An Allocation Framework for Optimizing Memory Power Consumption and Controlling Fragmentation

A Thesis

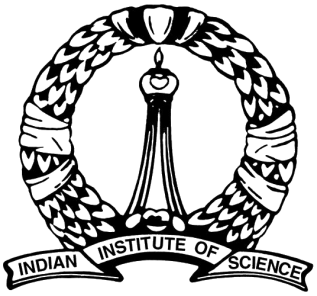
Submitted for the Degree of

Master of Science (Engineering)

in the **Faculty of Engineering**

by

Ashish Panwar



Dept. of Computer Science and Automation
Indian Institute of Science
Bangalore – 560 012 (INDIA)

JULY 2015

DEDICATED TO

My family and my teachers

Signature of the Author: .....

Ashish Panwar
Dept. of Computer Science and Automation
Indian Institute of Science, Bangalore

Signature of the Thesis Supervisor: .....

K. Gopinath
Professor
Dept. of Computer Science and Automation
Indian Institute of Science, Bangalore

Acknowledgements

I would like to start acknowledgement by thanking Indian institute of science (IISc) for providing wonderful environment for research. I also thank Ministry of Human Resource and Development (MHRD), Government of India, for their monetary support.

I am deeply grateful to Prof. K. Gopinath for providing me an opportunity to work at the Computer Architecture and Systems Laboratory. I am thankful for his supervision and guidance.

I am thankful to the department of Computer Science and Automation (CSA), IISc for providing excellent facilities to help me conduct my research. I feel proud to be part of the Computer Architecture and Systems Laboratory, CSA, IISc. I thank all my labmates and friends for their constant help, support and making my stay in the deparment memorable.

I thank all staff members who helped me in academic related issues and provided a comfortable environment to work. Lastly, but most importantly, I thank my family for their enormous support.

Abstract

Large physical memory modules are necessary to meet performance demands of today's applications but can be a major bottleneck in terms of power consumption during idle periods or when systems are running with workloads which do not stress all the plugged memory resources. Contribution of physical memory in overall system power consumption becomes even more significant when CPU cores run on low power modes during idle periods with hardware support like Dynamic Voltage Frequency Scaling.

Our experiments show that even 10% of memory usage can make references to all the banks of physical memory on a long running system. Operating systems can play a decisive role in effectively utilizing the power management support of modern DIMMs like PASR(Partial Array Self Refresh) in these situations but have not been using them so far. Our experiments also reveal that memory hot-remove or memory migration for large blocks is often restricted, in a long running system, due to allocation policies of current Linux VM. Hence it is crucial to improve page migration for large contiguous blocks for a practical realization of power management support provided by the hardware.

We propose three different approaches for optimizing memory power consumption by inducing bank boundary awareness in the standard buddy allocator of Linux kernel as well as distinguishing user and kernel memory allocations at the same time to improve the removability of memory sections (and hence memory-hotplug) by page migration techniques. Through a set of minimal changes in the standard buddy system of Linux VM, we have been able to reduce the number of active memory banks significantly (upto 80%) as well as to improve memory-hotplug support (upto 85%).

Contents

List of Figures

List of Tables

Chapter 1

INTRODUCTION

The size of physical memory modules has increased significantly in recent times. Increased size and frequency of memory have enabled users to run heavy duty applications across platforms, ranging from small mobile devices to large scale server systems. Power consumption of computing systems is an important factor for end-users these days and is becoming more difficult with increased performance demands. CPU has been a focus of many power optimizing techniques like Dynamic Voltage Frequency Scaling [Sueur 2010] and current systems do well in the context of CPU power.

Memory has also drawn attention in recent times because they contribute to a significant portion of power consumption in servers (25%-40%) [Jantz 2013] [Intel 2008] as well as in mobile systems [Brandt]. Table 1.1 shows the power consumption of different memory size and configurations in a server system. Google's upcoming "Doze" power management project [Google IO 2015] for android platform also addresses the problem of memory power consumption. However, the main focus of power optimization till date has been on hardware resulting in memory modules supporting a wide range of power management features. Latest low power DDR3 SDRAMs support more than 10 different power states [Micron 2005], management of which is handled by memory controllers. We will be discussing only a subset of these power modes as not all of them are of interest to operating systems or system developers.

Physical memory modules are organized in multiple banks each of which is a contiguous array of addresses. Partial Array Self Refresh (PASR) enables each memory bank (or set of banks) to switch itself to self-refresh mode which reduces power consumption with data retention. Deep-power-down provides maximum power savings but results in data loss due to disabled refreshing of the capacitors which leak charge over time. Power savings come from the reduced current supply in low power states. A memory bank in deep power down mode draws as low as 10μA current compared to 0.3mA in self refresh which can be as high as 25mA-

Frequency (MHz)	DIMM (Configuration)	DIMM (Tech/Capacity)	Power/DIMM	64GB System Power
1066	QR x 4	2Gb/8GB	15.5 W	124 W
1033	QR x 8	2Gb/8GB	10.6 W	85.8 W
1333	QR x 4	1Gb/4GB	10.6 W	169.6 W
1333	QR x 4	2Gb/16GB	20.5W	82 W
1600	QR x 8	2Gb/8GB	10.1 W	80.8 W
1600	QR x 4	2Gb/8GB	19.1 W	152.8 W

Table 1.1: DDR3 DIMM Power Consumption by Frequency, Configuration, and Capacity.

80mA in higher power modes in Micron mobile DRAMs [Micron 2005]. Table 1.2 shows relative power consumption of different memory power states. Brandt et al. [Brandt] proposes several implementation methodologies for reducing DRAM power consumption. In this work, we will be dealing with Bank-Selective PASR where each memory bank can transit its state from one power mode to another independently of other banks.

For better utilization of low power modes, operating systems' memory management policies should be coupled with memory controller. It can be achieved by consolidating memory allocations to a subset of memory banks. Page migration can also be utilised to limit the number of memory banks used by the applications in long run. If memory allocations are restricted to a subset of available banks, rest of the banks can be put to deep-power-down mode without worrying about data loss. However, power savings is a secondary goal of any memory manager and sacrificing its performance should not be considered as an option.

In this thesis, we present three different solutions for implementing a memory power optimization framework in Linux. We modify some components of Linux VM subsystem to induce memory bank awareness in the standard buddy allocator. Memory bank boundary awareness in the page allocation path helps in significantly reducing the number of in-use memory banks to reduce the overall memory power consumption. In a long running system, when memory references spread over a large portion of physical address space, we use the page migration technique from the existing memory-hotplug infrastructure of Linux to limit the number of memory banks referenced by applications.

However, the hot-remove part of the memory-hotplug infrastructure is not yet very well supported by Linux. Since it depends on migrating a large contiguous chunk of physical memory, it is often restricted by the presence of unmovable pages. External memory fragmentation is another related issue which is also caused by the mixing of movable and unmovable physical pages. Using page migration on memory bank granularity makes it even more critical to solve the external fragmentation problem for effectively utizing page migration to optimize memory power consumption. Our solution to the memory fragmentation problem has a direct and

Power State	**Relative Power Consumption**
Read/Write	100%
Active-Idle	63%
Self-Refresh	7%
Deep-Power-Down	<1%

Table 1.2: Relative power consumption of mobile DRAM power states

significant impact on hot-remove capability of memory-hotplug.

For the rest of the thesis, we use memory-hotplug to refer to memory hot-remove for simplicity and discuss it before memory power in most cases as the proposed memory power optimization frameworks rely heavily on memory hot-remove functionality.

Linux is different from other operating systems like Windows and Mac OS since it uses the same code base for different computing systems, from tiny embedded devices to large scale server systems. It is because of this fact that any practical solution in Linux should be scalable from small to large systems. For this reason, one of the main objective of this thsis is to provide a flexibile and compatible solution which can work across a variety of systems.

The primary contributions of this thesis are -

- Analysis of how/why the current memory management policies of Linux Buddy allocator obstructs power management and memory-hotplug.
- Design and analysis of Adaptive-Buddy (and a few simple techniques) for improving memory-hotplug and power management framework.
- Performance analysis of Adaptive-Buddy allocator with a set of application and micro-benchmarks.

Memory interleaving improve system performance significantly but causes more power consumption on memory modules at the same time. It has been constantly ignored over the years by developers trying to solve the problem of memory power consumption. To our knowledge, it is the first study to put together and analyze memory power management and memory interleaving in the same context. We provide a prototype implementation for this by integrating it with Adaptive-Buddy.

The rest of this thesis is organised as follows : Chapter 2 provides overview of linux memory manager, buddy allocator and issue related to memory power and memory-hotplug. Chapter 3 discusses related work and the motivation behind our proposed solution. Chapter 4 provides the design and implementation of our framework. Results related to the context of memory

power consumption and memory-hotplug are discussed in Chapter 5 and Chapter 6 provides a detailed evaluation of our proposed solution of keeping user and kernel memory isolated on physical memory modules. Finally, we conclude in Chapter 7.

Chapter 2

BACKGROUND

Current SMP systems typically provide processors with local memory for improving system performance in NUMA architecture. Access latency of memory on a remote node could be significantly higher based on the distance between memory and requesting CPU. Commercial applications can have performance deterioration of upto 20% because of remote memory accesses [FUJITSU 2011]. To optimize performance, operating systems memory management policies are generally NUMA aware and try to allocate memory from local node of the requesting process.

2.1 Overview of Linux Memory Manager

In linux, each node memory is divided in different groups called as memory zones. Figure 2.1 represents the standard linux kernel zone layout on x86 and x86_64 architectures. Zone DMA represents first 16MB of physical memory and is used for old ISA devices which can only address memory upto 16MB. Zone NORMAL represents the portion of memory which can be directly mapped in the address space of kernel. On 32 bit machines, 4GB address space is typically divided in the ratio of 3:1 between user and kernel space respectively. 128MB in kernel address space is reserved for mapping memory beyond 1GB, hence zone NORMAL represents memory upto 896MB and rest of the memory (if available) is managed by zone HIGHMEM.

On 64-bit architectures, zone HIGHMEM is not required as all the memory can be directly mapped in the kernel address space and DMA32 represents upto 4GB of memory which can be addressed by modern DMA devices.

Linux uses buddy allocator for managing page allocation and page freeing operations. 4KB is the most commonly used page size across different architectures and operating systems[1].

[1]Larger than 4KB page size or multiple page sizes are also common in practice but for simplicity only 4KB page size is considered in this thesis.

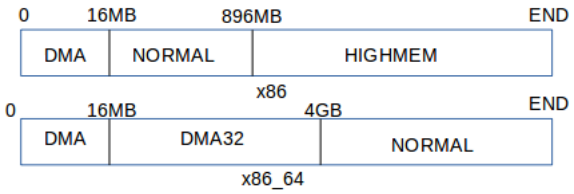


Figure 2.1: Standard Linux Kernel Zone Layout

It maintains MAX_ORDER (currently defined as 11) doubly linked lists of free blocks in a *free_area* structure each of which is dedicated to represent power-of-two, ranging from 2^0 to $2^{MAX\_ORDER-1}$, number of physically contiguous pages in a single list entry. While allocating a page, buddy allocator searches a desired list and allocate the first page from it. A higher order block can be splitted into multiple smaller blocks to satisfy a memory allocation request if the requested list is empty. While freeing a page, buddy allocator first checks if the neighbouring block (also called as buddy) of the same size is free. The blocks are merged into a larger block if the buddy is found to be free before inserting into the free list. The process is repeated until either the buddy is not free or the free blocks have already been merged to form the largest free block.

Despite having internal and external fragmentation issues, buddy allocator is preferred for its speed [Gorman 2005]. It can allocate even large chunk of contiguous pages in a single lookup on *free_area* structure as long as free memory is available. Internal fragmentation is a situation where more than required memory is allocated to objects and external fragmentation is where sufficient memory is available for allocation but is split in multiple blocks. Internal fragmentation is handled carefully by slab allocator in Linux which is used to allocate small objects. External fragmentation is somewhat mitigated by eliminating the need for large contiguous chunks of physical memory from different Linux subsystems and by providing virtually contiguous memory to process.

One important factor for external fragmentation is the presence of non-movable pages, pages that are pinned in memory and can not be moved to another physical location using page migration. Generally these pages are owned by kernel and contain critical data like process page tables, which is not considered safe to move. In general, all user application data is movable but their management structures owned by kernel are not. When system run out of large contiguous memory blocks and memory allocations start failing, zone compaction is performed to cluster free pages together by migrating some pages from one location to another. However, page migration is often restricted when non-movable pages scatter throughout entire address space and there are very few (or none) large contiguous blocks containing only movable pages.

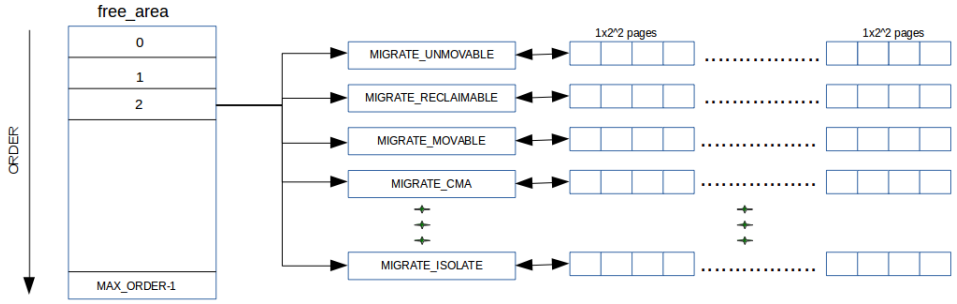


Figure 2.2: Layout of *free_area* structure. Anti-Fragmentation lists shown correspond to an O(2) block and hence contain 4 pages in a single block.

2.2 Linux Approach to External Fragmentation

To deal with the problem of non-movable pages, each order's free list is divided in different sublists as part of anti-fragmentation [M. Gorman 2005] which is a technique that tries to stop fragmentation from happenning in the first place. These sublists are managed by *free_list* structure which is embedded inside *free_area*. Figure 2.1 shows layout of *free_area* structure including relevant sublists for O(2) free list. MIGRATE_UNMOVABLE and MIGRATE_RECLAIMABLE typically represents memory used by kernel with one major distinction that UNMOVABLE pages can not be easily moved or reclaimed while RECLAIMABLE are pages which kernel can directly reclaim (such as those used by inode caches).

MIGRATE_MOVABLE and MIGRATE_CMA are generally associated with user applications which contain anonymous or page cache pages wchich can be migrated any time. One difference between CMA and MOVABLE migrate types is that migrate type of a pageblock containing MIGRATE_CMA pages can not be changed. Pages belonging to MIGRATE_ISOLATE free lists are not used for allocation which serves the purpose of temporarily isolating pages from the allocation path and is often required in cases like NUMA migration.

When an allocation request comes, kernel knows its migrate type based on the source of allocation, and allocates pages from the respective list. It can also steal pages (named as fallback in kernel source) from another list if no pages are present in the respective lists of same or higher order. The trick is to steal the highest order available page block (preferably 4MB) from the closest migrate type. Stolen pages from a higher order list are used to satisfy subsequent requests of same type which helps in clustering same migrate type pages together. Once non-movable pages are clustered in large blocks, compaction can provide large free pages by migrating pages from movable memory blocks.

Another solution available in Linux for controlling fragmentation is the presence of optional

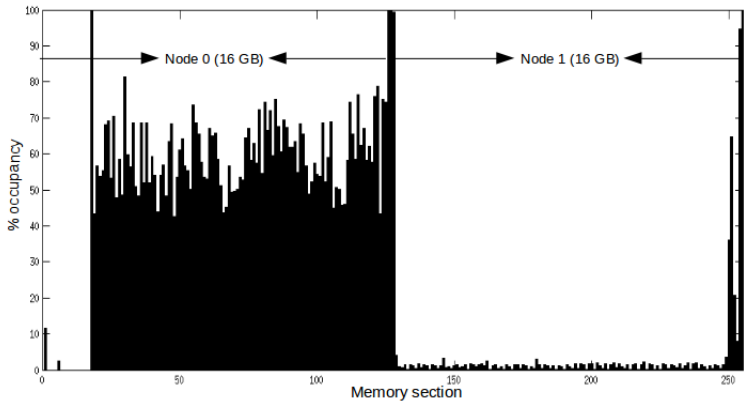


Figure 2.3: Impact of default allocation policies of linux

ZONE_MOVABLE [Gorman 2007]. It creates a zone that is usable only for movable allocations (anon/page-cache pages). Size of this zone is determined by the *movablecore* (or *kernelcore*) parameter specified at boot-time. Similar to ZONE_MOVABLE, we also consider everything to be unmovable except MIGRATE_MOVABLE and MIGRATE_CMA pages. Later in the thesis, we will discuss some issues related with ZONE_MOVABLE.

2.3 Memory-Hotplug

Memory-Hotplug allows users to increase/decrease the amount of memory. Generally, there are two purposes.

1. For changing the amount of memory. This is to allow a feature like capacity on demand.
2. For installing/removing DIMMs or NUMA-nodes. This is to exchange DIMMs/NUMA-nodes, reduce power consumption, etc.

One is required by highly virtualized environments while the other is required by hardware which supports memory power management. It operates on memory section granularity where each section is of 128MB physically contiguous memory. Removal of DIMMs (also referred to as memory hot-remove) works by offlining a memory section using page migration and making it invisible to the kernel i.e., hiding its pages from allocation path by putting them to MIGRATE_ISOLATE free list. In this work, we refer to logical memory hot-remove as memory-hotplug for simplicity.

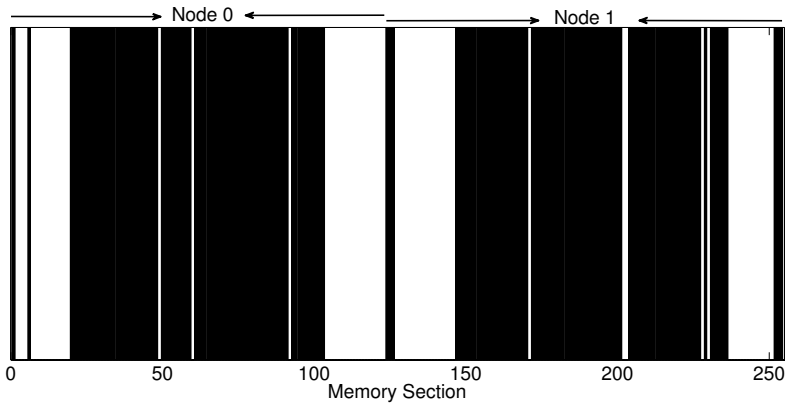


Figure 2.4: A snapshot of kernel memory footprint with default buddy allocator. Black sections represent memory sections containing kernel pages i.e., unremovable memory sections

2.4 Issues related to Memory Power Consumption and Memory-Hotplug

The bottleneck with respect to memory power is paging operations happening at the head of the free lists which is good for performance but prevents any opportunity of power savings by creating fragmentation. As processes allocate and free memory, lists get randomized, implying memory allocations spreading references over all memory banks. Figure 2.2 shows a situation where around 90% memory is free on node 1 but none of 128 MB memory sections is completely free.

Operating systems can help in preventing these situations by carefully guiding memory allocations to a subset of memory banks and hence facilitating memory harware to put other banks in low power modes. It may not be possible to keep memory allocations confined to a subset of memory in a long running system, specially when stressful workloads are running. But when idle periods follow an active period, page migration can be used to free some memory banks to save memory power consumption. Memory-Hotplug also relies on removability of memory sections using page migration.

However, page migration to free a large contiguous portion of physical memory is often restricted due to the presence of unmovable page. A single non-movable page is all that is required to make an entire memory section (or bank) unmovable. The anti-fragmentation approach of clustering same migrate type pages together work well as long as memory operations are performed on small blocks of pages (upto 4MB). However, from memory power management or memory-hotplug point of view, it fails because memory migration needs to be done on large contiguous blocks. Figure 2.4 depicts a situation where more than 170 out of 256 memory

sections contain unmovable memory blocks and hence can not freed. This snapshot was taken when more than 4GB memory was still free and had not been touched since system startup. Over a period of time, it is not difficult to come across a situation where all memroy sections contain some kernel pages.

Chapter 3

RELATED WORK

Several power management schemes have been developed over the years which can be broadly classified in two categories. Most of these solutions try to optimize memory power consumption per process by utilizing hardware facilitated low power modes which provide data retention. It is generally achieved by consolidating memory references (especially working set) of appliations to a subset of memor banks. On the other hand, some techniques try to control the amount of memory allocated to all processes and focus on facilitating deep power down mode. If memory is not being allocated from a memory bank, it can be transitioned into deep-power-down mode without worrying about any data loss. Our solution fall into the later category where we try to minimize the number of memory banks allocated and leave the management of low power data retention modes to hardware.

Jantz et. al. [Jantz 2013], in a recent work, facilitates collaboration between applications and host system by introducing a new abstraction "trays" inside each memory zone to organize separate free lists for each power-manageable unit. The solution, however, ignores the size of the zone and node structures which could grow very large for large memory systems. It also requires a linear search over the set of trays for page allocation which also introduces significant overhead in the page allocation path in a busy system.

Some techniques for incorporating power optimization decisions with context swithcing have also been discussed. Delaluz et al. [Delaluz 2002] keeps track of memory banks used for each application and selectively turn them on or off with each context switch to manage energy consumption of DRAM per application. Huang et. al. [Huang 2003] maintains set of power-manageable units for each task and tries to minimize the size of the union of sets of all tasks by using page migration and using separate units for heavily shared library pages. It also synchronizes the transitioning between different power modes with context switch to alleviate the resynchronization delay associated with mode transition. Lebeck et. al. [Lebeck 2000]

proposes several techniques for power aware page allocation to manage power states of each memory chip.

Koala [Snowdon 2009] lets the OS manage power according to an overall policy guided by predicted power and performance. It collects per-process performance statistics that characterize an applications' behavior and uses it at each scheduling event to determine the most appropriate operating condition. Hual et. al. [Huang 2005] introduces the concept of hot and cold ranks (for better utilization of low power modes), which are created dynamically by keeping frequently accessed pages on same ranks using page migration. It relies on a modified memory controller to accurately track the memory references. Actual memory references are typically transparent to operating systems as page table walk is done by the processor on most architectures. Some techniques exist to determine referenced memory regions like [Garg 2011] but require traversal over process page tables which causes as much as four memory references to check the access bit of a single page on x86 architectures and using them for power management purpose seem to be very costly.

A. Garg [A. Garg 2011] proposed memory-region based approach incorporating power manageable region information in buddy allocator but duplicates the entire zone structure for each memory region. S. Bhat, in his recent revision [S. Bhat 2013] of the approach followed by A. Garg, eliminates the need of duplicating the zone structure by capturing power-manageable hardware units in data structure parallel to memory zones. Both these approaches, however rely on an O(log n) sorting logic in page free path to maintain the order of free lists and also on migration from lower to higher memory regions independent of the workloads. Migrating from one end of memory bank array, irrespective of workloads running on the system, causes unnecessary migration overhead in some situations.

One major problem with the techniques discussed above is their inflexibility to address diverse execution environments and implementation issues and complexities of a real world. Per process memory power management techniques have been useful in single threaded environments but it is not clear how they scale with todays highly concurrent systems. Memory power management is also highly dependent on executing workloads and comes into play only when there are sufficient idle periods to utilize low power modes. Hence, any solution should be able to adapt itself to changing workloads to avoid any unnecessary overhead in a running system. Though several of these techniques rely on page migration, the problem of memory fragmentation which restricts migration of large contiguous chunks has not been discussed. It is vital to solve the problem of memory fragmentation for page migration to work on large memory blocks. Adaptive-Buddy system presented in this thesis is an attempt to address these issues.

Chapter 4

IMPLEMENTATION

Implementation of a memory bank aware buddy allocator requires knowledge of actual bank boundaries. This information can be acquired via device tree on some platforms (like ARM and Power) but is not available on x86 based architectures yet. ACPI 5.0 [ACPI 5.0] mandates the use of MPST (memory power state table) which provides information of all independently power manageable memory nodes. Due to the lack of actual bank boundary information, we consider each each memory bank to be 256MB based on the memory module size and JEDEC standards [JEDEC 2012]. However, for experimental purpose, this is not a problem because the actual transitioning between memory power modes is handled by memory controller. If the address mapping of memory banks can be fixed, rest of the subsystem do not require further changes.

In this chapter, we provide three different solutions that have been developed towards solving the poblem of memory power management. We also discuss some practical issues related to each of them.

4.1 A Naive Approach

A simple solution can be employed by maining free pages on memory banks rather than managing them per memory zone. It can be done by initilizing *free_area* structure on each memory bank. Set of memory banks can be managed inside or outside memory zones. The next two approaches discussed in this section maintain a *free_area* structure per memory bank which are managed inside memory zones. For this pupose, we introduce a new data structure i.e., *mm_bank* each one of which has an instance of *free_area* associated with it. The number of memory banks inside a zone depends on the number of pages spanned by the zone.

4.1.1 Array based Bank-Buddy Allocator

In this approach, we consider memory spanned by a zone as an array of memory banks. Once free lists are initialized on different banks, the allocator can choose to allocate pages from any one direction i.e., higher bank number to lower bank or vice-versa. This makes sure that a completely free bank is not used for a new allocation request as long as already used banks have the capability to satisfy that particular request.

To control mixing of movable and unmovable pages, it is important to keep kernel pages restricted to a minimum number of memory banks. Arranging free pages in a set of banks makes it feasible by following different directions for kernel and user memory allocations. In our implementation, we allocate kernel memory from lower to higher banks and user memory from higher to lower banks as shown in figure 4.1.

Over a period of time, all memory banks will be used to satisfy memory requirements and memory references will again spread over the entire physical address space. When applications exit or start releasing pages, pages can be migrated from low memory banks of user portion to higher banks to utilize deep power down mode without any data loss.

There are two main issues related to this approach -

1. **Overhead in Page Allocation Path:** For each memory request, allocator starts scanning the array of memory banks from one end of bank array. In the worst case, when a large portion of memory has been allocated, the allocator end up scanning entire bank array resulting in an O(n) allocation cost, where n is the number of memory banks in the zone.

2. **Memory Migration Overhead:** Software driven strategies usually rely on page migration in long running systems to stop unnecessary references to memory banks. Keeping kernel memory towards the low end of the array eases the task of migrating large groups of pages owned by user applications. But if pages are always moved from the lower banks of user space to higher banks, it can result in significant migration overhead as the number

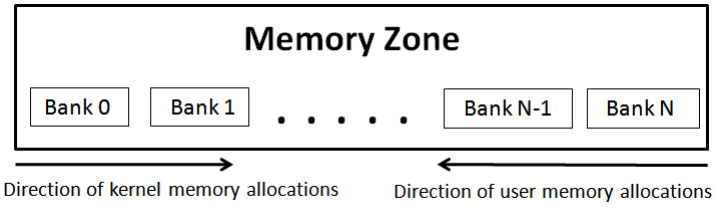


Figure 4.1: Array Based Bank-Buddy Allocator

of free pages in a bank will depend on the order in which pages are being freed. It can easily happen that a higher memory bank has significantly more free pages than a lower one.

There is no impact on the performance of page freeing path as given a page, we directly map it to its memory bank using the page frame number resulting in an O(1) operation. Cost of freeing a page remains the same for the subsequent solutions as well.

4.1.2 List based Bank-Buddy Allocator

To reduce migration overhead associated with array based approach, memory banks can be treated as a list of banks. List based organization provides the flexibility to treat each bank as an independent unit of memory, irrespective of its position inside memory zone. Fragmentation can be controlled by using separate lists for kernel and user allocations i.e., LIST_KERNEL and LIST_USER. Free memory banks also reside on a list named as LIST_FREE. Figure 4.2 depicts the list based organization of memory banks.

LIST_KERNEL and LIST_USER are empty lists at system startup and all memory banks belong to LIST_FREE. Memory requests are forwarded to respective lists based on their migrate types. When a memory request fails from its list, a bank from LIST_FREE is added to the head of that list and used for subsequent allocations.

Under high memory pressure, when LIST_FREE is empty, if an allocation request fails for a migrate type, free pages from other list are used to satisfy memory request to make sure that allocations do not fail as long as free memory is available. If a movable type memory request can not be satisfied from LIST_USER, allocation falls back to LIST_KERNEL and behaves exactly like the default fallback routine of standard buddy system. However, if a non-movable memory page is to be allocated from LIST_USER, we allocate it from a memory bank which has the highest number of free pages and add this bank to LIST_KERNEL as it becomes non-removable now. Selecting a memory bank carefully from LIST_USER for a non-movable memory request is very important because in some cases we observe that an arbitrary selection behaves worse than the standard buddy allocator with respect to memory-hotplug.

While list based approach helps in reducing overhead of memory migration by carefully selecting memory banks which are least occupied as migration candidate, it does not help with the worst case execution time of memory allocation path. A single allocation request may still result in O(n) operation under high memory pressure as it requires traversing the list of banks. However, average runtime can be improved by following two techniques-

- **Cache:** It involves caching the memory bank which resulted in the last successful allo-

cation and using it for further allocations before falling over the list of memory banks.

- **Optimal Selection:** It involves a kernel thread, whose task is to select a memory bank which has the maximum probability of satisfying a memory request (bank with maximum free pages) and putting it at the head of bank list after some interval.

Both the above optimizations work well under normal execution environments but fail when memory utilization reaches high. Caching a memory bank is not always fruitful especially when all the memory banks have very little memory left in their free lists. Problem with optimal selection lies in determining how frequently the optimal candidate should be placed at the head of bank list. If selection is invoked very frequently, it may result in lock contention over zone structure which may not be acceptable in some situations because zone lock is already one of the most highly contented locks in Linux kernel while putting the thread to sleep for a long time can result in the default behavior i.e., O(n) for a single allocation.

It can happen that a single memory bank is spanned by two zones. However, it can happen only on the first node in a system because all the other CPUs populate only the highest memory zone. Note that on the first CPU also, ZONE_DMA has to stay active at all times because DMA devices will get their memory from this zone only. This leaves us with one possible scenario where a bank may be covered by two zones i.e., by ZONE_NORMAL and ZONE_HIGHMEM on 32 bit architecture and between ZONE_DMA32 and ZONE_NORMAL on 64 bit architectures. While trying to free such a bank, page migration has to be successful on both the zones. In the array based approach it is easy to see that in ZONE_HIGHMEM (or ZONE_NORMAL on 64-bit systems), the overlapped bank will be used for non-movable pages and can not be freed by migration. Rather than complicating the VM subsystem even more to handle this particular situation, we do not consider this bank for page migration and always use it for allocation in both the zones. We follow the same approach in the Adaptive-Buddy allocator also which is discussed in the next section.

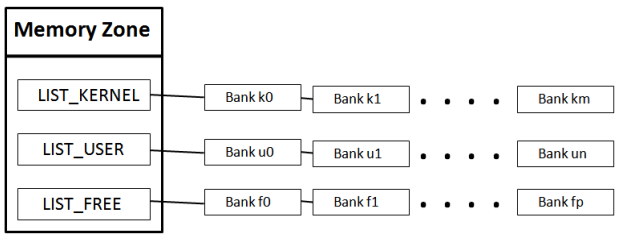


Figure 4.2: List Based Bank-Buddy Allocator

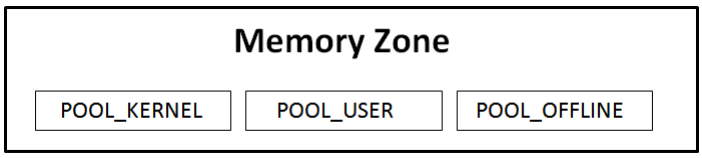


Figure 4.3: Adaptive-Buddy Allocator

4.1.3 Issues with Multiple Buddy Allocators per zone

Apart from the worst case allocation time being O(n) in the above two solutions, there is one more issue with initializing *free_area* on each memory bank and keeping it inside memory zone i.e., size of zone and node structures. Size of *free_area* structure when combined over 0 to MAX_ORDER-1 blocks amounts to 1144 bytes on x86_64 while the size of entire zone structure is 1920 bytes. Initializing *free_area* on each memory bank increases the size of zone structure by a factor equivalent to number of memory banks spanned by the zone. Impact on the size of node structure is even worse as a node spans multiple zones. Increased size of these structures is a critical issue because these are some of the most highly contended resources in Linux which means that they can occupy a large portion of system cache when accessed and affect performance of user applications.

4.2 Adaptive-Buddy Allocator

Keeping user and kernel memory on separate memory banks eases the task of memory migration. All memory banks belonging to the same allocation type can now be merged in a single structure i.e., memory pool. It reduces the amount of memory occupied by buddy management structures as there are only three pools of memory now i.e., POOL_KERNEL, POOL_USER, POOL_OFFLINE each with its own lists of free pages as shown in figure 4.3. In cases where an allocation request can not be satisfied from its dedicated pool, pages can be taken from POOL_OFFLINE or stolen from other pool. If a page is allocated from POOL_OFFLINE, rest of the pages belonging to that memory bank are merged with the lists of the pool based on the migrate type of the allocation request. Similar to the approach in list based organization, when a page is to be taken from POOL_USER for kernel memory allocations (when POOL_OFFLINE is empty), page is allocated from a bank which has the highest free pages at the moment and rest of the free pages of that bank are added to POOL_KERNEL to minimize the overall footprint of kernel memory.

4.3 Page Migration

Page migration is required when an active session is followed by a relatively idle period. Figure 2.2 represents a scenario where all memory sections on Node-1 are occupied but most of them are more than 90% free. It becomes important to migrate pages from these memory banks in such conditions for operating system driven strategies to be effective in saving power consumed by DIMMs.

There are some execution points from where page migration can be called in a running system i.e., when a process exits (*do_exit()* in kernel/exit.c), some page caches are dropped (*drop_pagecache_sb()* in kernel/fs/drop_caches.c) or dynamic memory operations. However all of them have unnecessary overhead issues when it comes to migrating large blocks of memory. Some are them are listed below-

- Not all process exits result in significant reduction in memory occupancy. It is unwarranted to call page migration for such processes.
- There are times when *do_exit* is called many times in a short span (such as during *"make install"*). Calling page migration can create havoc in system performance for such cases.
- Page caches are dropped by the kernel normally when page allocations start failing or zone watermarks are being hit. This happens under heavy memory pressure situations which is not a good place to worry about memory power.
- Dynamic memory operations like *free, munmap* or *sbrk* are more costly to invoke page migration because they are executed even more frequently and often operates on small memory blocks.

To avoid complexities of a running system, page migration is done by a separate kernel thread. Its simplified functioning on a single zone is discussed below -

- In the first scan, it reads the meta-data of all memory banks which currently belong to POOL_USER to figure out the candidate banks which can be freed in this pass.
- Mark all candidate banks as isolated. It is done to avoid page allocations from candidate banks which will happen during actual page migration.
- Call the actual page migration code (*do_migrate_range* from *mm/memory_hotplug.c*) on candidate banks that involves copying the page content and modifying the page table entries for these pages.

Bank range	Interleaved With	
0-15	32-47	**Node 0**
16-31	48-63	
64-79	96-111	**Node 1**
80-95	112-117	

Table 4.1: 2-way Interleaving used with Adaptive-Buddy

4.4 Memory Interleaving

Adaptive-buddy allocator manages banks in a way which is independent of their position in memory array. It provides opportunity for reducing the number of referenced memory banks even when interleaving is enabled. It can be achieved by defining sets of memory banks and mapping interleaved banks to a single set. Memory online and offline operations have to take place on bank-set rather than individual memory banks.

To be able to do implement this in real world requires knowledge of actual bank boundaries and the form of interleaving. Since this mapping information is not available on our Xeon platform, we define this mapping statically in kernel to analyze how much it impacts the ratio of offlined and free memory. We define a one-to-one mapping as shown is table 4.1 for our implementation. Bank sets are defined by using one bank from the first column and its corresponding interleved bank from the second column. For example bank 0 and 32 form one set, 1 and 33 form another and so on. All the operations which were operating on memory banks are modified to work on these sets.

Why not ZONE_MOVABLE? ZONE_MOVABLE was introduced for the purpose of controlling memory fragmentation. High level memory allocation policy of adaptive-buddy is very similar to ZONE_MOVABLE i.e., restricting unmovable allocations happening from entire physical memory space. However, there are some practical issues with it comes to using ZONE_MOVABLE for memory power optimization -

- Size of ZONE_MOVABLE is determined by a command line parameter but users are generally not aware of their memory requirements in advance. Too small a size may not produce desired results and too big a size can cause kernel allocations to fail even if sufficient memory is available.
- Memory allocations will still be happening from entire address space spanned by the zone in random order causing unnecessary migration overhead.

Adaptive-Buddy does not suffer from any of the above problems because it keeps on resizing its memory pools based on the dynamics of execution environment and allocations do not fail

Name	Size (Bytes)			
	Buddy	**Bank-Array**	**Bank-List**	**Adaptive-Buddy**
zone	1920	68864	71040	5696
pg_data_t	17152	284928	293632	32256

Table 4.2: Size of memory management structures in different allocators. In NUMA machines, pg_data_t corresponds to a node and represents its memory layout

as long as free memory is available. It also keeps on changing the in-use memory banks as different applications run on the system by carefully migrating memory banks which incur minimum overhead in the process.

Table 4.2 shows the size of the two memory management structures affected by the modified buddy allocators. As discussed earlier in this chapter, per-bank buddy allocators introduce significant space overhead. Adaptve-Buddy reduces this overhead to a great extent and it is possible to further reduce the size of these structures. Since we know that each memory pool is using only a few sublists, we can eliminate from each pool those lists that are not required. For example, POOL_USER will never need MIGRATE_UNMOVALE or MIGRATE_RECLAIMABLE lists as we are not allocating such pages from this pool. Similarly, POOL_FREE contain only MIGRATE_MOVABLE pages. By providing each pool only those lists that are need, we can reduce the size of zone structure to 2368 bytes and pg_data_t to 18944 bytes. It can be done by using non-uniform *free_area* structures for different pools.

Chapter 5

RESULTS

In this chapter, we discuss the effectiveness of adaptive-buddy towards solving memory-hotplug and power management problems. Different applications were run for different cases as performance of virtual memory managers depend heavily on the workloads that are running on the system.

5.1 Experimental Setup

Experiments discussed in this thesis were run on a desktop machine with two Intel Nehalam-based Xeon E5620 CPUs with 32GB(8x4GB) memory. Each CPU has 4 physical cores (8 threads with hyper-threading enabled) operating at a based frequency of 2.4GHz. The system has 32KB L1 cache(D/I), 256KB L2 and a shared 12MB L3 cache. Separate data and instruction TLBs are present with L1 TLB having 64 and L2 TLB with 512 entries for 4-KB pages which is also the page size used in this work. All the experiments are on Ubuntu 12.04 with a recent Linux kernel (version 3.17.4) as default operating system.

Some experiments have also been performed on a single CPU based Intel Xeon E3-1265L v2 server processor to test adaptive-buddy on small memory systems. This processor comes with 4 cores (8 threads) with a base frequency of 2.5GHz and has been equipped with 4GB memory. Memory bank size used on this machine is 128MB as bank size on mobile devices is generally smaller than desktop or server memory modules.

Results discussed in this section do not include array-based or list-based strategies because of their practical limitations (e.g., size of data structures) and the run-time overhead. They were primarily used for observing the behavior of Linux memory manager which provided useful insights in designing a flexible solution like adaptive-buddy.

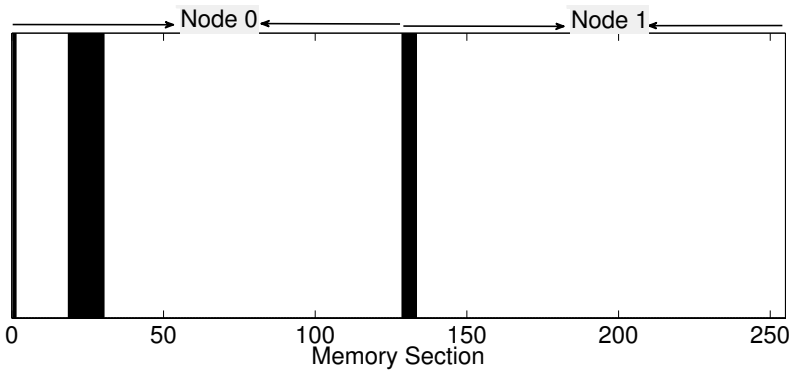


Figure 5.1: Kernel Memory Footprint with Adaptive-Buddy

5.2 Memory-Hotplug with Adaptive-Buddy

Fragmentation due to the mixing of movable and unmovable memory comes into picture in a long running system. Generally the total amount of unmovable memory is very low but because a page is always allocated from the head of free list which could belong to any memory section, it happens that most memory sections contain atleast a few unmovable pages. The same situation can occur easily when kernel has to keep a lot of meta-data information about the system such as while managing a large number of files or a large number of processes. Results discussed in this section pertain to a situation when file operations (open/read/write etc.) are being done on a large number of files.

We use *filebench* benchmark to generate a large number of small sized files and execute it on a filesystem mounted in physical memory i.e., *ramfs*. Number of files vary from 1,000 to 4,50,000 between different runs while the size of filesystem vary from 1GB to 30GB. When run multiple times, it deletes the filesystem which are mounted in the previous run resulting in a lot of memory freeing operations which randomizes the free lists.

Figure 5.1 shows the footprint of kernel memory on a total of 32GB system and where it gets stored on physical memory modules. The snapshot is taken after running the same sequence of *filebench* workloads which was used for standard buddy allocator in figure 2.4. As shown in the figure, unmovable memory is placed in contiguous memory sections starting from the first section of each memory zone which in turn simplifies the migration of a large contiguous chunks of memory.

Figure 5.2 shows how the number of removable memory sections decreases when system runs for a long time. As shown in the figure, an adaptive-buddy system does better than standard buddy at all times. It also suggests that in the beginning, when there are not much memory activities in the system, and free lists are not very random, a standard buddy system also does

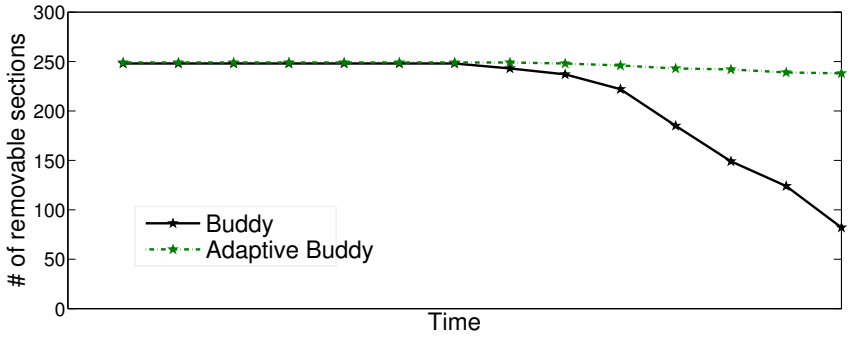


Figure 5.2: Comparison between Default and Adative-Buddy with respect to Memory-Hotplug

well. As soon as lists get randomized due to the repeated mounting and unmounting of *ramfs*, a standard buddy system fails drastically. When *filebench* finishes executing, the number of removable memory sections with adaptive-buddy is found to be 238 in contrast to 78 with standard buddy.

As discussed in previous chapter, when POOL_KERNEL is not able to satisfy a memory request and POOL_OFFLINE is also empty, it is very important to carefully select a memory bank belonging to POOL_USER to satisfy a kernel memory request. We observed the number of removable memory sections dropping to 28 with an arbitrary selection even with an adaptive-buddy system for the same workload which is significantly worse than a standard buddy system.

One point worth mentioning here is that a removable section does not guarantee that all pages belonging to this section can be migrated immediately. Real world situations are much more subtle and memory migration can still fail in some situations, such as when pages are undergoing heavy I/O. However, these conditions are temporal and keeping kernel memory on separate physical address space increases the probability of successfully migrating pages on bank granularity.

5.3 Memory Power Management with Adaptive-Buddy

Intel has recently started providing tools (e.g., *power-governer*) and libraries like *RAPL* (Running Average Power Limit) to measure power consumption of different system components including DRAM power. Unfortunately DRAM power measurement support is available on Sandy-Bridge microprocessor based server systems only. Since our experimental setup has a Nehalem microarchitecture based Xeon processor, we are unable to get actual power saving numbers. Due to this limitation, we produce these results in terms of offlined memory banks and percentage of free memory.

Workload	32GB			4GB	
	Max(%)	Average(%)		Max(%)	Average(%)
		Non-Interleaved	2-way Interleaved		
Light	81	59.5	55.2	64	48
Medium	56	45.7	40.7	45	26

Table 5.1: Offlined memory with Adaptive-Buddy

For power management purpose, we divide the workloads in two categories based on the memory footprint of all applications running on the system as below -

- **Light :** Memory footprint varying between 10%-50%.
- **Medium :** Memory footprint varying between 30%-80%.

Different workload scenarios were created by running applications from PARSEC benchmark suite. Applications were run with largest input set in a single threaded environment to sustain memory pressure for a long period of time. Other normal desktop applications like firefox, media player and document editors were also running in parallel. For medium workload, we also run a kernel build process on a clean kernel source tree in addition to the above applications and a few runs of *filebench* benchmark by mounting a filesystem in ramfs to increase memory pressure. Results summarized in table 5.1 are calculated over a period of more than 10 hours of medium workload and for 2-3 hours of light workload.

With our 2-way interleaved memory mappings, the average amount of offlined memory is little less than the non-interleaved solution for the same workload. It happens because while adding some memory from POOL_OFFLINE to the allocation pools, the operation takes place on a bank-set(banks which are interleaved together) rather than on a single bank. In situations where only a small amount of memory is need in allocation pools, the interleaved memory bank also has to stay in active pool even if it is not being used for allocation.

5.4 How Page Migration behaves with Adaptive-Buddy

Figure 5.3, which correnposnds to the medium workload of our experimental environment, shows a scenario where five memory banks were more than 90% free after some memory was released by applications and by dropping some amount of page cache. Page migration proves to be very helpful in this situation by first removing them from the allocation path and then quickly migrating their pages.

Figure 5.4 depicts the amount of free and offlined memory at all times during the experimental period in a running system where a total of 12GB memory was migrated over the entire

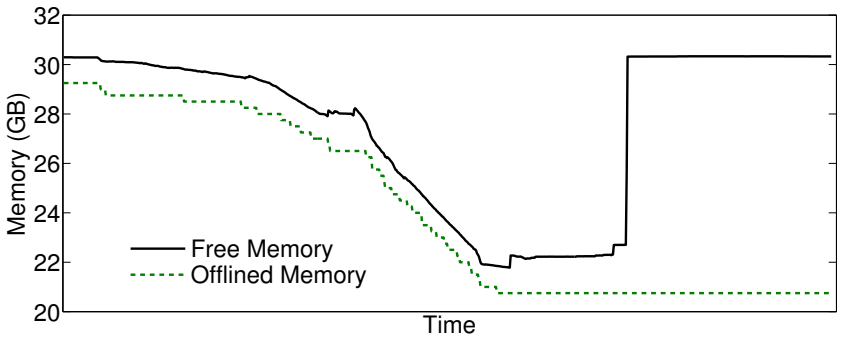


Figure 5.3: Free vs Offlined Memory (without migration) in Adaptive-Buddy

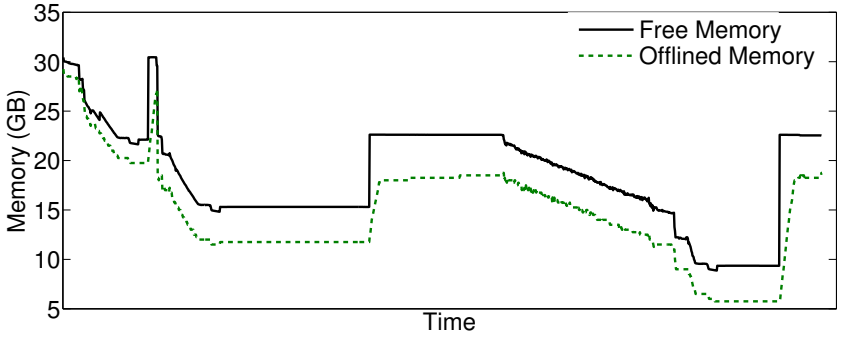


Figure 5.4: Free vs Offlined Memory (with migration) in Adaptive-Buddy

duration. It indicates that the amount of offlined memory is never too far from the amount of free memory even when a lot of memory gets freed suddenly. The total memory size of all memory banks that were offlined by adding to POOL_OFFLINE amounts to approximately 28GB. Figure 5.5 shows the amount of memory that was offlined vs the amount of memory that was actually migrated in the process.

5.5 Adaptive-Buddy vs an Optimal Solution

We can think of an optimal solution in terms of difference between the amount of free and offlined memory or in terms of percentage for a more generic sense. Let us say Δf is the percentage of free memory and Δd is the percentage of offlined memory. Effectiveness of a solution can be verified with the following relation between them -

$$\Delta d \approx c * \Delta f$$

Value of c, which represents the ratio of offlined and free memory, determines the effective-

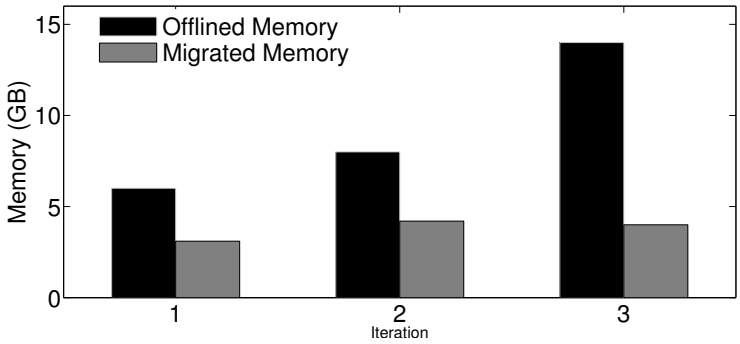


Figure 5.5: Comparison between Offlined and Migrated memory when page migration is invoked in an Adaptive-Buddy system

ness of a solution (higher is better). An optimal solution will keep it as close to 1 as possible. We have observed in our experiments that 1%-3% of total memory will be wasted even with the best of solutions when aggregated over all memory zones because each zone has some amount of free memory in the last unit of 256MB. c is varying between 0.75-0.90 at all times in case of adaptive-buddy with migration and does very well in average case (0.80 for medium and 0.85 for light workload). An average case comparison between the two on 32GB setup is summarized in table 5.2.

Workload	**Offlined Memory (%)**	
	Optimal	Adaptive-Buddy
Light	69	59.5
Medium	57	45.37

Table 5.2: Comparison between Adaptive-Buddy and an Optimal Solution

Since the difference between optimal and adaptive-buddy is not very high (15%-20%), we can safely assume that more sophisticated analysis on allocation or migration techniques will not provide substantial improvements over the proposed solution.

Chapter 6

EVALUATION

Adaptive-Buddy allocator changes the layout of kernel pages on physical memory modules and store them towards the lower address space of each memory zone at system startup. Application data is also stored in a subset of available address space as compared to the standard buddy system. Since memory is a critical resource, it becomes important to analyse the performance impact of these differences. We use a variety of benchmarks to measure the performance of Adaptive-Buddy on different system components and find no observable difference in comparison to the standard buddy allocator. In the next few sections, we look at different benchmarks used and the obtained results.

6.1 A Synthetic Benchmark

It was designed to measure the speed of memory allocation path for a large number of page requests as there are no standard benchmarks available which stress this particular subsystem in isolation. A micro-benchmark (*page_test*) from AIM9 does measure the speed of page allocation path but does not stress system memory. However, we will analyze this and other micro-benchmarks from AIM9 later in this chapter.

There are two components to this benchmark -

- **Source of Memory Allocation:** A simple *C* program which allocates small chunks of memory (4KB) using malloc successively. As malloc in itself does not cause an actual page frame allocation, we write a dummy value into each page only once to force page allocation and to make subsequent requests in quick succession. The process keeps on allocation upto 16GB memory without freeing anything to create memory pressure.
- **Allocation Time Measurement:** We instrument the kernel using *SystemTap*, a kernel instrumentation tool, to measure the time taken by *__rmqueue (in mm/page_alloc.c)* func-

No. of Pages	Size	Average (ns)		Worst(ns)	
		Buddy	Adaptive-Buddy	Buddy	Adaptive-Buddy
1 x 10^4	40 MB	285	267	755	1305
1 x 10^5	400 MB	266	260	988	2596
1 x 10^6	4 GB	260	255	999	20794
2 x 10^6	8 GB	250	255	999	20794
4 x 10^6	16 GB	249	277	999	21432

Table 6.1: Comparison between the allocation time of Buddy and Adaptive-Buddy

tion for the above process. Note that all memory requests generated by this process will be for O(1) pages as physical pages for a process heap are allocated only when they are actually accessed. It is important since we want to measure the same allocation sequence for two different kernels. We measure the *__rmqueue* function as it is the first point of change between the original and modified allocators.

Timing measurements obtained are shown in table 6.1. In the worst-case and especially over a large number of successive allocations, Adaptive-Buddy still introdues significant overhead despite being independent of the number of memory banks but performs very similar to the standard buddy allocator in average case. However, this benchmark is far from a realistic workload and because of the good average-case performance applications do not have an impact on their performance. We also intrumented page free operations and find no timing difference the two kernels.

6.2 Kernbench

Kernbech is a standard benchmark designed to compare different kernels on the same machine, or to compare hardware and is the most heavily used benchmark by kernel developers. It runs a kernel compile with multiple concurrent jobs many times. We measure the system time and user time separately for accurate results, take the average of five runs and find no measurable difference. For this benchmark, list based Bank-Buddy allocator shows some improvement in terms of user and real time. However, the difference is not very significant and hard to quanitify. Results are depicted in figure 6.1 and 6.2. We did not observe list based Bank-Buddy allocator behaving differently for other set of benchmarks used hence we will only compare Adaptive-Buddy and the standard Buddy allocator in the following sections.

6.3 Filebench

While kernbench provides useful statistics about overall system performance, it does not create enough memory pressure on our 32GB setup. We run *filebench* benchmark to test system per-

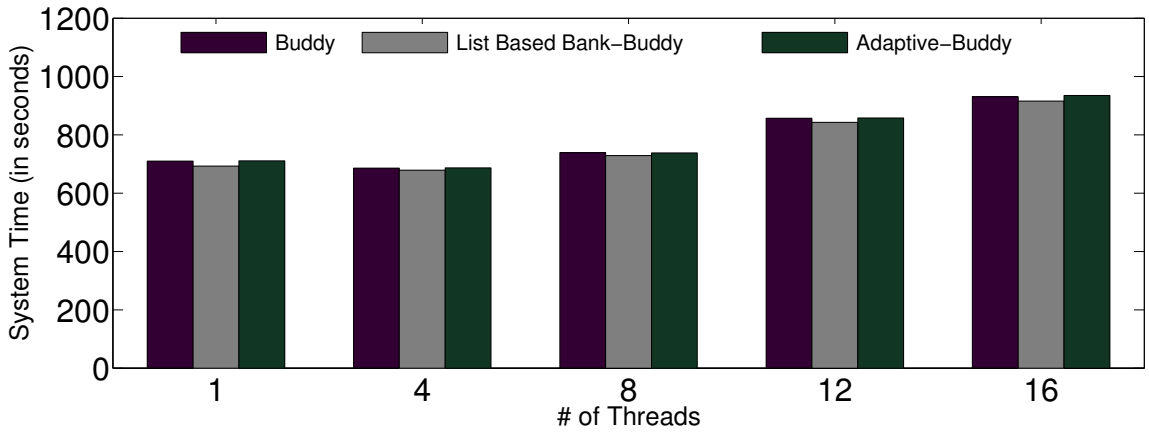


Figure 6.1: Comparison between kernbench system time

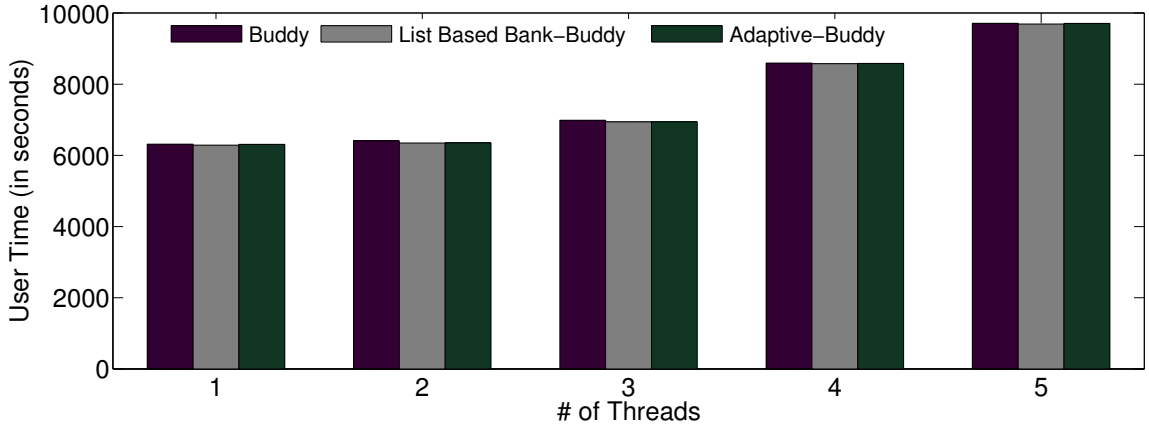


Figure 6.2: Comparison between kernbench user time

formance under heavy memory pressure. It is a benchmark for filesystem and storage specific execution environments and allows easy emulation of complex workloads. We run the benchmark with large number of small sized files (with *fileserver* workload provided with *filebench*) to create a mix of user and kernel memory usage. It performance filesystem related operations like create, read, write, open and delete file. We test the benchmark by mounting *ramfs* of different size (1GB to 28GB) with large number (upto 0.45 million) of files. Results (average of 3 runs) are shown in figure 6.3. Performance of adaptive-buddy is similar to that of standard buddy even under high memory pressure (29GB filesystem) when system throughput drops significantly.

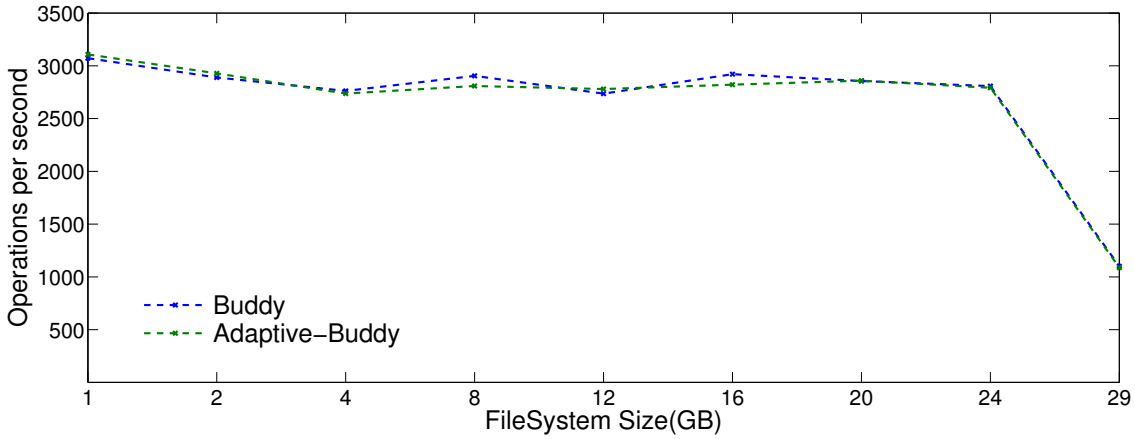


Figure 6.3: Filebench operations per second with different filesystem size

6.4 PARSEC

We also run applications from the PARSEC [Bienia 2011] benchmark suite which consists of 12 different benchmarks (9 applications and 3 kernels) covering a wide range of domains to measure the impact of Adaptive-Buddy on real scientific applications. PARSEC is a good representative of emerging scientific parallel workloads, a brief summary of applications is provided in table 6.2. We test the applications with different number of parallel jobs and notice no difference in their performance. Table 6.3 summarizes the execution time (system and user) taken by the applications with 16 threaded run of each on two different kernels.

Program	**Application Domain**	**Parallelization**		**Working Set**	**Data Usage**	
		Model	**Granularity**		**Sharing**	**Exchange**
blackscholes	Financial Analysis	data-parallel	coarse	small	low	low
bodytrack	Computer Vision	data-parallel	medium	medium	high	medium
canneal	Engineering	unstructured	fine	unbounded	high	high
dedup	Enterprise Storage	pipeline	medium	unbounded	high	high
facesim	Animation	data-parallel	coarse	large	low	medium
ferret	Similarity Search	pipeline	medium	unbounded	high	high
fluidanimate	Animation	data-parallel	fine	large	low	medium
freqmine	Data Mining	data-parallel	medium	unbounded	high	medium
streamcluster	Data Mining	data-parallel	medium	medium	low	medium
swaptions	Financial Analysis	data-parallel	coarse	medium	low	low
vips	Media Processing	data-parallel	coarse	medium	low	medium
x264	Media Processing	pipeline	coarse	medium	high	high

Table 6.2: Summary of PARSEC Application Benchmarks

Application	System Time (ms)		User Time (seconds)	
	Buddy	**Adaptive-Buddy**	**Buddy**	**Adaptive-Buddy**
bodytrack	864	672	290	290
blackscholes	3885	3915	352	352
canneal	7380	7432	297	299
dedup	27843	28363	56	55
facesim	9610	9836	1051	1058
ferret	2042	1900	813	818
fluidanimate	2934	2961	873	876
freqmine	2360	2382	987	981
streamcluster	12319	12614	1052	1094
swaptions	13373	12228	666	639
vips	4478	3848	273	272
x264	1603	1787	211	211

Table 6.3: Execution Time of PARSEC Benchmark Applications

6.5 AIM9

Finally we use *AIM9* to profile individual system components separately. It consists of several different micro-benchmarks for exercising different system areas such as I/O, process, memory, file system, networking and arithmatic operations. Table 6.4 provides a brief description of the tests used for the experiments along with the results obtained over a 60 second run for each. Apart from the mentioned tests, *AIM9* provide some more micro-benchmarks such as arithmatic operations (addition/ multiplication/division) and some math related series computation. These tests have also been tested, found to be behaving the same way and have been ommitted from results mentioned here. For these micro-benchmarks, throughput varies across different runs even for the same kernel and it is not feasible to quantify small differences. However, when we run each test for a long time and take an aggrerate over different runs, we see no apparent difference between the two kernels.

Test	Description	Operations per second		Difference
		Buddy	Adaptive-Buddy	
creat-clo	File Creations and Closes/second	151816	149883	-1.30%
page_test	System Allocations & Pages/second	552330	559781	+1.35%
brk_test	System Memory Allocations/second	3716713	3949383	+6.26%
exec_test	Program Loads/second	1396	1390	-0.43%
fork_test	Task Creations/second	3260	3193	-2.05%
link_test	Link/Unlink Pairs/second	133415	129502	-3.94%
disk_rr	Random Disk Reads (K)/second	432397	435468	+0.71%
disk_rw	Random Disk Writes (K)/second	350405	359620	+2.63%
disk_wrt	Sequential Disk Writes (K)/second	523008	537258	+2.72%
disk_src	Directory Searches/second	94153	93322	-0.88%
shared_memory	Shared Memory Operations/second	1054128	1053623	-0.05%
tcp_test	TCP/IP Messages/second	202990	202461	-0.26%
udp_test	UDP/IP DataGrams/second	458661	449836	-1.93%
fifo_test	FIFO Messages/second	1184720	1188516	+0.32%
stream_pipe	Stream Pipe Messages/second	1167396	1157741	-0.83%
dgram_pipe	DataGram Pipe Messages/second	1081990	1057350	-2.28%
pipe_cpy	Pipe Messages/second	1548410	1582370	+2.19%

Table 6.4: Summary of AIM9 Micro-Benchmark Test Results

We have also tested Adaptive-Buddy on *lmbench* which is another standard benchmark used to test different kernels and found no difference with it as well. Since none of the benchmarks show any measurable impact of Adaptive-Buddy on their performance except the worst case scenario of our synthetic benchmark, we conclude that an Adaptive-Buddy system does not suffer from any performance loss.

Chapter 7

CONCLUSIONS

References

[ACPI 5.0] Advance Configuration and Power Interface 5.0, 2011.
http://www.acpi.info/spec50.htm 13

[Google IO 2015] Google I/O Conference, 2015.
https://events.google.com/io2015 1

[JEDEC 2012] JEDEC DDR3 SDRAM Standard JESD79-3F, JULY 2012. 13

[FUJITSU 2011] Memory performance of xeon 5600 (Westmare-EP) based systems, 2011.
http://globalsp.ts.fujitsu.com/dmsp/Publications/public/wp-westmere-ep-memory-performance-ww-en.pdf 5

[Garg 2011] A. Garg, VM framework to capture memory reference pattern, 2011.
https://lwn.net/Articles/450217/ 12

[Micron 2005] Mobile DRAM Power-Saving Features/Calculations, 2005.
https://www.micron.com/ /media/documents/...note/dram/tn4612.pdf 1, 2

[Intel 2008] The Problem of Power Consumption in Servers, 2008.
https://software.intel.com/en-us/articles/the-problem-of-power-consumption-in-servers 1

[S. Bhat 2013] S. Bhat, Linux Memory Power Management, 2013.
https://lwn.net/Articles/568369/ 12

[Bienia 2011] Christian Bienia. *Benchmarking Modern Multiprocessors.* PhD thesis, Princeton University, January 2011. 30

[Brandt] Todd Brandt , Tonia Morris , Khosro Darroudi, "Analysis of the PASR Standard and its usability in Handheld Operating Systems such as Linux." 1, 2

[Delaluz 2002] Delaluz, V. and Sivasubramaniam, A. and Kandemir, M. and Vijaykrishnan, N. and Irwin, M. J., "Scheduler-based DRAM Energy Management", *Proceedings of the 39th*

Annual Design Automation Conference, DAC '02, pages 697-702, New York, NY, USA, 2002. 11

[A. Garg 2011] A. Garg, "Linux VM Infrastructure to support Memory Power Management", 2011.
https://lwn.net/Articles/445045/ 12

[Gorman 2007] Mel Gorman, "Create Optional ZONE_MOVABLE to partition memory between movale and non-movable pages", 2007.
https://lwn.net/Articles/224255/ 8

[M. Gorman 2005] Mel Gorman and Andy Whitcroft, "The What, The Why and the Where To of Anti-Fragmentation", 2006.
https://www.kernel.org/doc/ols/2006/ols2006v1-pages-369-384.pdf 7

[Gorman 2005] Mel Gorman and Patrick Healey, "Measuring the Impact of the Linux Memory Manager", 2005.
http://thomas.enix.org/pub/rmll2005/rmll2005-gorman.pdf 6

[Huang 2003] Huang, Hai and Pillai, Padmanabhan and Shin, Kang G., "Design and implementation of power-aware virtual memory", *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '03, pages 5–5, Berkeley, CA, USA, 2003. 11

[Huang 2005] Huang, Hai and Shin, Kang G. and Lefurgy, Charles and Keller, Tom, "Improving Energy Efficiency by Making DRAM Less Randomly Accessed", *Proceedings of the 2005 International Symposium on Low Power Electronics and Design*, ISPLED '05, pages 393-398, New York, NY, USA, 2005. 12

[Jantz 2013] Jantz, Michael R. and Strickland, Carl and Kumar, Karthik and Dimitrov, Martin and Doshi, "A Framework for Application Guidance in Virtual Memory Systems", *Proceedings of the 9th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '13, pages 155-166, New York, NY, USA, 2013. 1, 11

[Sueur 2010] Le Sueur, Etienne and Heiser, Gernot, "Dynamic Voltage and Frequency Scaling: The Laws of Diminishing Returns", *Proceedings of the 2010 International Conference on Power Aware Computing and Systems*, Hot-Power'10, pages 1–8, Berkeley, CA, USA, 2010. 1

[Lebeck 2000] Lebeck, Alvin R. and Fan, Xiaobo and Zeng, Heng and Ellis, Carla, "Power Aware Page Allocation", *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS IX, pages 105-116, New York, NY, USA, 2000. 11

[Malladi 2012] Malladi, Krishna T. and Lee, Benjamin C. and Nothaft, Frank A. and Kozyrakis, Christos and Periyathambi, Karthika and Horowitz, Mark, "Towards Energy-proportional Datacenter Memory with Mobile DRAM", *Proceedings of the 39th Annual International Symposium on Computer Architecture* (ISCA 2012), Portland, OR, USA, pages 37-48, 2012.

[Snowdon 2009] Snowdon, David C. and Le Sueur, Etienne and Petters, Stefan M. and Heiser, Gernot,"Koala: A Platform for OS-level Power Management", *In Proceedings of the 4th ACM European Conference on Computer Systems*, EuroSys '09, pages 289-302, New York, NY, USA, 2009. 12

[Zhang 2009] Zhang, Xiao and Dwarkadas, Sandhya and Shen, Kai, "Towards Practical Page Coloring-based Multicore Cache Management", *Proceedings of the 4th ACM European Conference on Computer Systems*, EuroSys '09, pages, 89–102, Nuremberg, Germany, 2009.

[Yun 2014] Heechul Yun, Renato, Zheng-Pei Wu, Rodolfo Pellizzoni, "PALLOC: DRAM Bank-Aware Memory Allocator for Performance Isolation on Multicore Platforms", *IEEE Intl. Conference on Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2014.