Controlling Physical Memory Fragmentation in Mobile Systems

Sang-Hoon Kim[†] Sejun Kwon[‡] Jin-Soo Kim[‡] Jinkyu Jeong[‡]

[†]Korea Advanced Institute of Science and Technology, Daejeon, South Korea

[‡]Sungkyunkwan University, Suwon, South Korea

sanghoon@calab.kaist.ac.kr sejun000@csl.skku.edu jinsookim@skku.edu jinkyu@skku.edu

Abstract

Since the adoption of hardware-accelerated features (e.g., hardware codec) improves the performance and quality of mobile devices, it revives the need for contiguous memory allocation. However, physical memory in mobile systems is highly fragmented due to the frequent spawn and exit of processes and the lack of proactive anti-fragmentation scheme. As a result, the memory allocation for large and contiguous I/O buffers suffers from highly fragmented memory, thereby incurring high CPU usage and power consumption.

This paper presents a proactive anti-fragmentation approach that groups pages with the same lifetime, and stores them contiguously in fixed-size contiguous regions. When a process is killed to secure free memory, a set of contiguous regions are freed and subsequent contiguous memory allocations can be easily satisfied without incurring additional overhead. Our prototype implementation on a Nexus 10 tablet with the Android kernel shows that the proposed scheme greatly alleviates fragmentation, thereby reducing the I/O buffer allocation time, associated CPU usage, and energy consumption.

Categories and Subject Descriptors D.4.2 [*Operating Systems*]: Storage Management—Main memory; D.4.2 [*Operating Systems*]: Storage Management—Allocation / deallocation strategies

General Terms Design, Management, Performance

Keywords Memory fragmentation, Mobile systems

1. Introduction

Since the advent of the memory management unit (MMU) and scatter/gather direct memory access (DMA), the fragmentation of physical memory has not been a primary concern. These two hardware-supported features have eliminated the need for allocating contiguous memory by providing a contiguous address space over physically scattered memory pages. Consequently, physical memory is split into a unit size, or physical page, and is managed at that granularity, which leads to fragmenting physical memory at the page granularity [13, 14].

Emerging mobile systems, such as smartphones, have revived the need for contiguous memory for their I/O buffers. Some device vendors statically reserve a part of memory or adopt similar approaches to secure the required large buffers (tens of MB) [18, 19]. Some other vendors adopt an input-output memory management unit (IOMMU) [23, 28], which can eliminate the need for the reservation and allocation of contiguous memory by dynamically mapping physical pages to I/O addresses. However, using large pages (e.g., 64 KB or 1 MB) instead of small ones (4 KB) has proven efficient in terms of allocation time, mapping and unmapping costs, and I/O translation look-aside buffer (IOTLB) overhead [7, 36]. The operating system (OS), therefore, still needs to provide contiguous memory allocations.

Previous approaches to providing contiguous memory allocations relied on compacting physical pages [13, 14]. When a contiguous memory allocation cannot be handled in a given free space, the allocator compacts memory by migrating movable pages. This solution has been acceptable for a small amount of small contiguous memory allocations (a few MBs at the granularity of 8–32 KB) because the required contiguous memory can be secured by migrating only a few pages. This process, however, is impractical to satisfy the significant number of large contiguous memory allocations for I/O buffers (hundreds of MBs at the granularity of 64 KB or 1 MB).

This paper proposes a region-based physical memory management scheme that is robust to the memory fragmentation in mobile systems. Different from PCs or server systems, memory allocation and deallocation in mobile systems have a unique characteristic that is associated with application caching [3, 35]. Virtual memory (VM) pages for an application are allocated independently via demand paging, whereas the pages are deallocated together when the appli-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.

Copyright is held by the owner/author(s). Publication rights licensed to ACM.

ISMM'15, June 14, 2015, Portland, OR, USA
ACM. 978-1-4503-3589-8/15/06
http://dx.doi.org/10.1145/2754169.2754179

cation is terminated for memory reclamation. By exploiting this characteristic, the pages can be grouped at allocation time even if their lifetimes are unknown, and be isolated from the pages with different deallocation times. To this end, a *region domain* is created and associated with a process when the process is spawned for an application. The region domain contains fixed-size *regions*, each of which is physically contiguous and hosts the VM pages of the process. When a cached application is killed to reclaim memory, the regions allocated to the application are freed. Accordingly, a successive contiguous memory allocation can be satisfied from the freed regions without requiring significant effort for compacting pages.

In order to make this scheme practical, (1) internal fragmentation in region domains should be minimized, and (2) all pages other than VM pages should also be managed compactly without fragmenting physical memory. First, internal fragmentation occurs in the form of unused free pages in regions. We greatly reduce the internal fragmentation by compacting each region domain and sharing region domains for applications sharing the same high-level lifetime. Second, the pages other than VM pages are managed in special region domains that are optimized for their characteristics. For instance, file pages are allocated in a region domain that is maintained in a compact state by collaborating with the system-level page replacement process. Unmovable kernel pages are confined in a special region domain so that they do not aggravate the fragmentation.

The proposed scheme is implemented on an Android tablet and evaluated with a realistic workload. The evaluation results show that the proposed scheme effectively prevents memory from being fragmented when compared to the stock memory allocation policy. As a result, I/O buffers can be comprised of large pages within an order of magnitude shorter allocation time. Since I/O buffers are not fragmented, the overhead associated with allocating and managing the buffers is greatly reduced, thereby decreasing CPU usage by 20% as well as energy consumption by 16%.

The rest of the paper is organized as follows. The following section describes the current approaches to handling contiguous memory allocation and the characteristics of memory management in mobile devices. Section 3 characterizes the memory usage in mobile systems, and Section 4 describes the proposed region-based physical memory management. Implementation issues are illustrated in Section 5. Section 6 shows the evaluation results of the proposed scheme. Section 7 discusses the related work. This paper concludes in Section 8.

2. Background and Motivation

2.1 Memory Management Regarding Fragmentation

Memory fragmentation is an inherent problem in dynamic storage allocation with variable size units [20]. However, it has not been a major concern in modern OSes, since physical memory is divided and managed at a unit size: a page, for example. On CPU-side, memory management unit (MMU) supports dynamic translation from virtual to physical addresses. Accordingly, a process can see a virtual contiguous address space while corresponding physical pages are scattered. On I/O device-side, scatter/gather DMA supports similar dynamic mappings between a contiguous I/O address space and scattered physical pages. Therefore, the fragmentation of physical memory has not been a problem.

However, depending on the implementation of the OS kernel or the restriction of underlying CPU architecture, it is occasionally required to manage the fragmentation of physical memory to satisfy contiguous memory allocations. For example in Linux, which is prevalently used in mobile systems [4, 26, 33], the size of a kernel stack is 8 KB (two contiguous pages) [22]. A slab page, which is used in a slab allocator for providing kernel-level dynamic memory allocation, can be larger than 4 KB in order to minimize the internal fragmentation inside slabs [9]. In a 32-bit ARM architecture, the size of page global directory (PGD) is 16 KB (four contiguous pages) [30]. In order to satisfy these small but essential contiguous page allocations, the Linux kernel has employed *page clustering* and *memory compaction* [13, 14].

Page clustering places pages with the same movability together in a fixed-size contiguous region, called a *page block*. A movable page, such as a process virtual memory (VM) page and file page, can be migrated and reclaimed on demand since the references associated with the page are managed. On the other hand, an unmovable page, such as a kernel stack and slab page, cannot be migrated because the pointers referencing the page are not managed. When unmovable pages are scattered in a contiguous region, the region cannot be used for a contiguous memory allocation until the unmovable pages are explicitly deallocated. To avoid this fragmentation problem, physical memory is divided into large blocks (i.e., page blocks), which are assigned a type movable or unmovable. Then, pages are allocated in the page blocks based on their movability. Since the movability can be readily determined at allocation time, the physical memory allocator can control the placement of the two types of pages. This policy can be overridden temporarily when little free memory remains. In this case, an unmovable page can be allocated within a movable page block and vice versa.

Memory compaction is applied with the page clustering to create a contiguous free space on demand. Although the page clustering enables to group pages with their movability, the grouping does not necessarily mean a contiguous region is available; hence, the pages within a page block can be scattered, thereby fragmenting the page block. Accordingly, when a contiguous allocation cannot be satisfied from the given free memory, the kernel actively migrates movable pages to secure a contiguous free space. In more detail, the movable pages in low addresses are migrated to free pages in high addresses. When the compaction completes, the physical memory is compacted, with free pages in low addresses and movable pages in high addresses.

The *unusable index* [12, 13] quantifies the degree of memory fragmentation. It specifies the fraction of free memory that is unusable for the allocation of a specific size. Assume a memory allocator that manages an address space in power of two pages (e.g., 2^1 or order-1 to refer to an 8 KB region). Then, the unusable index of free space for 2^x pages is defined as

$$UI(x) = 1 - \frac{\sum_{i=x}^{n} 2^i \times f_i}{\sum_{i=0}^{n} 2^i \times f_i} \qquad (1)$$

where f_i denotes the number of free pages of size 2^i, and n denotes the largest allocable order in the system. When the unusable index is 0, all contiguous chunks in free memory can satisfy the allocation of the specified size. When the unusable index is 1, none of the chunks in free memory can satisfy the allocation. This may lead to memory compaction when the amount of free memory is larger than the specified allocation size. If the free memory itself is smaller than the requested size, physical memory is reclaimed first, and (if necessary) memory compaction is conducted [14].

2.2 Memory Management in Mobile Devices

Modern mobile systems, such as smartphones, tablets, or smart TVs, are general-purpose, and can run many applications developed by third-party developers. They are multi-programmed systems but run a limited number of foreground applications at a time (usually one to a few) due to their limited resources and screen size. Some background activities are allowed, but are limited in functionality, such as receiving push notifications.

A number of studies have reported that users tend to interact with many applications with a certain degree of locality in mobile systems [11, 29, 31]. In addition, the burst of each application is usually short (e.g., checking emails or new messages in social network services). In this regard, it is important to minimize the launch time of applications to make the responsiveness of a system high. To this end, many mobile systems adopt application caching in memory [15, 35]. When a user leaves an application, the application is not terminated but paused in memory. When a user accesses a cached application later, the application is quickly resumed, which is much faster than launching the application from scratch.

However, the number of applications that can be cached in memory is limited by the capacity of physical memory. Thus, when free memory becomes below a certain threshold some cached (unused) applications are killed to reclaim memory pages allocated to the applications (e.g., process virtual memory (VM) pages). This is the main role of *low memory killer* in Android. From this characteristic, reclaiming process VM pages by killing a cached application is a primary means of refilling free pages in mobile systems.

```
iobuffer* alloc_buffer(int size)
{
  static int orders[] = {8, 4, 0};
  iobuffer *buffer = kmalloc();
  int size_remaining = size;
  int *max_order = &orders[0];
  while (size_remaining > 0) {
    page = alloc_pages(*max_order);
    if (!page) {
      max_order++;
    } else {
      size_remaining -= 1UL << *max_order;
      buffer->append(page, *max_order);
    }
  }
  return buffer;
}
```

Figure 1. A C code snippet for allocating an I/O buffer in the ION allocator

2.3 Motivation

Modern mobile devices are equipped with many auxiliary hardware devices to enhance system performance and user experience. For instance, GPU accelerates texture rendering on 2D or 3D surfaces, providing high-quality visual experience. A hardware codec accelerates encoding of video feed from a camera, enabling live video chatting over networks. An audio decoder enables users to listen to music at low power consumption.

These hardware accelerators require large I/O buffers, each of which is tens of MBs, since the accelerators usually manipulate high-resolution multimedia data. When such devices do not support scatter/gather DMA, memory reservation is one of common solutions to allocate such physically contiguous large buffers against the physical memory fragmentation [1, 19]. Meanwhile, with the adoption of IOMMU, such large I/O buffers can be allocated dynamically. Physically scattered pages (e.g., 4 KB page) can be contiguous in an I/O address space because IOMMU provides on-demand mappings between physical and I/O addresses. Consequently, the need for large contiguous memory has been mostly diminished.

However, IOMMU does not completely eliminate the need for contiguous memory. Comprising an I/O buffer with only 4 KB pages is inefficient in terms of IOMMU mapping and unmapping costs [7, 36] and IOTLB overhead [2], since it requires many I/O page table updates and causes high IOTLB miss ratios by reducing IOTLB reach. To avoid the overhead, many device drivers attempt to allocate I/O buffers with large pages that IOMMU supports. For example in Linux, Figure 1 shows a C code snippet for a buffer allocation in the ION memory allocator, which is a specialized memory allocator for device drivers such as hardware accelerators [38]. In this example, IOMMU supports three types of pages (1 MB, 64 KB, and 4 KB pages) as in the `orders` array. The allocator tries to allocate the largest page first. When the allocation fails, the allocator falls back and attempts to allocate smaller pages by incrementing the `max_order` pointer. This operation repeats until the required

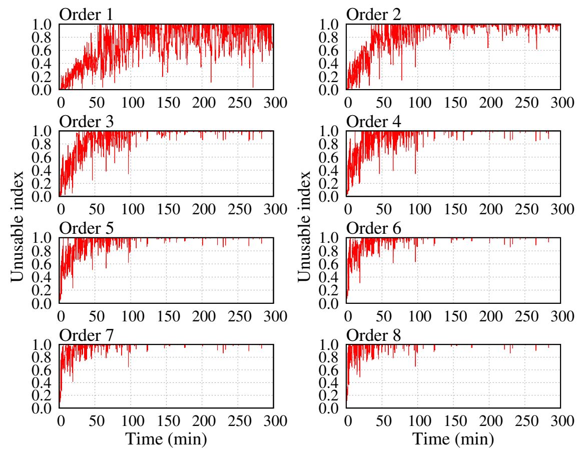


Figure 2. The change in the unusable index over time

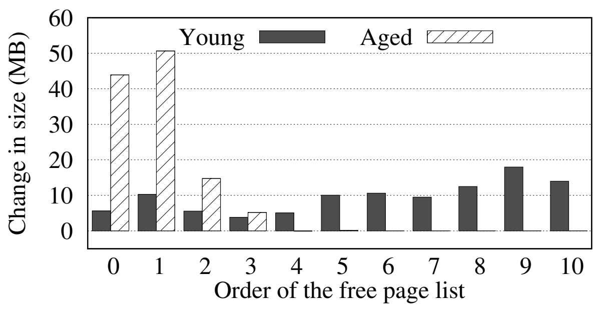


Figure 3. The change in the number of free pages in each order in the buddy allocator upon an application termination

amount of memory is allocated. For instance, a 15 MB frame buffer, which is one of representative I/O buffers in our test bed, can be composed of fifteen 1 MB pages, or, at worst, 3,840 4 KB pages.

In practice, due to severely fragmented physical memory, it is difficult to allocate I/O buffers with large pages. Figure 2 shows the change in the unusable index for sizes from 2^1 to 2^8 pages (from the smallest to largest allocatable pages) over time while we ran a series of application launches that follows realistic user behavior; the details are described in Section 6. After 100 minutes, the unusable indices for large pages are saturated to 1, indicating that the desirable pages are hardly available. Consequently, I/O buffers are badly fragmented so that over half of the pages comprising the I/O buffers are 4 KB pages; see Section 6 for details. This result indicates that the memory compaction and page clustering are unsatisfactory to satisfy large volume of large page allocations for I/O buffers.

The severe fragmentation of physical memory is primarily caused by the memory usage policy in mobile systems. Due to application caching, the utilization of physical memory is kept high. While free memory is low, allocating small pages having different deallocation times exacerbates the fragmentation of free space [25]. Although a bulk of process VM pages are reclaimed when a cached process is killed, however, the freed pages do not help to ease severe fragmentation. Figure 3 presents the change in free pages in the buddy system allocator after killing a cached application when the physical memory is young (after 10 application launches) and aged (after 1,000 application launches), respectively. The application has approximately 110 MB of pages in both cases. In the young case, the amount of high-order free pages is significantly increased after killing the application. In the aged case, over half of the freed pages are order-0 (4 KB), order-1 (8 KB), or order-1 pages produced by coalescing order-0 buddies. Free pages larger than order-3 are not reclaimed or produced by coalescing smaller pages. This result implies that bulk memory reclamation by killing cached applications does not relax memory fragmentation, since the pages allocated to an application are already fragmented.

3. Memory Usage Characterization

It is important to understand the usage of physical memory to effectively control the physical memory fragmentation. To this end, we analyzed the memory usage of Android, which is the most popular platform for mobile systems. Android runs on top of the Android kernel, which is an extended version of the Linux kernel to support Android-specific functionalities such as binder and low memory killer. Its physical memory management is the same as that of the Linux kernel.

The ways of using physical memory are roughly fourfold: process VMs, OS file caches, device I/O buffer, and kernel memory. These commonly interact with a buddy system allocator, which is the core memory allocator in the Linux kernel at the granularity of power of two pages. Memory allocation is explicitly initiated by each component. For example, when a process wants to expand its stack, a page fault handler requests allocation of physical pages and maps them to the desired address space. When a process wants to play a video stream, the device driver of a hardware codec allocates I/O buffers.

Memory deallocation, however, is conducted in two ways: (1) explicitly by each component and (2) implicitly by the page replacement policy in OS. In the former case, a component returns unnecessary pages to the buddy system allocator. An example includes the deallocation of I/O buffers after finishing the function of an I/O device. The latter case is triggered when free memory is low. A page reclamation thread (e.g., *kswapd* of the Linux kernel) continues to reclaim pages until free memory becomes sufficient. Without swap storage, cache (or file) pages are the only targets of this page-level reclamation. Instead, the low memory killer reclaims process VM pages when free memory is below certain thresholds as described in Section 2.2.

In addition to the basic memory usage characteristics, the following details the specific characteristics of each memory type.

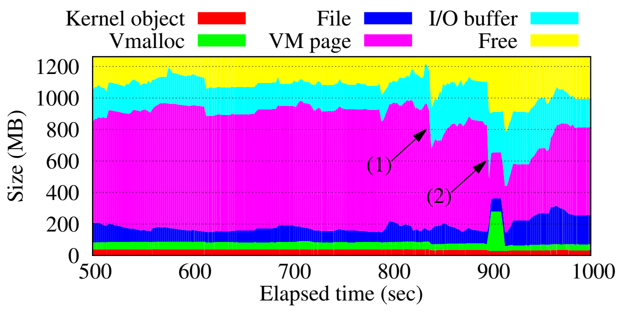


Figure 4. A 500 second-snapshot of memory usage in a Google Nexus 10 tablet running the benchmark workload.

Type	Alloc. granularity	Dealloc. unit	Usage portion	Movable?
Process VM page	One page	Mostly per-process	Largest	Yes
File page	One page	Per-page	Moderate	Yes
I/O buffer	1, 16 or 64 pages	Per-buffer	Second largest	No
Kernel memory excluding vmalloc objects	1–8 pages	Per-object	Minimal	No
Vmalloc objects	One page	Per-object	Fluctuating	No

Table 1. Summary of memory usage in mobile systems

- **Process VM pages.** Process VM pages, or VM pages, comprise stacks and heaps of processes. In mobile systems, VM pages account for the largest fraction in memory usage, as shown in Figure 4, due to the application caching. VM pages are allocated at the 4 KB page granularity via demand paging. Since their allocation granularity is small and allocation times spread over time, VM pages for a process are usually scattered one a physical address space. Although the allocation times of VM pages for a process are different, most of them share the same deallocation time since they are reclaimed in a per-process basis by the low memory killer. In Figure 4, the sudden increases in free pages pointed by the two arrows indicate terminations of cached applications. VM pages are movable since the references (e.g., page table entries) to them are tracked and managed by the kernel.
- **File pages.** Modern OSes usually cache file data in memory to accelerate accesses to slow secondary storage. These file pages are usually allocated individually on a file cache miss at the page granularity. Unlike VM pages, file pages are deallocated at a page granularity via the page replacement policy. Since the page replacement scheme reclaims file pages prior to VM pages, file pages account for a moderate fraction of total memory usage, as shown in Figure 4. File pages are movable, although some of them become temporarily unmovable by file system journaling.
- **I/O buffers.** As described in Section 2.3, an I/O buffer can be composed of various sizes of pages depending on the supported pages in IOMMU (4 KB, 64 KB and 1 MB in our test bed). An I/O buffer allocator allocates an I/O buffer by allocating pages repeatedly from the buddy system allocator until a desired size is allocated. The I/O buffer allocator tries to allocate as many large pages as possible. When a large page is not available, the allocator falls back to smaller page allocations. Hence, when the memory is not fragmented, an I/O buffer can be composed of a few large pages. If the memory is highly fragmented, an I/O buffer is composed of many small pages. The deallocation of an I/O buffer returns all pages comprising the buffer, so that the pages in an I/O buffer have the same lifetime. I/O buffers account for the second largest fraction in total memory usage, as shown in Figure 4. Finally, pages comprising I/O buffers are unmovable.
- **Kernel memory.** The kernel requires memory for several types of usages including kernel stacks, page tables, and kernel objects. The kernel objects are divided into two classes: physically contiguous objects (*kmalloc* or slab objects) and virtually contiguous objects (*vmalloc*). The allocation granularity varies depending on the type of usage. For example, a kernel stack is 8 KB, PGD is 16 KB, and a slab is between 4 KB and 32 KB. The size of virtually contiguous objects varies but the allocation granularity is 4 KB, since scattered pages can be mapped to be contiguous in the kernel virtual address space. As shown in Figure 4, except for the virtually contiguous objects, the amount of kernel memory (denoted as *kernel object*) is the smallest among the memory usages and stable over time. The amount of virtually contiguous objects (denoted as *vmalloc*) is usually small but fluctuates occasionally. In the figure, the sudden increase of virtually contiguous objects is caused by a camera application, which requires a large volume of memory in the kernel space.

Table 1 summarizes allocation granularity, deallocation unit, fraction in total memory and movability of each memory type. Based on the characteristics, the following section describes how each memory type is managed to control the fragmentation of physical memory.

4. Region-based Physical Memory Management

According to the characterization of memory usage in mobile systems, we found that a large number of free pages are refilled when a cached application is killed. This implies that if the freed pages are physically contiguous, subsequent contiguous memory allocations can be easily satisfied on the freed pages without incurring additional operations, such as memory compaction. To realize the idea, we propose a region-based memory allocator that tries to group pages having the same deallocation time in a contiguous region. Fig-

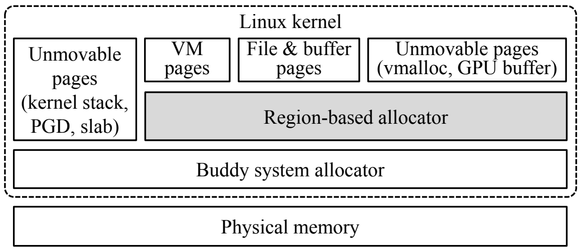


Figure 5. The memory management hierarchy of the proposed scheme

ure 5 depicts the overall memory management hierarchy of the proposed scheme. The rest of this section describes how the region-based memory allocator realizes the idea with the various types of memory usage.

4.1 Process Memory Management

Our scheme focuses primarily on the placement of process VM pages in a contiguous memory space. To this end, we divide the physical memory into fixed-size regions, and dynamically allocate regions to a process. The number of regions allocated to a process depends on the memory demand of the process. All regions belonging to a process are reclaimed when the process is terminated.

To group regions belonging to the same process, we define a *region domain*, or domain for short. When a process is created, a domain is created and the process is associated with the domain. Process VM page allocations and deallocations are fulfilled within the domain. When the owner process requests for a page, a free page in the domain is provided. If the domain has no region with any free pages, a region is allocated from the physical memory allocator (i.e., buddy system allocator) and associated with the domain. When every page in a region becomes free, the region is returned to the physical memory allocator. The free pages in a region are managed similar to the *quick fit* [16], which enables fast allocation and deallocation of pages. Figure 6 illustrates an example of our process memory management. There are three region domains (three processes) and each domain has two or three regions allocated. The small squares in the regions indicate pages that are allocated (shaded) or free (blank).

Since free pages in a region domain can only be used by the owner process, these unused free pages can be considered as internally fragmented from the perspective of the buddy system allocator. However, such free pages in a region domain are different from the traditional internal fragmentation, which cannot be utilized for any purposes, in that they can be used by the owner. Nevertheless, minimizing the amount of such unused pages is important in terms of memory utilization.

4.1.1 Region Domain Compaction

If a process allocates and deallocates pages monotonically, its region domain has only one partially used region as Domain A in Figure 6. In practice, however, the allocation and deallocation are not monotonic, and a process allocates and deallocates pages dynamically according to its memory usage and demand. Thus, a region domain can have multiple partially used regions as Domain B and C in Figure 6. The problem is that the amount of unused pages can be larger than the size of a region as Domain C is. Since the allocator cannot control the locations of page deallocations, these holes should be managed properly to make regions compact.

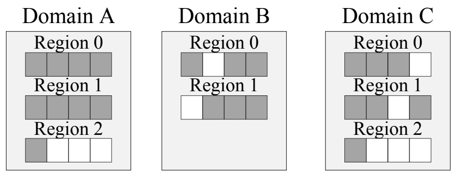


Figure 6. Region domains

Since VM pages are movable, we can easily compact regions having holes; allocated pages in the region with the lowest utilization are migrated to the region that is not full but has the highest utilization. However, the timing of this compaction is an important issue. If page allocations are followed by compaction, page migrations during the compaction are unnecessary. We avoid the unnecessary compaction by compacting a domain when its owner process goes to the background, since the process in the background is paused and does not incur memory activities in general [17].

The evaluation results indicate that the region domain compaction requires fewer CPU cycles than the memory compaction of the kernel. The primary difference is that the kernel memory compaction scans the physical address space to locate the pages to migrate and proper free pages. This scanning cost is significant since each scan needs to test the metadata of each page frame. In contrast, the region domain compaction can directly locate pages to move out and proper free pages to move in by traversing a linked list.

4.1.2 Region Domain Sharing

Although the region domain compaction can reduce the amount of unused pages within a domain, the amount of system-wide unused pages can be significant if there are many region domains. When a region domain is compacted, the domain has only one partially used region while the other regions are full. The expected amount of unused pages in a single domain is $1/2r$ where r is the size of a region. If there exist n region domains in a system, the amount of system-wide unused pages is $n \times 1/2r$. In practice, our test bed has 55 processes running at boot-up, thereby creating 55 region domains and the amount of unused pages is approximately 28 MB when a region is 1 MB. These unused pages cause memory under-utilization and trigger unnecessary page reclamation due to a lack of free memory in the physical memory allocator. Therefore, the system-wide unused pages should be managed.

To reduce the system-wide unused pages, we found that many of processes in a system are persistent ones supporting the application framework. For instance, system_server, media_server, and zygote are in charge of managing the applications, providing media operations, and spawning a new application, respectively. The low memory killer never kills these persistent system processes; hence, their VM pages have the same lifetime. For this reason, our first solution is to create a *shared region domain* to which persistent processes belong.[1]

Another characteristic to exploit from mobile systems is that some processes provide background activities. The application framework supports a background activity as the name of service. For example, when a Facebook application is in the background, it can receive push notifications as a background service. Since the application framework prefer service applications to normal applications, service applications are given higher priority than normal applications. Accordingly, the low memory killer tends to kill normal applications rather than service applications. Even if a service application is killed, the application is likely to be spawned again to handle a new incoming event. Hence, service applications are likely to outlive normal applications. In this regard, we make service applications share a domain called a *service domain*.

In the test bed, there are 25 persistent system processes and 21 service processes out of total 55 processes. By applying the domain sharing, the number of region domains is reduced from 55 to 11 (one shared domain, one service domain, and nine domains for normal applications) at boot-up. Consequently, the amount of system-wide unused pages is reduced to one fifth.

4.2 File Cache Management

Since file (or cache) pages are allocated and deallocated at the page granularity, if the placement of them is not controlled, file pages can also be scattered, thereby fragmenting physical memory. In order to curb the fragmentation of file pages, we make a single region domain, *cache domain*, which hosts all file pages. The rationale behind this is twofold. First, file pages are movable so that the domain can be compacted at any time if the domain has many holes. Second, since file pages store cache data of secondary storage, the domain compaction can have an additional option of discarding cached data. Modern mobile systems are equipped with high-performance secondary storage, such as high-class secure digital (SD) cards, embedded multimedia card (eMMC) or universal flash storage (UFS), which shows much higher performance than traditional disks. Thus, discarding file pages does not cause a high performance penalty while increasing the flexibility of the domain compaction.

[1] Shared pages between processes are allocated in the shared domain, since the shared pages are still allocated in memory when one of the processes is killed. The amount of shared pages is approximately 500 KB, which negligibly affects the blowup of the shared domain.

Therefore, the domain compaction for the cache domain is specialized as follows; the domain compaction is triggered by the page reclamation process in the kernel when it attempts to reclaim file pages in a system. The process selects a file page as a victim according to the page replacement algorithm. For the victim file page, the pages belonging to the same region are also selected. Among them, *active* pages, which are deemed in the current working set, are migrated to other regions. Dirty pages are also migrated to other regions instead of being written back to secondary storage to avoid long write-back I/O delay. Finally, the rest of the pages (clean *inactive* pages) are discarded, and the region is returned to the physical memory allocator.

In OS buffer cache, block device pages (a.k.a buffer pages in the Linux kernel) are isolated from the cache domain and managed in a different manner since these pages are sometimes neither movable nor discardable for a long time (up to 5 seconds). The buffer pages store file system metadata and journal data [18]. When a buffer page has a dirty data of a committing transaction in file system journaling, the page is neither movable nor discardable until the file system journal is finally written to secondary storage. Moreover, if such pages are scattered in the cache domain, the regions containing them cannot be reclaimed, thereby making many unused pages in the cache domain.

To alleviate this problem, we store buffer pages in a special domain, *buffer domain*. These pages account for only a few MBs throughout the evaluation, and the footprint of the buffer domain is bounded. In contrast to the cache domain, the domain compaction of the buffer domain is performed only using page migration.

4.3 Unmovable Page Management

In spite of the effort to manage the majority of the memory, we found that unmovable kernel pages can aggravate the fragmentation severely though they occupy only a small fraction of the memory. Recall that the kernel pages include I/O buffers, kernel stacks, process page tables, and slabs.

Interestingly, we already benefit from the page clustering policy of the kernel [13, 14]. The page clustering places unmovable pages in a limited set of page blocks. In addition, the amount of such pages is small and stable, as presented as *kernel object* in Figure 4. For this reason, we do not actively control unmovable kernel pages in general.

As an exception, we manage vmalloc pages by allocating them from a special region domain, *vmalloc domain*. Vmalloc pages are different from other kernel objects and rather similar to process VM pages in that they are allocated at the page granularity and can be scattered over the physical memory. Thus, we localize their location to prevent them from fragmenting memory.

In our test bed, GPU buffers have the same characteristics as vmalloc pages; GPU buffer pages are allocated at the page granularity and can be scattered on a physical address space.

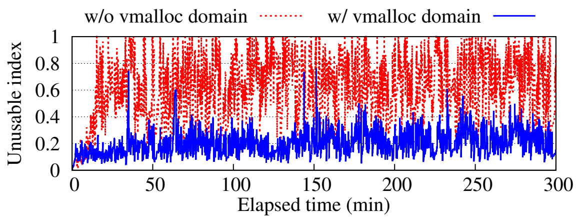


Figure 7. The unusable indices of order-6 pages with and without using the vmalloc domain

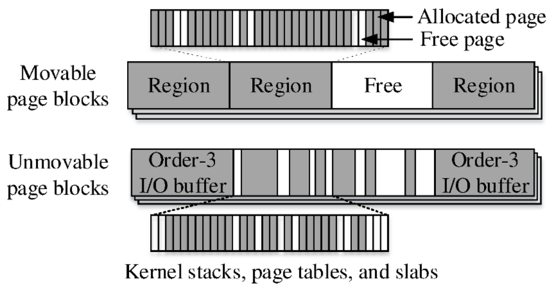


Figure 8. An example of the memory usage in the proposed scheme. The size of a page block is assumed at 512 KB, and the size of a region is assumed at 128 KB.

Therefore, pages for GPU buffers are forced to be allocated from the vmalloc domain.

Except for the GPU buffers, we do not explicitly manage the pages for I/O buffers. When physical memory is not fragmented, an I/O buffer can be composed of large contiguous pages rather than small ones. These contiguous pages are reclaimed intact when the I/O buffer is deallocated. Hence, if we prevent physical memory fragmentation from the very first, the allocation and deallocation of I/O buffers will not aggravate memory fragmentation.

Figure 7 shows the effect of the vmalloc domain. When the vmalloc domain is not used, free memory is highly fragmented so that the unusable index frequently reaches to 1; hence no free region is available in the free memory. However, when the vmalloc domain is employed, the unusable index is kept low, primarily between 0.1 and 0.4.

Figure 8 shows an example of the physical memory usage in our scheme. The regions for VM pages and file pages are allocated from movable page blocks. Unmovable page blocks provide unmovable page allocations for I/O buffers, kernel stacks, PGDs, slabs, and regions for the vmalloc domain, which are not depicted in the figure.

4.4 Memory Reclamation

In the Android kernel, the page reclamation process (i.e., kswapd) initiates memory reclamation when the amount of free memory gets below pre-defined thresholds. This approach offloads time-consuming memory reclamation from the critical path of application execution. During the reclamation, the reclamation process invokes the low memory killer if the amounts of free pages and file pages are below specified thresholds at the same time.

Recall that the unused pages in regions are different from the traditional internal fragmentation in that their owner process can use them. Hence, these unused pages in regions can be considered as a sort of free memory. We adjust the thresholds for checking free memory so that they take the unused pages in regions into account. Our evaluation shows that the modification did not incur any side effects such as an allocation failure due to a lack of free memory in the physical memory allocation.

Finally, the special domains (shared domain, service domain, and buffer domain) are compacted via the memory shrinker that is invoked automatically when free memory is low. Consequently, time-consuming page migrations occur only if those are worthwhile. Recall that the cache domain is compacted by the page reclamation process and the per-process domains are compacted when its corresponding application goes to the background.

5. Implementation

We implemented the proposed scheme in the Android kernel 3.4.5. The region-based allocator is implemented on top of the buddy system allocator, and interacts with the buddy system to allocate or deallocate regions.

Regarding the cache domain, we implemented two cache domains in the system due to the memory zones in the Linux kernel. The kernel partitions physical memory into two memory zones, *normal* and *highmem*, and manages them separately. The majority of our implementation is independent of the zones; however, the page replacement algorithm for file pages works per-zone basis. To cooperate with the page replacement algorithm, we maintain one cache domain for each zone.

Regions are allocated from any of the zones by specifying the `GFP_HIGHMEM` flag, except for the buffer domain. The regions for the buffer domain are allocated from the normal zone as they are assumed to. All region allocations for movable domains are specified with the `GFP_MOVABLE` flag to let the buddy system know the pages in the region are movable.

The Android application framework notifies the kernel of the class of each application via the *proc_fs* interface. By instrumenting the notification path, the kernel is aware of the class of processes, such as a persistent process, service application, and the state changes to the background.

6. Evaluation

6.1 Methodology

We evaluated the proposed scheme on a Google Nexus 10 tablet, which is equipped with a dual-core Cortex-A15 CPU and 2 GB of RAM, and runs Android 4.2.2 (Jelly Bean). We composed an in-house benchmark for use as the workload, which launches Android applications in a given order. The benchmark launches an application by injecting an intent for

Category	Applications	%
Productivity	Adobe Reader, Evernote, Calendar	9.8
News	ESPN Sports Center, CNN, The Weather Channel	4.8
Games	Angry Birds, Subway Surfer, Candy Crush Saga	18.5
Communication	Gmail, Hangout, Facebook, Google+, Flipboard, Feedly	39.9
References	Google Maps	4.8
Entertainment	Movie, Camera, Gallery, Youtube	11.5
Shopping	Play Store	3.9
Browser	Browser	6.8

Table 2. The application composition of the benchmark

the application using the am command-line tool, and waits 15 seconds for the application's launch. The benchmark then launches the home launcher application and waits for another 5 seconds. The benchmark performs the procedure on the applications specified in a workload file. We chose 22 popular Android applications from various categories in Android Play Store, and generated a workload comprised of 1,000 application launches. The workload was carefully generated by referring to LiveLab research [31] to imitate a realistic user behavior. Table 2 summarizes the applications and their proportion in the workload. Note that the 1,000 launches are equivalent to 18.2 days of uptime according to the LiveLab trace.

We attempted to use the stock vanilla kernel as the baseline; however, preliminary evaluations show that the kernel halts in the middle of the benchmark and cannot proceed to complete it. In-depth analysis reveals that a conservative memory system parameter caused the halt. When the kernel requests for an order-1 page for a new kernel stack, the allocation fails because high-order (1 or larger) free pages in the buddy system are deemed *insufficient* though a few high-order pages exist. The level of sufficiency is determined by a parameter, `min_free_order_shift`, and its default value is configured to be too conservative. By relaxing the parameter value from 1 (default) to 2, no kernel halt occurs anymore. We refer to this configuration as VAN.

We evaluated the region-based allocation using varied region sizes from order-5 to order-8 pages, each of which is larger than the second largest IOMMU page (64 KB) and smaller than the largest IOMMU page (1 MB). We will refer to RGN as the configuration that uses an order-N contiguous page as a region. For example, RG5 uses the $2^5 \times 4$ KB $=$ 128 KB regions. Thus, we compare RG5, RG6, RG7 and RG8.

To reduce unpredictability during evaluation, we set the CPU frequency governor to the *performance governor*. Other parameters are set to default unless stated.

6.2 Unusable Index

First, we analyze the influence of the region-based allocation on memory fragmentation. While running the benchmark, we collected the number of free pages in the buddy system

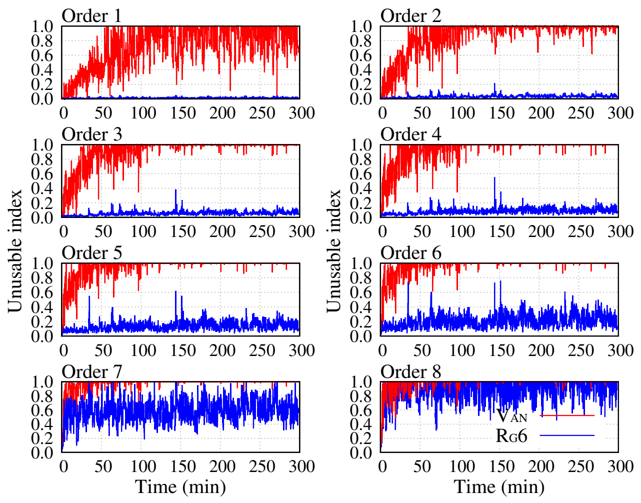


Figure 9. The change in the unusable indices of VAN and RG6 over time

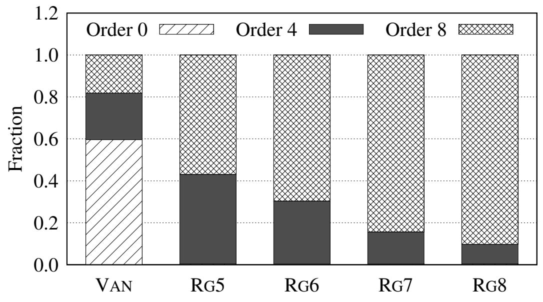


Figure 10. Breakdown of the allocation unit for I/O buffers

allocator, and calculated the unusable index for each order using Equation 1. Figure 9 compares the unusable indices of VAN and RG6 over time. We omitted the results of other region sizes for brevity.

As described in Section 2, we observed rapid increases of unusable indices for VAN, which imply heavy memory fragmentation in all sizes. In contrast, for RG6, the unusable indices of pages smaller than order-6 are limited to low during the entire benchmark execution, because an application termination yields many order-6 pages to the buddy system allocator, thereby lowering the unusable indices for the pages smaller than or equal to an order-6 page. As the proposed approach does not guarantee the contiguity of areas larger than the region, the unusable indices for order-7 and order-8 are high. However, as the memory is allocated/deallocated at a large granularity, the fragmentation indices are maintained lower than those of VAN.

6.3 I/O buffer

We analyzed the page composition of I/O buffers. As explained in Section 2, the ION allocator first attempts to allocate high-order pages from the buddy system allocator and then falls back to lower-order pages. We counted the occurrence of each allocation unit, and presented the break-

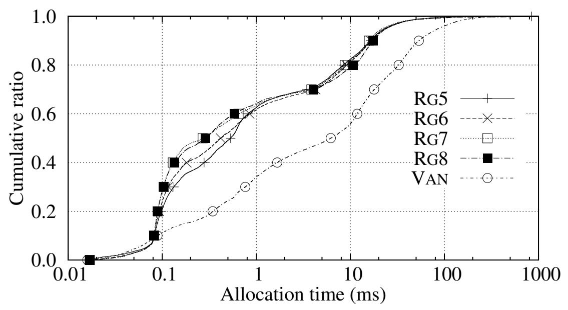


Figure 11. Allocation time for I/O buffers

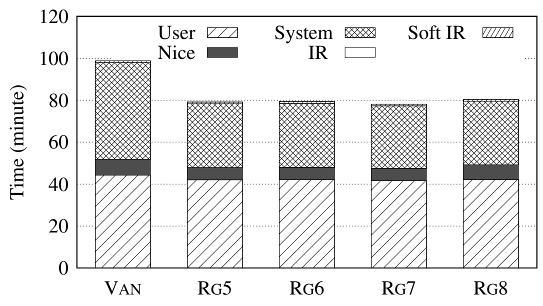


Figure 13. Breakdown of CPU times

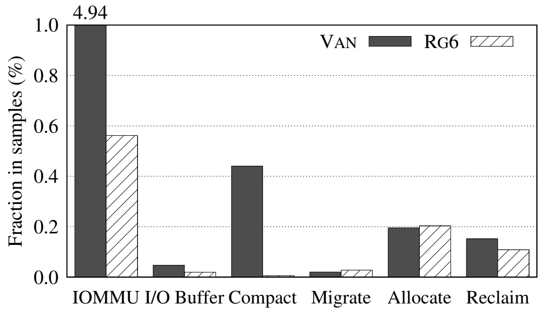


Figure 14. Breakdown of CPU cycles

down of buffer composition sizes in Figure 10. We can verify that I/O buffers are heavily fragmented in VAN but not in the region-based allocation. In VAN, approximately 60% of the allocations are served with order-0 pages, whereas RGNs have negligible fractions for the size, implying that the I/O buffer allocator in VAN has to allocate many small pages, whereas the region-based allocation can accomplish the buffer allocation by using large pages.

In addition, we observed that the fraction of higher-order allocation increases with the size of the region; this is due to the lower memory fragmentation in the high-order pages for larger region size.

To further investigate the effect of page allocation for different configurations, we collected the allocation time for I/O buffers while running the workload. Each configuration allocated approximately 147 GB of I/O buffer over 13,300 allocations. Figure 11 illustrates the allocation times; note that the x-axis is in log-scale.

The allocation time for the region-based allocation is reduced significantly when compared to VAN. Specifically, the medians of the allocation times are less than a millisecond; 0.528, 0.423, 0.265, and 0.280 milliseconds for RG5, RG6, RG7, and RG8, respectively. The median of VAN is 5.871 milliseconds, which is an order of magnitude longer than that of the region-based allocation. We attribute the slow buffer allocation of VAN to the I/O buffer fragmentation. VAN attempts high-order allocations, which may incur memory compaction. Memory compaction cannot satisfy all required large pages, and consequently, VAN falls back to allocate many small pages. In contrast, the region-based allocation can allocate I/O buffers with a smaller number of large pages, which require fewer occurrences of page allocations.

From these results, we can conclude that the proposed region-based allocation can effectively prevent memory from being fragmented, thereby improving the allocation performance as well as I/O buffer fragmentation.

6.4 System Performance

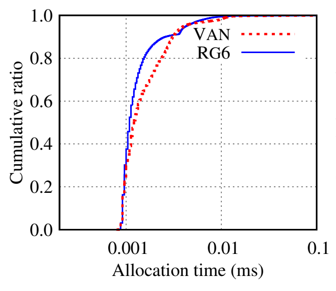
To further understand the implications of the region-based allocation other than I/O buffers, we measure the allocation time of non-I/O buffer pages. Figure 12 summarizes the results. On our evaluation system, the size of the page allocation ranges from order-0 to order-3, excluding the allocation for I/O buffers. In addition, we only present the result from RG6 for brevity, as the configurations with other region sizes exhibit in a similar manner.

For the order-0 page allocation, which dominates the page allocations, we observe a slightly improved performance in RG6 when compared to VAN. We can see two peaks from the distribution for RG6. We attribute the first peak to the allocation from the region, which is handled at a low cost, whereas the second peak is attributed to the allocation from the underlying buddy system allocator.

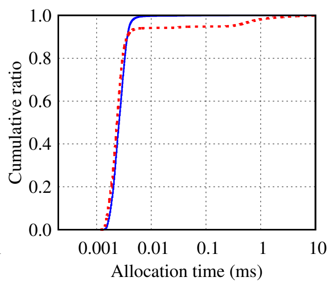
For the higher-order allocations of VAN, we can observe a plateau between 0.01 to 0.1 milliseconds. We believe that the long tails are caused by the case when the buddy system allocator migrates one or more pages for the high-order allocation, thereby delaying the allocation. In contrast, the region-based allocation can handle high-order allocations in a more consistent manner. The majority of the allocations can be served within the region. If this fails to allocate a page from the region, the allocation is served from a newly allocated region. In this case, allocating the region can be quickly served, as the memory fragmentation is maintained at a low level.

From these results, we can conclude that the region-based allocation outperforms the original policy in non-I/O buffer memory allocation.

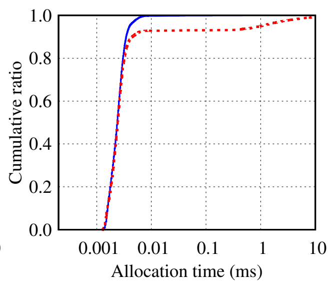
We analyze the CPU usage by breaking down the total elapsed ticks into corresponding modes. As shown in Figure 13, all configurations spend a similar amount of time in each mode with the exception of the system (kernel) mode.



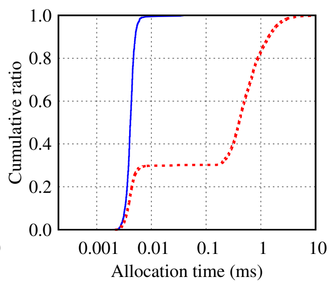
(a) Order 0



(b) Order 1



(c) Order 2



(d) Order 3

Figure 12. Allocation time for non-I/O buffer pages

Configuration	Energy (kJ)	Ratio
VAN	24.4	1
RG6	21.0	0.86

Table 3. Energy consumption while running the workload

Specifically, VAN spends 33.6 to 35.3% more time in the system mode than the region-based configurations, and the time in system mode comprise 46.6% of total CPU time. To further analyze the reasons for the time increase in the system mode, we sampled the current function name after every 50,000-th CPU cycle using OProfile [27], and then classified the samples into categories according to their function. We then aggregated the number of samples for each category. Figure 14 summarizes the fraction of relevant categories among the total samples. We omit the samples that are irrelevant to our scheme.

As shown, a large fraction of the samples in VAN come from manipulating I/O buffers, including mapping and unmapping, which is listed as *IOMMU*. Specifically, VAN spends 4.94% of total time in IOMMU-related operations, the sum of which is 8.76× longer than that of RG6. The ratio of allocation units for I/O buffers summarized in Figure 10 explains the result, in that the I/O buffer fragmentation increases the CPU cycles to manipulate IOMMU. This also explains the slight increase in CPU cycles for I/O buffer allocation/deallocation listed as *I/O Buffer*.

The other notable difference originates from *Compact*, which includes the functions relevant to page compaction—the fraction of compaction is significant for VAN and is negligible for RG6, a difference originating from the frequent failure of high-order allocations that trigger costly page compaction.

To quantify the effect of the region-based allocation in terms of energy consumption, we have attempted measure energy consumption using the built-in battery monitoring unit. Unfortunately, we cannot run the entire workload using the battery, because the condition of the battery is not sufficient to complete the entire workload. Instead of running the entire workload, we split the workload into two parts. The first is comprised of the former half of the workload, and is used for aging the memory subsystem while the power is supplied from an external power source. Upon completion of this part, we change the power source to the internal battery and run the second part of the workload. During the second part, we measured current (in mA) and voltage (in mV) from the battery each second. We then obtained the energy consumption by aggregating the products of the current and the voltage. Table 3 shows the energy consumption of VAN and RG6 in joules. We can verify that the region-based allocation reduces the energy consumption by 16.0% in practice, which is promising for mobile systems.

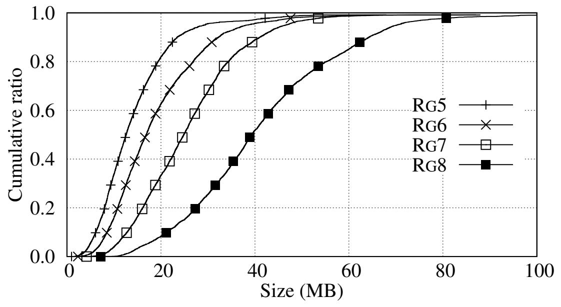


Figure 15. Unused pages in regions

Configuration	VAN	RG5	RG6	RG7	RG8
Fraction	0.408	0.439	0.441	0.437	0.440

Table 4. Fraction of the application resume

6.5 Unused Pages in Regions

To estimate the overhead of the region-based allocation, we measured the amount of unused pages in regions at an interval while running the benchmark, and depict the values with cumulative distribution in Figure 15. When the region is small, we can limit the unused pages to a low level. Specifically, the unused pages within order-6 page regions occupy less than 20 MB during 80% of the entire lifetime, and increase along with the size of the region. When the region is an order-8 page, the unused pages take up approximately 40 MB of the memory on average. Recall that the unused pages in regions are not actually wasted but can be utilized when the owner application becomes active.

The implications of these unused pages can be estimated using memory utilization metric. In mobile systems, the fraction of application launches served by resuming cached applications is used as a practical metric. To this end, we measured the types of application launches while running the benchmark; Table 4 shows the results. Note that the higher ratio of the application resume implies better memory utilization in practice.

We can observe that the region-based configurations have higher rates of application resumes than VAN. This implies that the unused pages in regions do not harm the memory utilization; the reason for the increase is still being investigated.

7. Related Work

Memory fragmentation and its implications have been discussed in the literature for decades. A number of studies explore memory fragmentation in dynamic memory allocation [5, 6, 8, 10, 16, 20, 21, 24, 32, 34]. In particular, Johnstone and Wilson [20] argue that memory fragmentation is highly affected by the allocation policy via comparing the degree of memory fragmentation across various combinations of allocation policies. Other research considers memory fragmentation to bound the time for the garbage collection of heap-allocated objects [5, 10]. The previous work differs from our study in that our scheme focuses on memory fragmentation at the page-level.

It is widely known that grouping memory objects with similar characteristics can improve both fragmentation and performance in memory allocators. In particular, considering the lifetime of objects is one of the most popular and effective aspects from many studies [6, 16, 21, 24, 32, 34]. Lieberman [21] utilizes the lifetimes of objects to improve the realtimeness of a garbage collector. Hanson [16] attempts to segregate short-lived objects from long-lived objects. Barrett and Zorn [6] further enhance the Hanson's approach by predicting lifetimes from the behavior of previous runs of the program. Generational garbage collectors [24, 32, 34] utilize the high mortality of young objects. However, the previous work also focuses on fragmentation within the heap and concerns the objects within a single process rather than system-wide perspectives.

Page-level memory fragmentation has received less attention than the heap due to the MMU and the scatter-gather DMA. Gorman et al. point out that memory fragmentation is getting one of the most concerning issues for the Linux memory manager [12] and introduce page clustering [13, 14]. Page clustering classifies pages in terms of their movability at the allocation time, and groups pages with the same movability together. Their proposals are considered practical, and are hence adopted as the primary countermeasure for memory fragmentation in the Linux; however, page clustering focuses primarily on ensuring the physical contiguity of small-order pages, and is inefficient in dealing with high-order pages such as I/O buffers. Yeoh [37] proposes to allocate memory to processes in variable sized chunks to prevent memory fragmentation from building up over time, though he does not fully develop the idea and provides a limited evaluation on an experimental operating system. Navarro et al. [25] propose a reservation-based scheme to support superpages, and claim that fragmentation must be controlled to sustain the benefits of superpages; however, their approach attempts to restore fragmented memory rather than to prevent the fragmentation proactively, thereby incurring management overheads.

To accommodate the demand for physically contiguous memory, modern mobile systems usually provide APIs for the dynamic allocation/deallocation of physically contiguous memory [1, 38]. A number of studies attempt to utilize the memory reserved for peripheral devices when the owner device is not in operation [18, 19, 28]. In particular, CMA [28] allocates migratable pages from the reserved memory, and migrates the pages to other locations when the owner device is about to use the reserved memory. DaaC and Rental Memory [18, 19] minimize the cost of migrating pages by allocating easily reclaimable pages such as file pages. This work differs from our work in that they assume the I/O buffers are statically allocated with physically contiguous pages and are free from memory fragmentation.

8. Conclusion

This paper introduces a region-based memory management scheme that exploits the per-process memory reclamation in mobile systems to ease the fragmentation of physical memory. The key idea is to group process virtual memory pages with the same deallocation time in contiguous fixed-size regions so that the termination of a process yields a set of contiguous regions. The reclaimed contiguous regions can be used to satisfy subsequent I/O buffer allocation, thereby reducing the overhead associated with I/O buffer allocation/-management and memory compaction. The evaluation results show that the overhead reduction is exhibited as reduced CPU usage and energy consumption.

Acknowledgments

This work was supported partly by the National Research Foundation of Korea (NRF) grant funded by the Korea Government (MSIP) (No. 2013R1A2A1A01016441) and by the Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Education (NRF-2014R1A1A2054658). Jinkyu Jeong is the corresponding author.

References

[1] Android kernel features. URL `http://elinux.org/Android_Kernel_Features`.

[2] N. Amit, M. Ben-Yehuda, and B.-A. Yassour. Iommu: Strategies for mitigating the iotlb bottleneck. In *Proceedings of*

the 2010 International Conference on Computer Architecture (ISCA '10), pages 256–274, 2012.

[3] Android. Managing the activity lifecycle, October 2013. URL `http://developer.android.com/training/basics/activity-lifecycle/index.html`.

[4] Android, January 2015. URL `http://www.android.com`.

[5] D. F. Bacon, P. Cheng, and V. Rajan. Controlling fragmentation and space consumption in the Metronome, a real-time garbage collector for Java. In *Proceedings of the 2003 Conference on Language, Compilers, and Tools for Embedded Systems (LCTES '03)*, 2003.

[6] D. A. Barrett and B. G. Zorn. Using lifetime predictors to improve memory allocation performance. In *Proceedings of the 1993 Programming Language Design and Implementation (PLDI '93)*, pages 187–196, 1993.

[7] M. Ben-Yehuda, J. Xenidis, M. Ostrowski, K. Rister, A. Bruemmer, and L. Van Doorn. The price of safety: Evaluating IOMMU performance. In *Proceedings of the 2007 Ottawa Linux Symposium (OLS '07)*, pages 9–20, July 2007.

[8] E. D. Berger, B. G. Zorn, and K. S. McKinley. Reconsidering custom memory allocation. In *Proceedings of the 2002 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA '02)*, 2002.

[9] J. Bonwick. The slab allocator: An object-caching kernel memory allocator. In *Proceedings of the USENIX Summer 1994 Technical Conference (USTC '94)*, volume 1, 1994.

[10] S. S. Craciunas, C. M. Kirsch, H. Payer, A. Sokolova, H. Stadler, and R. Staudinger. A compacting real-time memory management system. In *Proceedings of the 2008 USENIX Annual Technical Conference (USENIX ATC '08)*, 2008.

[11] H. Falaki, R. Mahajan, S. Kandula, D. Lymberopoulos, R. Govindan, and D. Estrin. Diversity in smartphone usage. In *Proceedings of the 8th International Conference on Mobile Systems, Applications, and Services (MobiSys '10)*, pages 179–194, 2010.

[12] M. Gorman and P. Healy. Measuring the impact of the Linux memory management. In *Proceedings of the 2005 Libre Software Meeting (LSM '05)*, 2005.

[13] M. Gorman and A. Whitcroft. The what, the why and the where to of anti-fragmentation. In *Proceedings of the 2006 Ottawa Linux Symposium (OLS '06)*, pages 369–384, 2006.

[14] M. Gorman and A. Whitcroft. Supporting the allocation of large contiguous regions of memory. In *Proceedings of the 2007 Ottawa Linux Symposium (OLS '07)*, pages 141–152, 2007.

[15] D. Hackborn. Multitasking the Android way, 2010. URL `http://android-developers.blogspot.kr/2010/04/multitasking-android-way.html`.

[16] D. R. Hanson. Fast allocation and deallocation of memory based on object lifetimes. *Software-Practice and Experience*, 20:6–12, 1990.

[17] S. hun Kim, J. Jeong, and J. Lee. Selective memory deduplication for cost efficiency in mobile smart devices. *IEEE Transactions on Consumer Electronics*, 60(2):276–284, May 2014.

[18] J. Jeong, H. Kim, J. Hwang, J. Lee, and S. Maeng. DaaC: device-reserved memory as an eviction-based file cache. In *Proceedings of the 2012 International Conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES '12)*, pages 191–200, 2012.

[19] J. Jeong, H. Kim, J. Hwang, J. Lee, and S. Maeng. Rigorous rental memory management for embedded systems. *ACM Transactions on Embedded Computing Systems*, 12(1s):43:1–43:21, March 2013.

[20] M. S. Johnstone and P. R. Wilson. The memory fragmentation problem: Solved? In *Proceedings of the 1998 International Symposium on Memory Management (ISMM '98)*, pages 26–36, 1998.

[21] H. Lieberman and C. Hewitt. A real-time garbage collector based on the lifetimes of objects. *Communications of the ACM*, 26(6):419–429, June 1983.

[22] R. Love. *Linux Kernel Development*. Addison-Wesley Professional, 3rd edition, 2010. ISBN 9780672329463.

[23] R. Mijat and A. Nightingale. Virtualization is coming to a platform near you, 2011.

[24] R. Morad, M. Hirzel, E. K. Kolodner, and M. Sagiv. Efficient memory management for long-lived objects. Technical Report TR-RC24794, IBM Research, 2009.

[25] J. Navarro, S. Iyer, P. Druschel, and A. Cox. Practical, transparent operating system support for superpages. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI '02)*, 2002.

[26] Open webOS, January 2015. URL `http://www.openwebosproject.org`.

[27] OProfile. Oprofile: A system profiler for Linux. URL `http://oprofile.sourceforge.net/news/`.

[28] Z. Pfeffer. The virtual contiguous memory manager. In *Proceedings of the 2010 Ottawa Linux Symposium (OLS '10)*, pages 225–230, 2010.

[29] A. Rahmati, C. Shepard, C. Tossell, M. Dong, Z. Wang, L. Zhong, and P. Kortum. Tales of 34 iPhone users: How they change and why they are different. *Technical Report TR-2011-0624*, 2011.

[30] D. Seal. *ARM Architecture Reference Manual*. Addison-Wesley Longman Publishing Co., Inc., 2nd edition, 2000. ISBN 0201737191.

[31] C. Shepard, A. Rahmati, C. Tossell, L. Zhong, and P. Kortum. LiveLab: measuring wireless networks and smartphone users in the field. *Performance Evaluation Review*, 38(3):15–20, January 2011.

[32] D. Stefanovic, K. S. McKinley, and J. E. B. Moss. Age-based garbage collection. In *Proceedings of the 1999 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA '99)*, 1999.

[33] The Linux Foundation. Tizen: An open source, standards-based software platform for multiple device categories, January 2015. URL `http://www.tizen.org`.

[34] D. Ungar. Generation scavenging: A non-disruptive high performance storage reclamation algorithm. In *Proceedings of the First ACM SIGSOFT/SIGPLAN Software Engineering*

Symposium on Practical Software Development Environments (SDE '84), pages 157–167, 1984.

[35] T. Yan, D. Chu, D. Ganesan, A. Kansal, and J. Liu. Fast app launching for mobile devices using predictive user context. In *Proceedings of the 10th International Conference on Mobile Systems, Applications, and Services (MobiSys '12)*, pages 113–126, 2012.

[36] B.-A. Yassour, M. Ben-Yehuda, and O. Wasserman. On the DMA mapping problem in direct device assignment. In *Proceedings of the 3rd Annual Haifa Experimental Systems Conference (SYSTOR '10)*, pages 18:1–18:12, 2010.

[37] C. Yeoh. Managing memory in variable sized chunks. In *linux.conf.au*, 2006.

[38] T. M. Zeng. The Android ION memory allocator, February 2012. URL `http://lwn.net/Article/480055`.