Towards Practical Page Placement for a Green Memory Manager

Ashish Panwar and K. Gopinath
Indian Institute of Science
{ashish.panwar, gopi}@csa.iisc.ernet.in

***Abstract*—Increased performance demand of modern applications has resulted in large memory modules and higher performance processors in computing systems. Power consumption becomes an important aspect when these resources go underutilized in a running system; e.g. during idle periods or lighter workloads. CPUs have come a long way in optimizing away the unnecessary power consumption in both hardware and software for such scenarios through solutions like Dynamic Voltage/Frequency Scaling. However, support for memory power optimization is still missing in modern operating systems despite hardware support being available for many years in the form of multiple power states and techniques like Partial Array Self-Refresh.**

In this work, we explore the behavior of Linux memory manager and report that even at 10% of memory utilization, there are references to all physical memory banks in a long running system due to random page allocation and ignorance of memory bank boundaries. These references can be consolidated to a subset of memory banks by using page migration techniques. Unfortunately, migration of large contiguous blocks is often restricted due to the presence of unmovable pages primarily owned by kernel.

We provide some techniques for utilizing the hardware facilitated Partial Array Self-Refresh by introducing bank awareness in the existing buddy allocation framework of Linux memory manager as well as for improving the page migration support of large contiguous blocks. Through a set of simple changes in Linux VM, we have been able to reduce the number of referenced memory banks significantly. Memory-hotplug framework, which relies on page migration of large contiguous blocks, also shows significant improvement in terms of number of removable memory sections. Benchmark results show no performance degradation in the modified kernel which makes the proposed solution desirable.

I. Introduction

Modern applications are growing in size and complexity at an unprecedented rate. It requires more memory and higher performance processors in all varieties of systems, ranging from small mobile devices to large scale server systems. Smart-phone devices are being shipped with multi-core processors and many GBs of memory while servers exist with TBs of main memory. Since systems are provided with plentiful resources, power consumption becomes a considerable factor when these devices stay idle. Google's latest *Doze* power management project [2] for Android M, which focuses on power optimization of idle devices, is a good indicator of growing concerns in this direction. CPUs have attracted a major focus of power optimization techniques primarily due to their significant contribution in overall system power. DVFS has been a successful technique in this context where hardware facilitated power modes are managed by operating systems depending upon the workloads. Intrinsically such optimizations make main memory a significant contributor in total system power during low workloads (or idle periods) e.g., 25%-40% in server systems [5, 17]. However, the main focus of memory power optimization till date has been on hardware resulting in memory modules supporting a wide range of power management features. Recent low power DDR3 SDRAMs support more than 10 different power states [4], management of which is typically handled by memory controllers.

Physical memory modules are organized in multiple banks each of which is a contiguous array of addresses. Such an organization allows memory to be optimized for power (via multiple power-states for individual banks) and performance (via bank-parallelism) at the same time. Some of the power states are shown in table-I along with their relative power consumption against peak power. Power savings are achieved by disabling some hardware circuitry or by reducing the current supply to memory banks in low power modes. For example, refreshing the DRAM capacitors is a very critical operation to preserve memory content. While it is normally done with the device clock, a bank in Self-Refresh mode makes use of its internal refresh counter for refreshing the capacitors and does not rely on the clock. Partial Array Self-Refresh (PASR), which allows refreshing only a subset of memory banks, is another way of reducing memory power consumption but needs to be handled carefully as it results in data loss of banks that are not being refreshed. Brandt et al. [9] proposes several implementation methodologies for reducing DRAM power consumption via PASR. In this paper, we will be working with Bank-Selective PASR where each memory bank can be put to Deep-Power-Down mode independently of other banks.

Operating systems can make use of data retention low power modes as well as PASR. For data retaining modes, it is required to control memory references in a way so that active pages reside on a subset of memory banks. However, it requires tracking the working set of processes in a running system which is a costly operation as physical memory references are transparent to operating systems. Though this information can be acquired on most platforms from page table reference bit [6], it involves a costly operation of traversing the entire virtual address space of processes and their page tables, clearing the reference bit for each page frame and flushing the

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

TABLE I: Relative power consumption of mobile DRAM power states [4].

TLBs. A sequential page table scan takes around 36 cycles per page table entry for collecting this information on a 3.0 GHz processor [19]. Such heavy duty operations are not practical for memory power optimization as they are more likely to result in unnecessary overhead or perhaps even more power consumption. Sacrificing the performance of memory manager is not a viable option in many situations especially when memory power is a lesser concern. Hardware can also help in collecting this information with much less overhead but such support is not available in traditional architectures.

For better utilization of PASR, a page allocator needs to be aware of memory bank boundary information. Memory controller can put unused banks to Deep-Power-Down mode without worrying about data loss if page allocations are consolidated to a subset of banks. In a long running systems, when in-use pages are scattered throughout the entire physical address space, page migration can be used to reduce the number of in-use memory banks. Linux already has a page migration framework (as part of memory-hotplug [1]) that operates on memory sections of 128MB (physically contiguous) each that can be used for this purpose. However, under the current memory management policies, this page migration is often not possible because of unmovable pages and requires policies to improve it for effective power management.

In this work, we present simple page placement algorithms for introducing power management in Linux memory manager. Primary contributions of this paper are -

- Providing an overview of how current Linux page allocator obstructs memory-hotplug and hardware facilitated low power modes for memory.
- Analyzing the challenges with respect to memory power management in multi-core systems and modern applications.
- Design and implementation of a few page placement algorithms for improving memory-hotplug and memory power management in Linux.
- Evaluating the performance of our proposed solution.

II. Related Work

Power management schemes developed over the years can be broadly classified in two categories. One, solutions that try to optimize memory power consumption per process by utilizing hardware facilitated low power modes with data retention. It is generally achieved by consolidating memory references (especially working set) of applications to a subset of memory banks. On the other hand, some techniques try to limit the number of in-use memory banks for all processes and focus on facilitating Deep-Power-Down mode. We take the later approach and try to minimize the number of memory banks allocated and leave the management of low power data retention modes to hardware.

David et. al. [10] uses DVFS (similar to what is used for power optimization of CPUs) to adaptively change the voltage and frequency of memory modules based on memory traffic to reduce memory power consumption. Jantz et. al. [17], 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 some crucial kernel data 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 might introduce significant overhead in the page allocation path in a busy system.

Some techniques for incorporating power optimization decisions with context switching have also been discussed. Delaluz et al. [11] 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. [15] maintains set of power-manageable units for each task and try 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 on context switch to alleviate the resynchronization delay associated with mode transition. Lebeck et. al. [18] proposes several techniques for power aware page allocation to manage power states of each memory chip. Hual et. al. [16] introduces the concept of hot and cold ranks which are created dynamically by keeping frequently accessed pages on same ranks using page migration. However, the solution relies on a modified memory controller to accurately track the memory

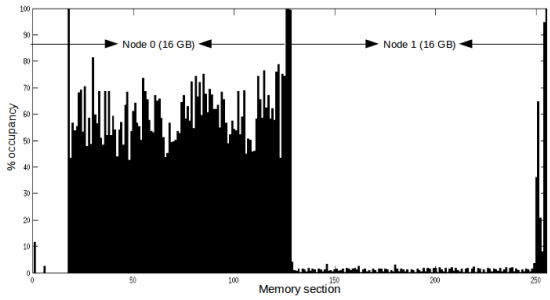


Fig. 1: Impact of local node arbitrary page allocation algorithm. Arbitrary allocation spreads allocations over all memory sections even if few pages are being used (Node-1). Local node policy allocates pages from the same node as running process as long as sufficient free memory is available (in this case Node-0).

[1]Memory-hotplug requires a memory section to be removable (i.e., all its in-use pages should be movable) before offlining it. For simplicity, we refer to removability of memory sections as memory-hotplug in this paper.

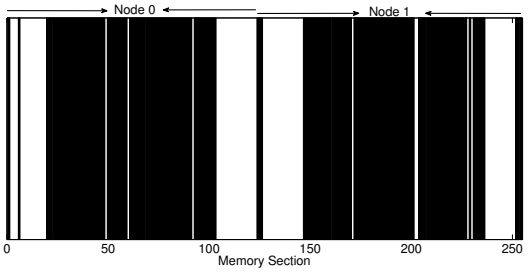


Fig. 2: Memory sections (colored black) being referenced by kernel in a standard buddy system (i.e., sections that contain atleast one kernel page each.)

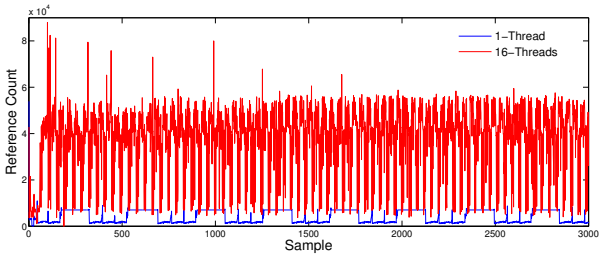


Fig. 3: Memory reference pattern of *facesim* with 1 and 16 threaded execution. Reference count is large as well as changing rapidly when running with 16 threads.

references and such support is not available on commercial platforms.

A. Garg [12] 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 [7] of the same work, eliminates this duplication by capturing power-manageable hardware units in a 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. They also migrate pages 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 solutions discussed above is their inflexibility to meet diverse execution scenarios of modern multi-core systems. Memory power management is highly dependent on executing workloads and comes into play only when there are sufficient idle periods to utilize low power modes of memory. Even though many of these techniques rely on page migration, the problem of non-removable memory sections has not been discussed. This paper addresses these practical issues associated with memory power management.

III. Background

A. *Overview of Linux Memory Manager*

Linux handles paging operations (allocation and free) at zone level with the help of a binary buddy allocator. It maintains MAX_ORDER (currently defined as 11) doubly linked lists in a data structure named as *free_area* in the kernel source. Each list, from order 0 to MAX_ORDER-1, represents 2^{order} contiguous pages in a single list entry. A buddy allocator is primarily chosen for its speed as it can allocate even large chunks of memory in a single lookup on *free_area* structure as long as free memory is available. Pages are always allocated from (and added to) the head of free lists for simplicity.

In a long running system, as processes allocate and free memory, free lists get randomized which creates two main problems with respect to memory power management and memory hotplug -

- Random page allocation from arbitrary memory banks scatters memory references over all of them. Figure-1 represents the number of in-use pages of each 128MB memory section on our 32GB 2-node desktop machine in a long running system. Note that none of the 128MB memory sections is completely free on Node-1 even though more than 90% of its pages are free. In this case, none of the memory banks can be put to Deep-Power-Down mode. Also note that major proportion of allocated memory is from Node-0 while most of the memory on Node-1 is free, reflecting the impact of local node allocation policies.
- Migrating a page involves copying the page content to another page and modifying all its corresponding page table entries to point to the new location. Since kernel pages are not mapped via page tables, it is not possible to figure out which virtual pages from kernel address space map to a given physical page. When an arbitrary physical page is allocated to the kernel, the entire memory section that corresponds to it becomes unmovable. It is very difficult to migrate memory sections due to this effect in a long running system because most of them contain some kernel pages. Figure-2 depicts a scenario where more than 170 out of 256 memory sections contain atleast one kernel page each.

Linux provides a solution for improving upon fragmentation and removability of memory sections in the form of an optional ZONE_MOVABLE. It creates a zone that is usable only for movable allocations (i.e., anon/page-cache pages owned by user applications). Size of this zone is determined by the *movablecore* (or *kernelcore*) parameter specified at boot-time [13]. Similar to ZONE_MOVABLE, we also consider everything to be unmovable except anonymous and page cache pages. Later in the paper, we discuss issues related to ZONE_MOVABLE.

B. *Challenges on multi-core Systems*

Since any combination of processes may be scheduled at a time on multi-core systems, strategies that minimize the number of memory banks (i.e., [15]) for individual applications no longer work in such systems. Modern applications are parallel in nature which poses another challenge with respect to

memory power optimization. To analyze the memory reference behavior of parallel applications, we use sequential page table scan [6], a common technique for tracking the active pages, for some applications. We calculate reference count for all threads of an application as the number of referenced bits corresponding to the page table entries that were set by the processor in the last 10 milliseconds period. Note that multiple references for a page are counted as one with the help of page table reference bit for private pages. However, for a shared page, reference will be counted once per thread that accessed it in the given sample. Hence the reference count represents only a subset of actual memory references. In our experiments, we make the following observations -

- Working sets grow large in proportion to the thread count for most applications.
- Working sets change very rapidly in a multi-threaded execution.

Figure-3, which represents reference count for *facesim* for 1 and 16-threaded execution (calculated after every 100ms for a 10ms sample period), shows the complexity of working sets. A 16-threaded execution for this application accesses approximately 50% of the pages from its resident set within each sample, 23% for 8 threads, 13% for 4 threads and a mere 5% in single threaded run in the average case. Figure-3 also shows that reference count is changing very rapidly for 16-threaded run which makes it even more complex for an operating system to keep track of pages that can provide idle periods between successive memory references.

Even if the problem of finding inactive pages may be feasible to solve, it would require heavy computation. Operating systems generally do not have the luxury of either space or time to afford such heavy duty operations. Another factor that makes it hard to achieve idle periods between memory references is the increasing size of memory banks. A 256MB memory bank has approximately 65k pages which is likely to grow even more for future memory modules. Considering the speed of modern processors, a few active pages are sufficient to keep an entire memory bank active.

On the contrary, if operations are handled at a higher-level i.e., at allocation time, references could be consolidated to a subset of memory banks relatively easily when memory pressure is not very high. We intend to achieve this in our design that is discussed in the next section.

IV. Design and Implementation

The objective in designing a power aware allocation framework is to consolidate memory allocations to a subset of banks. If bank boundaries are known, page placement can be handled at different levels -

- **Node:** In this, memory allocation policies are defined at node level, on top of memory zones. Linux Memory Power Management framework by A. Garg [12] and S. Bhat [7] fall in this category.
- **Zone:** Memory allocations can be controlled at zone level also by making the zone buddy allocator aware of bank boundary information.

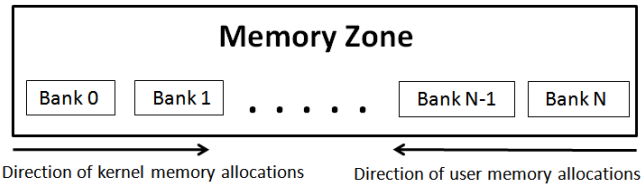


Fig. 4: Array-based Bank-Buddy Allocator

- **Bank:** Most of the proposed solutions (such as [15], [17]) use per bank buddy allocator which reside inside zone and provide guidance to the allocator at zone level. Array-based and List-based approaches discussed in this paper fall under this category.
- **Pool:** We suggest one alternative solution in the form of Adaptive-Buddy which manages physical pages in memory pools to reduce the overhead and complexities associated with per bank allocators.

However, the actual bank boundary information is not known on all platforms. On some architectures (like ARM and PowerPC), it can be acquired via device tree but is not available on x86 based architectures yet. ACPI 5.0 [1] mandates the use of Memory Power State Table (MPST) to provide information of all independently power manageable memory units which is likely to be available soon. Due to the lack of actual bank boundary information, we consider each memory bank to be 256MB[2] based on the memory module size and JEDEC standards [3] for DDR3. However, for experimental purposes, this is not a problem because their actual transitioning between power modes is handled by memory controller which is transparent to OS. Once the address mapping is fixed, rest of the subsystem does not require further changes.

A. *Array-based Bank-Buddy Allocator*

In this approach, each zone, rather than having a single large *free_area* structure, maintains its free pages in an array of *mm_bank* structures. Each *mm_bank* structure corresponds to a contiguous set of physical pages. Pages can be allocated from any one direction i.e., higher bank number to lower bank or vice-versa once free lists are initialized. This makes sure that a completely free bank is not used for a new request as long as already in-user banks have the capability to satisfy it. We follow different directions for user and kernel memory to avoid mixing of movable and unmovable pages which improves the removability of memory sections. In this context, we are considering kernel memory from lower to higher banks and user memory from higher to lower banks as shown in figure-4.

Over a period of time, all memory banks may be used to satisfy memory demands of applications. When relatively light workloads follow a busy session and free pages are released

[2]JEDEC standardizes 8 memory banks per rank on a memory module. We have 4GB dual rank DIMMs and hence consider bank size to be 256 MB.

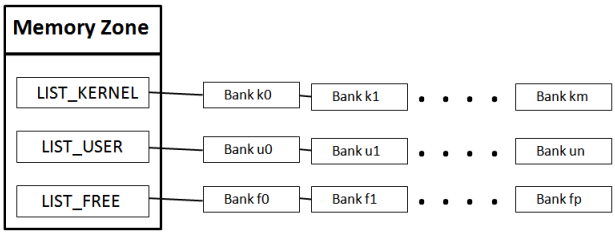


Fig. 5: List-based Bank-Buddy Allocator

to the allocator, migration can be used to move pages from lower memory banks of user address space to higher banks.

There are two practical issues associated with Array-based Bank-Buddy allocator-

*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 some situations it is possible to optimize the run time cost by passing some hints to allocator. But in the worst case, when most of the memory has been allocated and banks have very few pages left, the allocator would end up scanning the entire bank array resulting in an O(n) operation, where n is the number of memory banks in the zone.

*2) **Unnecessary Page Migration Overhead**:* Keeping kernel memory towards the beginning of bank 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 address space to higher banks, it can result in significant migration overhead as the number of free pages in a bank depends on the order in which pages are released. It can easily happen that a lower memory bank requires significantly more pages to be moved than a higher bank.

There is no impact on the performance of page freeing path as the page being freed is directly mapped to its corresponding bank using the page frame number. Cost of freeing a page remains O(1) for the subsequent solutions as well and will not be discussed further.

B. 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. Removability of memory sections can be improved using separate lists for kernel and user memory i.e., LIST_KERNEL and LIST_USER. Free memory banks reside on a separate list named as LIST_FREE. Figure-5 depicts the list based organization of memory banks.

LIST_KERNEL and LIST_USER are empty lists at system start-up 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 requirements 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 after doing so. Selecting a memory bank carefully from LIST_USER for a non-movable request is very important. In some cases we observe that an arbitrary selection behaves worse than the standard buddy allocator with respect to memory-hotplug.

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

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

While list based approach helps in reducing the unnecessary 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. 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 in following ways -

- **Allocate from the most recently used bank:** In this, we try to allocating a page from a memory bank which resulted in the most recent successful allocation before falling over the list of memory banks.
- **Allocate from a bank with highest free pages:** 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. The first approach is not always fruitful especially when all the memory banks have very little memory left in their free lists. A problem with the second approach lies in determining how frequently the candidate selection should be done to place it at the head of bank list. If selection is invoked very frequently, it may result in lock contention over zone structure. It may not be acceptable in some situations because zone lock is already one of the most highly contented locks in Linux kernel. Putting the thread to sleep for a long time can result in the default behavior i.e., O(n) for a single allocation.

C. Issues with Bank-Buddy Allocators

There are two serious issues with the per bank allocators discussed above -

- When there are multiple structures, each with its own list of free pages, it results in an O(n) allocation time in the

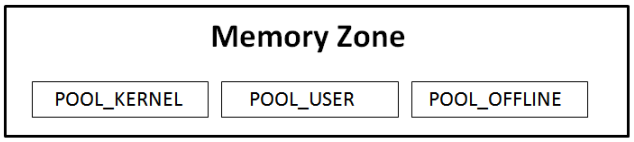


Fig. 6: Memory Pools in Adaptive-Buddy Allocator

worst case. Performance is a serious issue when system starts running out of memory. Such allocation overhead is not acceptable in most situations and should be avoided.

- Size of memory management structures (e.g., struct *mm-zone*) could grow very large if *free_area* is initialized on each memory bank which in itself is quite large (1144 bytes for MAX_ORDER free lists on x86_64). When array or list of memory banks is traversed, a large portion of memory caches may be swiped out causing performance loss for applications and possibly OS jitter. The fact that these structures are accessed quite often in Linux makes it a critical factor. The size of two most relevant memory management structures for different implementations is shown in table-II.

D. Adaptive-Buddy Allocator

In this approach, all memory banks belonging to the same list in List-based approach are 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-6. 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. However, when a page is to be taken from POOL_USER for kernel memory allocations (when POOL_OFFLINE is empty), page will be allocated from a bank with maximum number of free pages at the moment.

A Deeper Look at Page Allocation Cost: Adaptive-Buddy allocates movable pages from POOL_USER, POOL_FREE or POOL_KERNEL in the same order depending upon the availability of free pages. This implementation is independent of the number of memory banks and results in an O(1) operation (but only for movable pages). While allocating unmovable pages, it may have to traverse through the movable banks (when POOL_KERNEL and POOL_FREE both are empty) to find a suitable candidate. It results in an O(n) operation which can be optimized to yield O(1) performance by maintaining memory banks in a heap like data structure (on the basis of number of free pages). Such an arrangement would require some computation every time the number of free pages is updated in a bank. Considering the fact that unmovable allocations constitute only a fraction of overall memory demand, we try to avoid this extra computation. Experimentally, we observed very few of these O(n) operations which does not affect the overall performance of memory manager.

When memory bank crosses zone boundary: Since each memory node except node-0 has only one populated zone in a system, there are atmost two such cases when a memory bank may be spanned by two zones i.e., on the first node. We always keep these banks in allocation pool for two reasons. First, it does not impact the number of memory banks that can be offlined otherwise since some memory is always used from all zones. Second, it results in a much simpler implementation.

E. *Page Migration*

It becomes important to migrate pages for operating system driven policies for effective power management in situations like the one depicted in figure-1 where even though most memory is free, references are still scattered. 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 they are more likely to result in unnecessary overhead 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 release. 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.

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 III: Comparison between the allocation time of standard and Adaptive-Buddy.

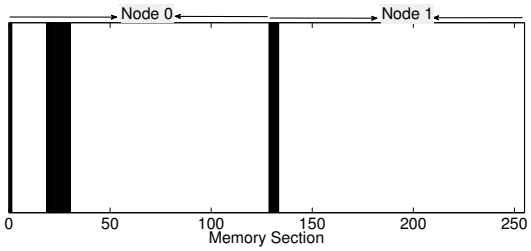


Fig. 7: Kernel memory footprint (black sections) with Adaptive-Buddy.

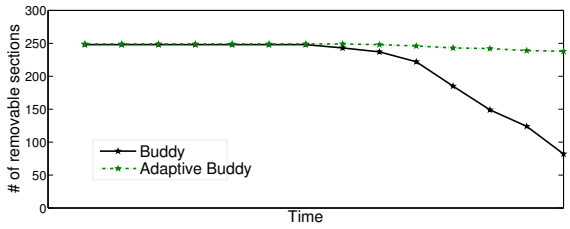


Fig. 8: Comparison between standard and Adaptive-Buddy page placement on memory section removability.

- Mark all candidate banks as isolated. It is done to restrict page allocations (that happens during migration) from candidate banks. Note that isolating the candidate banks is very crucial to avoid multiple migration for a single page.
- 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.

In our experiments, we observed that page migration need not be invoked very frequently. The thread sleeps for 2 seconds initially between successive calls but keeps on doubling its sleep time if consecutive iterations detect no opportunity of migrating pages. However, maximum sleep time allowed is 60 seconds. Once the thread sleeps for 60 seconds, it keeps on sleeping for same amount of time between successive calls unless page migration is needed. Once page migration is performed, sleep time is again reduced to 2 seconds to be effective in situations where a lot of memory is being freed successively.

F. A Potential Solution for Interleaved Memory

The Adaptive-Buddy allocator manages banks in a way that is independent of their position in memory array. It can provide a solution 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.

However, a practical solution requires information about the interleaved mapping of memory banks. To the best of our knowledge, this mapping information is not available to operating systems which demands further study on this topic. For a prototype implementation, we define a 2-way mapping statically in kernel which results in comparable results in terms of the percentage of offlined memory as compared to the non interleaved implementation as shown in table-IV.

G. 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-

- Size of ZONE_MOVABLE is determined by a command line parameter but users are generally not aware of the memory requirements in advance. Too small a size may not produce desired results and too big a size can cause unmovable allocations to fail even if sufficient free memory is available.
- Some control in still required to prevent random memory allocations.

Adaptive-Buddy provides the necessary control and keeps on resizing its memory pools based on the execution environment. It also makes sure that allocations do not fail as long as free memory is available.

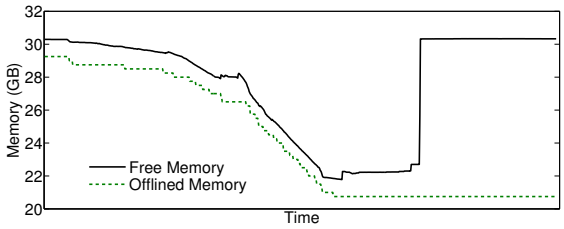
V. RESULTS

In this section, we discuss the effectiveness of Adaptive-Buddy towards solving memory-hotplug and power management problems. Since we are not concerned about application specific behavior in this work, we do not worry much about what applications are running. However, we use only realistic workloads (except one synthetic benchmark used for measuring page allocation time) during our experiments for a generic evaluation (mostly applications from PARSEC benchmark suite).

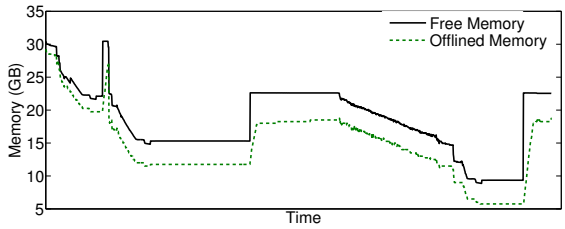
Experimental Setup: Experiments discussed in this paper were run on a desktop workstation machine with two Intel Nehalam-based Xeon E5620 CPUs with 32GB(8x4GB) memory. Each CPU has 4 physical cores (8 threads with hyper-threading enabled) operating at a base frequency of 2.4GHz. Some experiments have also been performed on a system with 4GB memory (with 128MB bank size) as well to test Adaptive-Buddy on small memory systems. All the experiments are on Ubuntu 12.04 with a recent Linux kernel (version 3.17.4) as default operating system.

A. Memory-Hotplug with Adaptive-Buddy

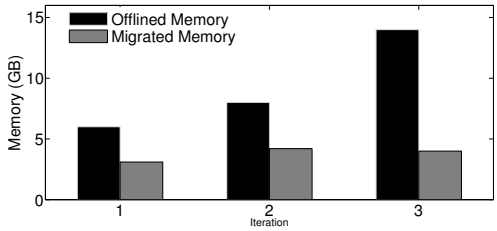
We use *filebench* benchmark to generate a large number of small sized files and execute it on a file system mounted in physical memory i.e., *ramfs*. Number of files vary from 1,000 to 4,50,000 between different runs while the size of file system varies from 1GB to 29GB. When run multiple times, it deletes the file system which was mounted in the previous run



(a) Without Page Migration



(b) With Page Migration



(c) Offlined vs Migrated Memory for three different iterations of page migration

Fig. 9: Behavior of an Adaptive-Buddy System.

resulting in a lot of memory freeing operations randomizing the free lists.

Figure-7 shows the footprint of kernel memory on physical address space. The snapshot is taken after running the same sequence of *filebench* workloads which was used for standard buddy allocator in figure-2. 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 from user pool.

Figure-8 shows the comparison between standard and Adaptive-Buddy allocator for the same execution sequence. Standard buddy allocator does well in the beginning because a large number of free pages belong to unmovable free lists at system reboot. These pages were used during system initialization for loading kernel, kernel modules and by boot-memory allocator. Once this pool of memory gets saturated, the impact of Adaptive-Buddy is easily visible against the standard buddy allocator. In some cases we see as much as 85%-90% improvement in terms of the number of removable memory sections.

Note that a removable section does not guarantee that all pages belonging to it can be migrated immediately. Real world situations are much more subtle and memory migration can still fail in some situations e.g., under heavy I/O. However, these conditions are temporal and keeping kernel allocation on separate physical address space increases the probability of successfully migrating pages on bank granularity.

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 IV: Percentage of Offlined Memory with Adaptive-Buddy (and page migration).

B. *Power Management with Adaptive-Buddy*

Intel provides 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. Due to the lack of actual bank boundary mapping and power measurement support from hardware, we are unable to get actual power savings and hence provide results in terms of offlined memory banks.

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 workloads were created by running applications from PARSEC benchmark suite. Applications were run with

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 V: Execution Time of PARSEC Applications.

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 VI: Summary of AIM9 Micro-Benchmark Test Results.

largest input set in a single threaded environment to sustain memory pressure over 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 file system using *ramfs* to increase memory pressure. Results summarized in table-IV are calculated over a period of more than 10 hours of medium workload and for 2-3 hours of light workload.

C. *How Page Migration Helps*

In figure-9a, 5 memory banks were more than 90% free after memory was released at once by applications. Page migration proves to be very helpful in this situation by quickly migrating pages from these banks and removing them from the allocation path. Figure-9b (medium workload) depicts the way migration works in a running system where a total of 12GB memory was migrated over the entire duration while total memory size of all memory banks that were offlined by adding to POOL_OFFLINE amounts to approximately 28GB. Figure-9c indicates the effectiveness of our candidate selection for page migration ($\approx$ 40% pages need to be migrated on average to offline a memory bank).

D. *Adaptive-Buddy vs an Optimal Solution*

We can think of a hypothetical 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 effectiveness of a solution (higher is better). An optimal solution which performs ideally in all cases will keep it as close to 1 as possible. 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).

Since the difference between optimal and Adaptive-Buddy is not very high, we can assume that more sophisticated analysis on allocation or migration (e.g., Statistical/ Machine Learning) will not provide substantial improvements over our methodology.

E. *Performance Evaluation*

Since physical memory is a very critical resource, changes in memory management subsystem may cause significant performance impact. Sometimes it can be very non-intuitive e.g., page-coloring [19] and performance-isolation [14]. We study the performance of Adaptive-Buddy with a variety of benchmarks discussed below.

*1) **A Synthetic Benchmark**:* There are no standard benchmarks available for measuring the speed of page allocation path for a large number of successive requests[3]. We design a synthetic benchmark to test the effectiveness of Adaptive-Buddy allocation algorithm. 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 actual page 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 allocating 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)* function 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

[3]*page_test* from AIM9 does measure the speed for O(1) page allocations but frees the allocated page before making the next request and hence does not create memory pressure in the system.

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 kernel.

Timing measurements obtained are shown in table-III. In the worst-case and especially over a large number of successive allocations, Adaptive-Buddy still introduces 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 worst case behavior for this benchmark appears when POOL_USER is free and a memory bank from POOL_OFFLINE is added to it before satisfying current request. Adding an entire memory bank to the allocation pool in such a case makes sure that the subsequent requests are satisfied quickly resulting in good average case performance. Note that this benchmark is far from a realistic workload and because the average case scenario is still good, applications do not reflect any performance impact. We also instrumented page free operations and found no timing difference between the two kernels.

*2) **PARSEC Benchmarks**:* We test applications from the PARSEC [8] benchmark suite which consists of 12 different benchmarks (9 applications and 3 kernels) covering a wide range of domains to measure the performance of Adaptive-Buddy on real applications. PARSEC is a good representative of emerging scientific parallel workloads. Table-V summarizes execution time taken by the applications with 16-threaded run of each on vanilla and modified kernels. To understand the impact of Adaptive-Buddy on TLB and system caches, we profile PARSEC applications (different run than the timing tests) using *perf*. No quantifiable difference was found for the profiled events of TLB, L1-Cache and Last Level Cache(LLC) as well.

*3) **AIM9 Micro-Benchmarks**:* Finally, *AIM9* is used to profile individual system components separately. It consists of several micro-benchmarks for exercising different system areas such as I/O, process, memory, file system and networking. Table-VI provides a brief description of the tests used for the experiments along with the results obtained over a 60 seconds run for each test. For these micro-benchmarks, throughput varies across different runs even for the same kernel and it is not feasible to quantify such small differences. However, when we run each test for a long time and take an aggregate over different runs, we see no apparent difference between the two kernels.

The benchmarks discussed above indicates that Adaptive-Buddy does not have any negative impact on the overall system performance which makes it a desirable solution. However, one aspect of its performance i.e., impact on memory latency still needs to be analyzed which requires exact mapping of the bank boundaries and could not be done on our platform.

VI. Conclusion

This paper presents several page placement algorithms for improving power management and memory-hotplug support in Linux by modifying and inducing bank awareness in standard buddy allocator. We present the effectiveness of Adaptive-Buddy toward solving these problems with different use-cases. Since Adaptive-Buddy shows promising results without affecting system performance we believe it could be an answer to the problem of memory power consumption on a variety of systems. Next we intend to integrate Adaptive-Buddy on a platform that provides memory bank boundary information and support for memory power measurement.

References

[1] Advance configuration and power interface specification 5.0, 2011. . http://www.acpi.info/spec50.htm.

[2] Google I/O conference, 2015 . https://events.google.com/io2015/.

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

[4] Mobile dram power-saving features/calculations . https://www.micron.com/ /media/documents/...note/dram/tn4612.pdf.

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

[6] Looking Inside Memory, Tooling for tracing memory reference patterns. In *Proceedings of the Linux Symposium*, Canada, 2010.

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

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

[9] Todd Brandt, Tonia Morris, and Khosro Darroudi. Analysis of the pasr standard and its usability in handheld operating systems such as linux.

[10] Howard David, Chris Fallin, Eugene Gorbatov, Ulf R. Hanebutte, and Onur Mutlu. Memory power management via dynamic voltage/frequency scaling. In *Proceedings of the 8th ACM International Conference on Autonomic Computing*, ICAC '11, pages 31–40, New York, NY, USA, 2011. ACM.

[11] V. Delaluz, A. Sivasubramaniam, M. Kandemir, N. Vijaykrishnan, and M. J. Irwin. Scheduler-based dram energy management. In *Proceedings of the 39th Annual Design Automation Conference*, DAC '02, pages 697–702, New York, NY, USA, 2002. ACM.

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

[13] Mel Gorman. Create optional ZONE_MOVABLE to partition memory between movable and non-movable pages . https://lwn.net/Articles/224255/.

[14] Z. Wu R. Pellizzoni H. Yun, R. Mancuso. PALLOC: DRAM Bank-Aware Memory Allocator for Performance Isolation on Multicore Platforms, RTAS, 2014.

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

[16] Hai Huang, Kang G. Shin, Charles Lefurgy, and Tom Keller. Improving energy efficiency by making dram less randomly accessed. In *Proceedings of the 2005 International Symposium on Low Power Electronics and Design*, ISLPED '05, pages 393–398, New York, NY, USA, 2005. ACM.

[17] Michael R. Jantz, Carl Strickland, Karthik Kumar, Martin Dimitrov, and Kshitij A. Doshi. A framework for application guidance in virtual memory systems. In *Proceedings of the 9th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '13, pages 155–166, New York, NY, USA, 2013. ACM.

[18] Alvin R. Lebeck, Xiaobo Fan, Heng Zeng, and Carla Ellis. Power aware page allocation. In *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. ACM.

[19] Xiao Zhang, Sandhya Dwarkadas, and Kai Shen. Towards practical page coloring-based multicore cache management. In *Proceedings of the 4th ACM European Conference on Computer Systems*, EuroSys '09, pages 89–102, New York, NY, USA, 2009. ACM.