, E. Greef, F.Balasa, L.Nachtergaele, and A. Vandecappelle. Custom memory management methodology exploration of memory organization for embedded multimedia systems design, Boston, MA: Kluwer Academic Publ., 1998.
   http://www.kap.nl/book.htm/0-7923-8288-9
10. M. T. Kandemir, N. Vijaykrishnan, M. J. Irwin, and W. Ye. Influence of compiler optimizations on system power.
    Proceedings of the 11th international conference on Architectural support for programming languages and operating systems, 2004, Boston, MA, USA.
    http://portal.acm.org/citation.cfm?id=1024393.1024415
11. A. R. Lebeck, X. Fan, H. Zeng, and C. S. Ellis. Power aware page allocation. Proceedings of Architectural Support for Programming Languages and Operating Systems, 2000.
    http://www.cs.duke.edu/~alvy/papers/papa.pdf
12. N. Vijaykrishnan, M. Kandemir, M. J. Irwin, H. S. Kim, and W. Ye. Energy-driven integrated hardware-software optimizations using simplepower, ISCA 2000: pp 95-106.
    http://ieeexplore.ieee.org/iel5/6892/18551/00854381.pdf
13. C. Lefurgy, K. Rajamani, F. Rawson, W. F. elter, M. Kistler, and T. W. Keller. Energy management for commercial servers, IEEE Computer, pp. 39-48, December, 2003.
    www.research.ibm.com/people/l/lefurgy/Publications/computer2003.pdf
14. Rambus' RDRAM® memory interface.
    http://www.rambus.com/us/products/rdram/index.html