Submitted to SLOW '05

PERFORMANCE DIRECTED ENERGY MANAGEMENT USING BOS

Pratap Ramamurthy, Ramanathan Palaniappan

Computer Sciences Department
University of Wisconsin-Madison

ABSTRACT

One of the major challenges in today's computing world is energy management in portable devices and servers.Power management in portable devices is essential to increase battery life. High end server systems use large clusters of machines that consume enormous amount of power. Previous research has devised both software and hardware techniques for memory energy management but has overlooked the performance of applications in such environments. The result is that some of these techniques slowed down an application by 835%. In this paper, we look at software techniques for memory energy management without compromising on performance. The paper conceives of a new approach called BOS - Ballooning in the OS *inspired from the VMware ESX server. The BOS approach consists of a kernel daemon which continuously monitors the accesses to memory chips and disk I/O. Based on the profiled information, the BOS daemon decides about powering down/up chips. Powering down is emulated within the kernel using mechanisms such as page migration and invisible buddy. Results indicate that chips with more number of allocated pages may not always be the most frequently accessed ones. A study has been done analyzing the effect of decreased memory size on disk activity and based on the study, a threshold based policy is proposed which is found to settle in the operating point for a simple application. A single page migration incurs a cost of approximately 13*μ*s and is one of the bottlenecks in the BOS approach.*

Keywords: Ballooning, BOS, invisible buddy, power, page migration, buddy allocator.

1. INTRODUCTION

People have been trying to increase memory speeds to keep up with processor speeds. Speed is the only issue as long as we are concerned with desktops and high end servers. But when we delve into the realm of handheld devices and laptops, a new constraint comes into the picture, namely power expenditure. Mobile devices are equipped with batteries to supply power and hence, it would be better if this battery power is utilized effectively. Researchers have proposed many solutions to conserve power in system components like processor, disk drives, flash, cache and main memory from a hardware perspective, but few people have explored it from a software standpoint. This project tries to build a power and performance aware module within an operating system that controls the main memory and disk power consumption with due consideration for performance.

2. MOTIVATION

Handheld and portable devices these days come with guarantees of more than 24 hours of battery life. The size and the weight of these devices depend very much on the battery required. Hence, saving power in such devices will lead to compact battery sizes with longer lifetime. Popular internet servers like google.com and search.msn.com have set up huge data centers called farms which dissipate lot of power and hence, special cooling techniques have to be installed even for normal operation. All these end up with additional costs. Therefore, reducing energy/power consumption by systems becomes an important issue.

Current hardware technologies allow various system components to operate at different power levels and corresponding performance levels. Hardware devices are so complex that billions and billions of transistors make up the circuit thereby increasing power consumption. This has motivated hardware vendors like Intel and RAMBUS to include power saving modes into their processors and memory chips respectively. Other places where power could be saved include the power aware displays (CRT/LCD). In the power saving mode, the device consumes less power at a reduced clock speed. One fine example is the CPU switching to low power-low performance circuits in power conservation mode. Recently there has been an increased interest in software techniques to save power in systems. The software approaches include resourceful use of devices and utilizing the hardware based power saving modes. These techniques involve decision making using heuristics and then operating in a power efficient mode. Quite obviously, all these power saving modes have a performance penalty associated with them. The basic premise should be *how to conserve power without significantly hurting the performance of the system.*

Previous research has shown that by judiciously exploiting the various power levels and depending on the workload, it is possible to have energy-limited systems built from high performance and high-peak power components with a minimal impact on the battery life of the device. A subtle point overlooked in previous research [1,2,3] is that memory energy management has been addressed disjointly without due consideration for disk power consumption. This paper takes the view that the memory and the disk are closely coupled. A reduction in the available memory pages might potentially lead to more disk I/O. This implies that considering the totality of memory and thrashing effects would be a more accurate way of looking at it. In previous approaches, although experiments boast of saving 50%-70% of memory power, no mention is made about the thrashing effects in these situations. We feel a balance should be achieved between the memory and the disk. This balance is termed as the *operating point of memory*. Thus, our approach is to find an optimal operating point (for eg., the number of chips in the ON state) above which any further increase in the number of available pages will neither enhance performance nor reduce power consumption. The notion of power consumption in this paper includes the power dissipated by the memory subsystem as well as the power spent due to thrashing. Previous research [6] has shown that energy aware algorithms can reduce the hard disk power consumption by a significant amount with large memory sizes. The power aware algorithm explained in this paper continuously monitors the memory activity to make intelligent decisions.

In section 3 we give an overview of the BOS approach followed by related work in section 4. In section 5 we discuss the main BOS architecture. In section 6 we discuss the BOS implementation , the mechanism of power up/down of a chip which involves page migration and also some novel techinques like invisible buddy. In section 7 we discuss the various policy issues. In section 8 we discuss the studies done for resolving policy issues and the working of the OS in steady state is shown. We conclude in section 9 summarizing the paper.

3. BOS APPROACH

The software technique developed in this paper is called as the *BOS* approach which stands for *Ballooning in the OS*. BOS technique was inspired from the ballooning technique implemented in the VMware ESX server[8]. The basic BOS premise is as follows:

BOS premise: ***Dynamically resize the memory based on memory traffic and disk I/O to hover around an optimal memory size where in power consumption and system performance will be balanced.***

3.1. Disk Activity as Performance Measure

The BOS technique consists of a kernel level daemon (which we refer to as the *BOS daemon*) that is constantly in search for this elusive yet feasible operating point. The performance metric considered in this paper is the *disk activity* and one inherent feature of the BOS approach is that disk power is indirectly saved without explicitly transitioning them into low power mode. The BOS daemon dynamically increases the memory size whenever it finds a pronounced increase in disk activity thereby reducing the energy spent in unnecessary disk seeks. When memory size is reduced below a certain limit, disk I/O will start to increase as data which were previously cached would now be out of the cache and hence, more memory accesses will result in disk I/O. This means that increased disk I/O will potentially increase the response time of an application. Thus, considering the amount of disk activity as the basis of performance seems to be a reasonable approximation.

3.2. BOS details

BOS consists of a kernel level daemon that continuously monitors memory and disk activity to make policy decisions. The BOS approach uses a "two threshold" policy to make power down/power up decisions. When disk activity lies below a low threshold α, power down is triggered while if it lies above a high threshold β, chips are powered up to ease the memory pressure. On the other hand, if the disk activity lies between the two thresholds, then no action is taken as this signifies that the system is in equilibrium. This approach also prevents the potential oscillatory behavior that arises in systems due to a hysteresis loop.

The BOS daemon does not actually power down the chips but instead emulates that effect within the kernel. This is achieved through suitable mechanisms as page migration, invisible buddy explained in later sections. Using mechanisms like page migration means that powering down involves overheads and hence the thresholds selected play an important role in the overall system performance.

Experimental results show that chips with more number of allocated pages may not necessarily be the most accessed ones. Chips with very few allocated pages were accessed heavily over the entire period. This means that powering down those chips will impose a performance penalty as processes accessing those chips will be blocked until migration has been done. Though this kind of a policy for selecting chips to be powered down imposes less migration overheads, the fact that it directly affects system performance motivated us to choose an alternate approach based on chip access pattern. Results show that the threshold based policy works well for a small application that continuously dirties the pages in memory. But the big question is how to set the thresholds? The thresholds set vary depending upon the application workload and hence, having dynamically varying thresholds could solve some of the problems. Also, results indicate that the costs of migration (at a probabilistic level) depends on the number of applications currently active and an increase in the number of applications increases the migration costs. The average time required to migrate a page was found to be around 13μs.

The strength of the BOS approach lies in its ability to balance memory and disk power consumption with the help of simple thresholds. Another important point is its global approach to solving the problem rather than the per process solution adopted in previous research. A per process approach imposes nap down[1] overhead for every context switch which could vary based on the number of applications currently active, while in contrast, the BOS approach uses a fixed sampling interval which is defined as an *epoch*. Though this might take some time to enter the operating range, over long timescales, the BOS daemon would tune to the application demands. The potential drawback of the BOS approach is its delicate threshold based policy which might breakdown under different application workloads and the migration cost is another additional source of overhead.

4. RELATED WORK

People have devised hardware and software techniques for saving power in systems. [1] deals with page allocation policies that can be employed by an informed operating system to complement the hardware power management strategies. It deals with mechanisms for optimal placement of data and code on chips so that less number of memory chips is in use at any point in time. [2] is a software approach to memory energy management in which a simple BUT(Bank Usage Table) is used to keep track of accesses to memory banks on a per process basis. Its premise is that banks accessed previously are more likely to be accessed again and hence they will be switched ON while other banks can be pushed to a low power mode. [3] proposes two approaches, viz., a hardware and a OS based approach to memory energy management. The paper identifies mechanisms that will estimate the page miss ratio on the fly which can then be used to steer the page allocation for a process. This information is subsequently used for memory energy management. A power aware virtual memory layer was designed in [4] which tried to reduce the power expended by allocating all the pages of a process into minimum number of chips. The authors described a *worst-fit* algorithm to accomplish this. The advantage with this approach is that on a context switch, one needs to switch on chips that are only needed for the current process since they contain all the required pages. But the authors did not consider the case where in all the chips of a process were completely filled and if a request for a new page came, one had to dynamically switch on one of the chips in low power mode. The paper did not quantify the frequency of such failed allocation requests. The paper also used techniques like page migration to forcibly increase the locality of accesses to the chips. [5] proposed a pure hardware based approach to memory and disk power conservation with execution time as the performance metric. An observation made by [5] was that an application's performance running on a power aware system degraded by 835%. This observation was the main motivation to our BOS daemon considering not only the chip access pattern but also the corresponding increase in disk I/O. Our approach draws from [2] and [4] in that we monitor accesses to memory chips and we also use page migration to achieve our goals, but it is radically different from both in the following ways.

1. ***The most important contribution of the BOS approach is its ability to dynamically resize memory by completely powering down[2] chips rather than transitioning them into low power modes temporarily.***
2. In [2] the decisions are based on the most recent access observed but the BOS approach makes use of history over a number of epochs. This effectively eliminates the powering down decisions caused due to spikes[3] in memory accesses.
3. Both [2] and [4] operate at a per address space level and hence they optimize local performance but our approach takes a global view of memory and thus tries to optimize the overall performance of the system.

5. BOS ARCHITECTURE

The BOS architecture is shown in Figure 1. The BOS daemon is a kernel level daemon that wakes up periodically and performs some actions based on the activity in the previous epoch. The initial step of the daemon is to collect system parameters such as:

1. Collecting information about disk activity in the previous epoch.
2. Determining accessed chips in the previous epoch.

Disk I/O information is used to decide whether this is an appropriate epoch to power down/power up a chip and the latter information is used to select the victim chip, in case, a power down decision is made.

If a power down decision is made by the BOS daemon, all the pages in the victim chip are traversed and suitable processing is done for each type of page. If the page is

[1] moving to a low power mode

[2] we emulate power down due to hardware constraints

[3] random accesses by an application

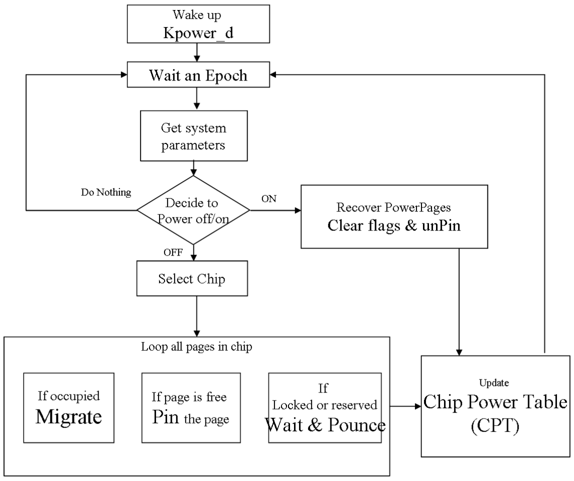

Figure 1: *BOS Architecture: Kpower_d is the BOS daemon*

an allocated page, then it is migrated to a new place. On the other hand, if it is a free page, then the invisible buddy technique is used to prevent the page from being allocated. Once all the pages in the chip have been traversed, the *chip power table* is updated to reflect the new state of the chips. The chip power table maintains a single bit for each chip indicating whether it is powered on or off.

6. BOS IMPLEMENTATION DETAILS

6.1. Power Down Mechanisms

The power down mechanisms were emulated within the kernel instead of actually powering down the chips. This meant that one had to clearly define the state of a powered down chip. The implementation was done based on the following definition for a powered down chip.

Definition: ***A chip in the OFF state is defined as one whose pages are totally invisible to the kernel. These pages must not be accessed by any process in the system which implies that they must not be allocated to any system processes by the kernel. In other words, the kernel should believe as if it had one memory chip less.***

Thus, the major challenge in the implementation involved devising mechanisms to emulate the above effect within a kernel that has the notion of a statically fixed size memory. The following mechanisms were used to power down [4] the chips.

- Page Migration
- Invisible Buddy

[4] power down => emulating power down

6.1.1. Page Migration

A chip to be powered down consists of both free pages and allocated pages. Two mechanisms were thought of to deal with allocated pages. The first method dealt with them by simply flushing the pages to disk [5] and then unmapping the pages. One potential problem with this method is that the linux kernel does a lazy writeback of pages and hence one cannot be assured that the pages were indeed forced to disk. Another issue is that, if the chip contained a "hot" page, flushing it to disk creates a performance penalty for future accesses. An alternative to the above solution that potentially overcomes both the problems is Page Migration. When an allocated page is encountered within a victim chip, the page is migrated to a new location on a different chip and all the processes that map to the current page are adjusted so that they map to the new page[6]. This method has the advantage that it reduces disk activity and additionally, any hot page in the victim chip will still be resident in the main memory but in a different location. This solution poses no problem for uniprocessor machines but it might not scale well with multiprocessor machines which have NUMA issues. Page migration is illustrated in Figure 2.

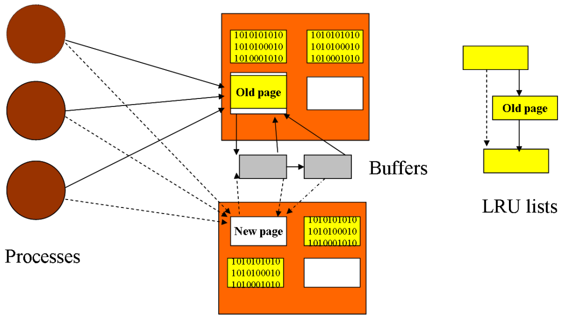

Figure 2: *Illustrating Page Migration. The dotted arrows represent the new remapped pointers.*

Page Migration Algorithm

1. Get a free page from the kernel.
2. Transfer data to the new page.
3. Remap page table entries to point to the new page.
4. Update the page cache/ swap cache as necessary.
5. Update the buffers associated with the page to point to the new page.
6. Modify the LRU lists.

[5] if they were dirty
[6] Linux 2.6 rmap facility is used to remap the pages

In order to migrate a page, a free page is required. This is accomplished by requesting a free page from the kernel. In the pathological case that the kernel offers a page on the same chip, the new page is pinned and the request is repeated. Considering the worst case scenario wherein there is only one allocated page to be migrated while all others being free pages on a chip, if the kernel continuously allocates pages from the same chip, there should be a point at which the kernel allocates pages from a different chip since all pages in the current chip would have already been allocated to the BOS daemon.

Once the page has been allocated, the data is transferred using a single *memcpy* operation. The page table entries of all processes that map to the current page are modifiied to point to the new page. Pages can be classified as:

1. File pages
2. Anonymous pages
3. Swapcache pages

File pages refer to those which contain data read from a file or those that have a backing store. Anonymous pages are formed when a user mallocs or in general, they refer to dynamically allocated memory. Swapcache pages are anonymous pages that have a valid swap space and hence a valid swap entry.

If a page is a file page or a swapcache page, then the page cache or the swapcache [7] must be updated respectively to reflect the new page location. Pages contain buffers which are referenced by their location on the disk. In linux 2.6, the buffer cache and the page cache are unified and hence, there is only a single buffer associated with a page. This buffer must be updated so that it points to the correct page which contains its data. The above steps ensure that any future access to the data will be automatically diverted to the new page and hence effectively, we have backed up the page.

6.1.2. Denying access to migrated pages

The question that remains is how to ensure that the old page is prevented access to any of the other processes in the kernel and to the kernel itself ? This question arises since it is equally possible that the swap daemon could reclaim the page which could be later reallocated to a process thereby violating the definition of a powered down chip. A page could be touched by the swap daemon if and only if it is in the LRU lists. Thus, removing a page from the LRU list makes it invisible to the swap daemon and hence the page will not be touched in the future. Unless the chip recovery algorithm is run for this chip, the pages in this chip will be inaccessible to the rest of the world.

[7] a radix tree of pages

6.1.3. Invisible Buddy Technique

The previous section detailed out how to handle the allocated pages. In this section, we come up with a new technique called as the *Invisible Buddy* to prevent free pages in the victim chip from being allocated. A new bit called as the *Power bit* has been added to every page. When the BOS daemon encounters a free page during the powering down of a chip, it sets the Power bit signifying that this page has been powered down and hence should not be allocated by the buddy allocator. When the buddy tries to allocate a page that resides on a powered down chip, a small snippet of code that imposes minimal overhead diverts the allocation to the next buddy in the list. If all pages within the current order have the Power bit set, then the allocation is diverted by incrementing the order. The buddy technique of splitting and coalescing pages makes sure that this scheme works.

6.2. Chip Recovery Mechanisms

These mechanisms deal with powering up of a chip based on a decision by the BOS daemon. Two cases have to be considered depending upon whether the page was a free page or a migrated page.

Free Pages: The Power bit has to be cleared so that future allocations can be done on this page.

Migrated Pages: Since these pages have been removed from the LRU lists, there is no way that the kernel can add them to the buddy lists. Hence, the BOS daemon forcibly adds these pages to the buddy, thereby making these pages visible. When a chip is powered up, there will be a flurry of free pages available to the kernel which can relieve it of the memory pressure.

Recovery is always done from the end of the chip. This is due to the assumption made by the buddy allocator that if a list entry of order *n* is in the free list, then all pages following it upto a total of 2^n pages will be free. The buddy allocator stores only the beginning[8] of a list of 2^n free pages and hence if we recover from the beginning of a chip, the following inconsistency may arise. The BOS daemon could clear the Power bit of a list head after which it could be descheduled immediately. In the meantime, since the list head's Power bit has been cleared, all the contiguous pages following it could have been allocated to a process even though all the Power bits in the list have not yet been cleared. This creates some form of inconsistency within the system and hence recovery was decided to be done from the last. Now, if a block of free pages is made available, then one can be assured that all 2^n contiguous pages will also have their Power bits cleared. The Power bit was set in all the pages rather than only in the list head, since any page could become a list head at any point of time due to the split and coalesce technique of the buddy allocator.

[8] referred to as the list head

6.2.1. Partial Recovery

It is not always possible to completely power down a chip. Some of the pages could be locked or under I/O. There are two ways of solving this problem.

Chip Abort: The first solution is to abort the power down process and recover the pages pinned so far.

Wait and Pounce: The BOS daemon loops on the page for a while until a timeout period to check if the page gets unlocked. If the timer expires before the page is unlocked, then the chip is recovered and the power down process is aborted.

7. BOS POLICIES

BOS must deal with two kinds of policies:

1. Determining when to power down/up chips.
2. Selecting the victim chip.

In this section, we explain in detail the mechanism used to track memory accesses and the policy used for selecting victim chips while the base policy of deciding when to power off /on will be explained in section 8. The chip to be powered down can be selected based on two different policies.

1. Chips that have the least number of allocated pages with the assumption that they will be accessed less frequently and moreover, the costs of migration will be less.
2. Chips that have been least frequently accessed in the previous epochs.

The latter policy has been implemented in the BOS daemon and is explained in section 8. To track the least frequently accessed chip, information is required about the access history of every chip in the system over a period of time. Generally, any page hit bypasses the operating system and takes place in the hardware while only page faults trap to the kernel. This kind of a design poses a problem as we will not be able to monitor the chip accesses when no page faults occur. Hence, a mechanism has to be devised that will filter these accesses to the BOS daemon. Two mechanisms were devised and they are explained below in detail.

7.1. Page Present Bit

All page table entries contain a PT_PRESENT bit which indicates whether the corresponding page is present in memory or not. The hardware checks this bit and on finding it cleared, will trap to the page fault handler. This could be exploited to trap the first access of a page to the kernel. During every epoch, the BOS daemon will clear the PT_PRESENT bit of all the page table entries and hence the first access to any page in that epoch will trap to the kernel.

7.2. Page Accessed Bit

All page table entries contain a PT_ACCESSED bit which is set by the hardware whenever the page is accessed. The BOS daemon wipes off these bits during the beginning of each epoch and examines them at the end of the epoch. Depending upon whether the bits have been set or reset, one can determine whether the page was accessed or not. Using the above mechanisms, access to chips can be easily traced out. Based on this, the BOS daemon uses a history based policy to determine the victim chip.

7.3. History Based Policy

A 32 bit history is maintained for every chip in the system. A 1 in the i th bit of chip j indicates that chip j was accessed 32 - i epochs prior to the current epoch. Thus, the 32nd bit represents the most recent epoch. During every epoch, all the bits are shifted right by one and the most recent bit gets into the most significant bit. Hence, at any point of time, the chip which contains the least numeric value in its history is the least frequently accessed one. Note that only the first access to a chip is considered rather than the number of accesses to each chip in every epoch.

8. EXPERIMENTAL RESULTS

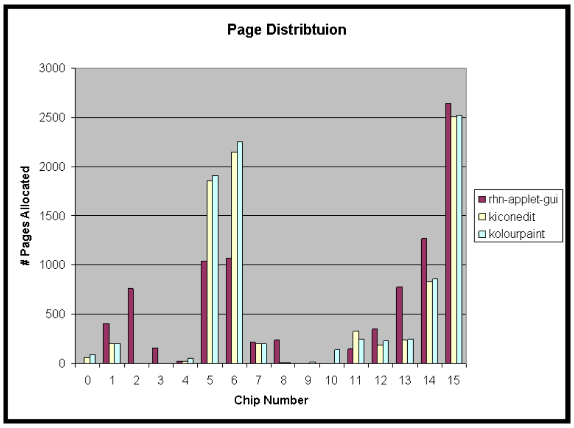

Figure 3: *Distribution of application pages across the chips.*

The experiments shown in Figures 3 and 4 were conducted on a Dell Latitude laptop[9] with 256 MB of RAM contained in 16 chips. Kernel code and data were allocated

[9] Pentium 4 - 3.2 GHz

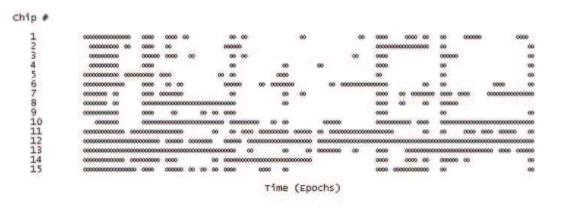

Figure 4: *Chip access pattern*

# chips OFF	File Size (MB)			
	100	200	300	400
0	8291895	15475297	23767297	53705193
10	8236489	16815762	25526496	56236439
15	8366292	17270871	27947219	60346511
20	8517397	17125774	28328536	60642874

Table 1: *Table showing the t_1 values for the various file reads in microseconds.*

on chip 0[10] and hence it was excluded from powering down by the BOS daemon. The remaining experiments were conducted on a Dell Latitude laptop with 512 MB RAM and hence a total of 32 memory chips.

As explained in section 7, BOS must follow a policy in determining the target chip to power down. Figure 3 shows the distribution of application pages to the various chips. X-windows applications were selected as the workload for the prime reason that these applications have huge memory requirements. The graph in Figure 3 was generated by exploiting the on-demand paging behavior of the OS. The page fault generated on the first access to a file or anonymous page is intercepted and accounted as explained in section 7.1. It can be seen that allocation of pages occurs in a clustered fashion with some chips being heavily allocated when compared to that of the others. This shows that pages are not allocated to the chips in a uniformly distributed manner. From this graph, one might jump to a conclusion that chips like 10 or 12 could be potential candidates for powering down as they have less number of allocated pages and moreover, the overhead due to page migration might be less, but Figure 4 gives a different picture.

Figure 4 shows the chip access pattern for the same workload in Figure 3 over a number of epochs. The graph in Figure 4 was obtained using the chip access pattern mechanism explained in section 7.3. The reverse mapping facility is used to retrieve the page table entries that map to a page and their PT_ACCESSED bits are cleared in every epoch as explained in section 7.2. An examination of the bits in the next epoch gives an overview of the chips that were accessed in the previous epoch.

The two graphs together lead to an interesting observation. Chips 10 and 12 have only a few pages allocated to them; in contrast, they have been accessed frequently over the entire time range. On the other hand, chip 15 which consists of 10 times more allocated pages than chips 10 or 12 was not accessed that frequently in commensurate with its number of allocated pages. This leads us to the important conclusion that chips with more allocated pages need not be the most heavily accessed ones. This conclusion has an important effect on the policy that we adopt for selecting the victim chips. Selecting chips that have less number of allocated pages but ones which are heavily accessed may not be a good choice as we will block the processes[11] accessing these pages for the entire time period of the power down process. This directly has an impact on the application performance as it will significantly increase their execution time. A policy based on access pattern will not have this problem as the chip being selected will be the one that has been less frequently accessed and hence probabilistically speaking, the chances of a request to a page in that chip during the power down process is very low. The cost of migration might be very high because the chip might have been completely allocated. A single page migration on a Pentium 4 3.2 GHz processor takes around 13μs which means that in the worst case scenario of migrating an entire chip, the total migration time would be around 53ms. Thus, while the allocation based policy may affect application performance, the access history based policy may have high migration overhead. The BOS approach trades off migration overhead for application performance.

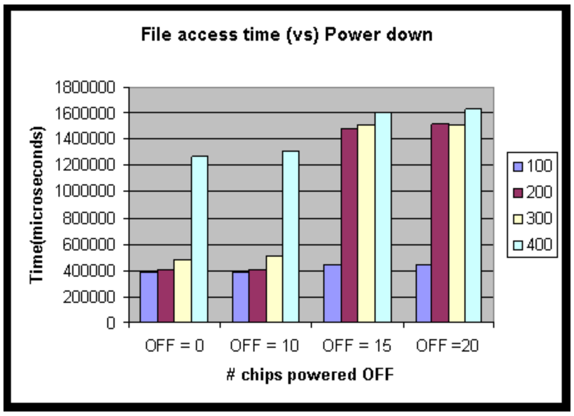

Figure 5: *File access time under varying page cache sizes*

The graph in Figure 5 shows the variation in file access time on repeated access to the same file. A file was read into the cache and the same file was accessed immediately

[10]initial 1252 pages

[11]during migration, pages are locked

without flushing the cache. This experiment was repeated for varying file sizes with different levels of cache size. The bars shown in the graph represent the time for the subsequent accesses (t_2) while the initial time values (t_1) are shown in table 1. It can be seen that t_2 values for smaller cache sizes are much higher that those for larger cache sizes since the BOS daemon had pinned most of the free pages, thereby causing eviction of the initially read pages. When we compare the t_1 values for the various experiments with their corresponding t_2 values, there seems to be a remarkable dip in the t_2 values. Thought we expect an LRU kind of a behavior from the file system, the ext2 journaling file system follows a 2Q replacement policy which has affected the results.

8.1. Impact of available memory on disk activity

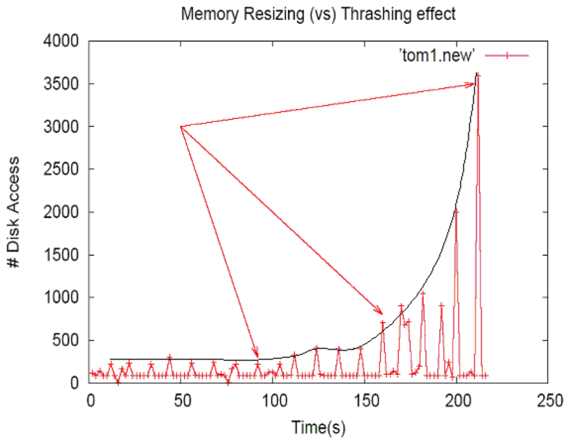

Figure 6: *Memory size vs. Thrashing effect*

In order to study the effects of memory resizing on the resultant disk I/O, an experiment was administered wherein chips were powered down by the BOS daemon at regular intervals of 30 seconds. This means that the available memory size was decreasing with time which gradually increased the memory pressure. The application used in this experiment was *tom1* which periodically dirties 56K[12] pages and sleeps for 30 seconds[13]. This application was chosen because it creates disk I/O proportional to the memory pressure. Each data point in the plot of Figure 6 was obtained every 5 seconds. This explains the relative inactivity between the spikes and the spikes indicate the actual I/O when the application wakes up. The spikes also indicate the time at which the power down takes place. As memory pressure is increased, we can see the disk I/O increasing gradually and when the available memory is decreased further beyond 13 chips, the disk I/O increases non linearly. As we join the peaks of the spikes, we can clearly distinguish 3 different operating regions. The first region has very low I/O because of low memory pressure. The I/O begins to increase in the second region where the balance between memory and disk is achieved. In the third region, it asymptotically rises to infinity.

[12] 56K pages = 14 * 4K * 4K bytes= 14 * 16 MB => 14 chips

[13] arbitrary sleep time

8.2. A Threshold Based Policy

From the above discussion, we would like to operate between the first and the second regions where the balance between available memory, required memory and disk I/O is achieved. We propose a simple two level policy to make a decision.

IF (# disk_access $<\alpha$)
    power_down()
ELSEIF (# disk_access $>\beta$)
    power_up()
ELSEIF ($\alpha <=$ # disk_access $<= \beta$)
    take_no_action()

α, β -> thresholds, $\alpha < \beta$

In this policy we make sure that the disk I/O is maintained within α and β under steady state conditions. [14].

8.2.1. *Dynamic response*

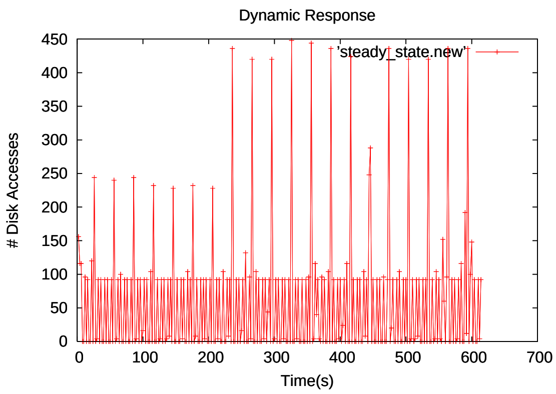

Figure 7: *I/O response of the threshold policy*

The dynamic disk I/O response of the system running the same application is shown Figure 7. In this policy α = 250 and β = 500 [15]. The system settles between α and β as expected.

[14] Steady state condition is defined as the state in which the application does not increase or decrease its memory requirements by itself and no new application is started

[15] α and β were manually set

8.2.2. Drawbacks of the threshold based policy

The thresholds were static and were manually selected and would not respond properly for an application with varying memory requirements. Also, this policy does not differentiate between disk I/O caused by an application writing to disk and I/O caused due to thrashing. But considering portable devices which mainly run only one application or few at best, a static policy would be good enough to achieve the desired effect. For high end servers a dynamic policy should do better.

8.3. How to verify power down?

Since we are not actually powering down the chips, we need to verify that our mechanism for power down indeed prevents access to those pages. Every physical page in the system consists of a *mapping* field. This field is set to NULL whenever the page is not used by any application or the kernel. The BOS daemon also sets this field to NULL once it migrates a page. An experiment was conducted in which the X-server and applications like mozilla, open office and kiconedit were run. 10 chips were powered down by the BOS daemon and the pages in those chips were monitored periodically by checking the *mapping* field. The *mapping* field was found to be NULL for all the pages in the powered down chips which indicates that these pages have not been allocated to any of the system processes. This effectively proves that the power down was emulated perfectly within the kernel.

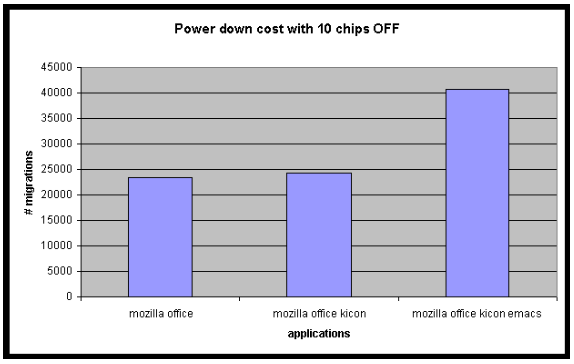

Figure 8: *Power down costs vs. # active pages when 10 chips are powered down*

8.4. Quantifying the cost of power down:

To quantify the cost of power down, the number of migrations was profiled when 10 chips and 15 chips were powered off. The applications were started immediately after the system booted and the daemon was started after that. This ensured that the daemon got a chance to migrate the pages

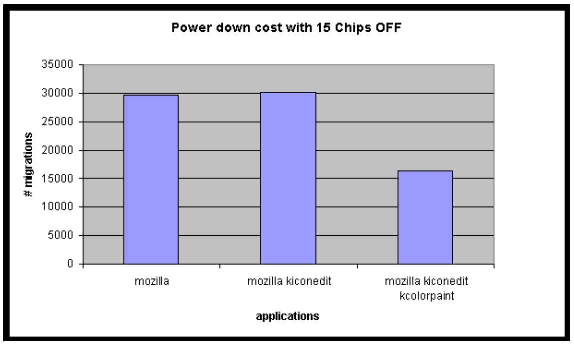

Figure 9: *Power down costs vs. # active pages when 15 chips are powered down*

of the applications that were under consideration. From Figures 8 and 9, it can be seen that as more applications are run, the number of migrations increases, which clearly shows that migration could become a major bottleneck of the BOS architecture. The final bar in Figure 9 is a lucky case because the daemon chose chips that had pages other than the ones which contained the pages of the workload. In general, we can infer that as the number of active pages increases, the average number of migrations increase proportionately.

9. CONCLUSION

This paper addressed the problem of memory energy management in computer systems. A totally different approach to power conservation was proposed by viewing the entire memory instead of a per process abstraction. This approach which we call as the *BOS*, tries to optimize the global performance of the system with little or no overhead. The BOS approach makes use of page migration and invisible buddy techniques to emulate powering down of chips within the kernel. Different policies were proposed to select the victim chips and the one based on chip access pattern was implemented trading off migration for application performance. An empirical study was conducted to study the relationship between available memory size and disk accesses. Based on the study, a simple threshold based policy was proposed with thresholds tuned to a particular application.

The BOS approach depends on thresholds to fare well. The thresholds used in this paper are appropriate for the given workload and they seem to perform well by settling in the operating region. But the thresholds might vary with application depending on its memory consumption behavior. Since the BOS approach will possibly be used in handheld devices and iPods that potentially have one or few applications active most of the time, manually setting thresholds should not be a problem. In any case, one could think of dynamically varying thresholds that adapt themselves to the current workload. This could be more appropriate for high end servers which have varying levels of demand. Page migration is one of the costs associated with the BOS approach, but migration helps us in retaining the pages in main memory for a longer period of time thereby reducing the number of page faults. Future work on this project may involve devising a more dynamic policy for making power down/up decisions, implementing variable epoch intervals and actually implementing power down/up.

10. ACKNOWLEDGEMENTS

We thank Prof. Remzi H. Arpaci-Dusseau for his excellent guidance throughout the project. We would also like to thank Muthian Sivathanu, Vijayan Prabhakaran and Lakshmi Bairavasundaram for their invaluable thoughts and discussions on this project. We also thank Amit Jhawar for helping us with the resources.

11. REFERENCES

[1] Alvin R.Lebeck, Xiaobo Fan, Heng Zeng and Carla Ellis, *Power Aware Page Allocation*, ASPLOS 2000.

[2] V. Delaluz, A. Sivasubramaniam, M. Kandemir, N. Vijaykrishnan and M.J. Irwin, *Scheduler-based DRAM Energy Management*, 39th Design Automation Conference, June, 2002.

[3] Pin Zhou, Vivek Pandey, Jagadeesan Sundaresan, Anand Raghuraman, Yuanyuan Zhou and Sanjeev Kumar, *Dynamic Tracking of Page Miss Ratio Curve for Memory Management*, ASPLOS 2004.

[4] Hai Huang, Padmanabhan Pillai and Kang G. Shin, *Design and Implementation of Power-Aware Virtual Memory*, USENIX 2003.

[5] Xiaodong Li, Zhenmin Li, Francis David, Pin Zhou, Yuanyuan Zhou, Sarita Adve and Sanjeev Kumar, *Performance Directed Energy Management for Main Memory and Disks*, ASPLOS 2004.

[6] Athanasios E. Papathanasiou and Michael L. Scott, *Energy Efficiency through Burstiness*, IEEE Workshop on Mobile Computing Systems and Applications 2003.

[7] John Zedlewski1, Sumeet Sobti1, Nitin Garg, Fengzhou Zheng, Arvind Krishnamurthy and Randolph Wang, *Modeling Hard-Disk Power consumption*, FAST 2003.

[8] Carl A. Waldspurger, *Memory Resource Management in VMware ESX Server*, OSDI '02, 2002.