Memory Management Scheme to Improve Utilization Efficiency and Provide Fast Contiguous Allocation without a Statically Reserved Area

MYUNGSUN KIM, Seoul National University and Samsung Electronics
JINKYU KOO, HYOJUNG LEE, and JAMES R. GERACI, Samsung Electronics

Fast allocation of large blocks of physically contiguous memory plays a crucial role to boost the performance of multimedia applications in modern memory-constrained portable devices, such as smartphones, tablets, etc. Existing systems have addressed this issue by provisioning a large *statically reserved memory area* (SRA) in which only dedicated applications can allocate pages. However, this in turn degrades the performance of applications that are prohibited to utilize the SRA due to the reduced available memory pool. To overcome this drawback while maintaining the benefits of the SRA, we propose a new memory management scheme that uses a special memory region, called *page-cache-preferred area* (PCPA), in concert with a quick memory reclaiming algorithm. The key of the proposed scheme is to enhance the memory utilization efficiency by enabling to allocate page-cached pages of all applications in the PCPA until predetermined applications require to allocate big chunks of contiguous memory. At this point, clean page-cached pages in the PCPA are rapidly evicted without write-back to a secondary storage. Compared to the SRA scheme, experimental results show that the average launch time of real-world applications and the execution time of I/O-intensive benchmarks are reduced by 9.2% and 24.7%, respectively.

4

Categories and Subject Descriptors: D.4.2 [**Operating Systems**]: Storage Management—*Main memory*

General Terms: Management, Algorithms

Additional Key Words and Phrases: Memory fragmentation, memory management, page cache

ACM Reference Format:
Myungsun Kim, Jinkyu Koo, Hyojung Lee, and James R. Geraci. 2015. Memory management scheme to improve utilization efficiency and provide fast contiguous allocation without a statically reserved area. ACM Trans. Des. Autom. Electron. Syst. 21, 1, Article 4 (November 2015), 23 pages.
DOI: http://dx.doi.org/10.1145/2770871

1. INTRODUCTION

State-of-the-art high-end portable devices, such as smartphones and tablets, are armed with high-resolution cameras and display panels to attract customers, for example, 3840×2160 in *ultra-high-definition* (UHD), 1920×1080 in *high-definition* (HD), etc. Such high-resolution demand has been satisfied by adopting advanced video compression standards (e.g., *high efficiency video coding* (HEVC) [Henot et al. 2013]) and more powerful hardware (e.g., higher CPU clock speed, heterogeneous processor [Wang and Song 2011], etc.). Along with such technologies to enhance the computing capability and efficiency, efficient memory management has long been regarded, and even emphasized in these days, as a key success factor to meet the demand because the higher pixel density spurs increase of multimedia content size.

Authors' addresses: M. Kim (corresponding author), Real-Time Operating Systems Laboratory, Seoul National University, Seoul, Republic of Korea; email: mskim@redwood.snu.ac.kr; J. Koo, H. Lee, J. R. Geraci, DMC R&D, Samsung Electronics, Suwon, Republic of Korea.
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.
© 2015 ACM 1084-4309/2015/11-ART4 $15.00
DOI: http://dx.doi.org/10.1145/2770871

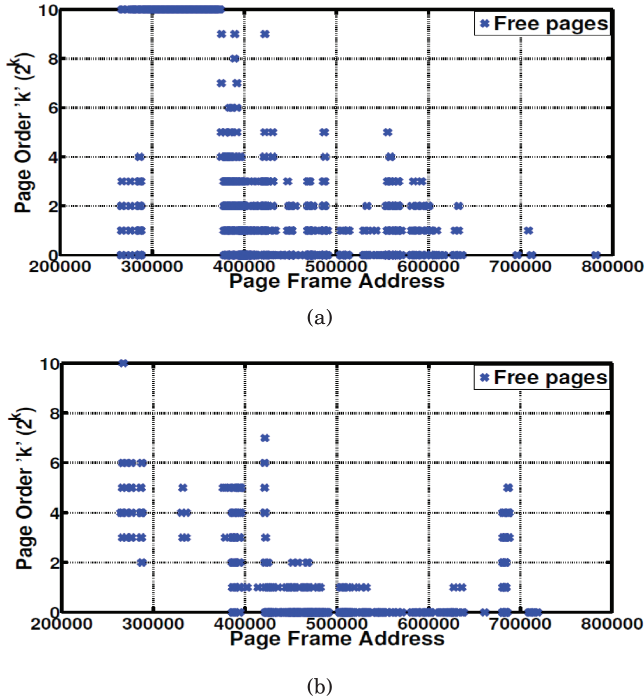


Fig. 1. Comparison of contiguous free page distribution between (a) just after booting and (b) after execution of a number of applications: the x- and y-axis represent the start *page frame number* (PFN) for 2^k contiguous pages and contiguous free page order, that is, k in 2^k pages, respectively.

Direct memory access (DMA) modules are commonly used in state-of-the-art systems to improve the efficiency of data transfers by allowing other hardware subsystems to access main memory without CPU intervention [Yu et al. 2007; Lee et al. 2011; Ammendola et al. 2013]. DMA modules, however, can only access physically contiguous memory blocks [Ammendola et al. 2013], so the speed of providing a chunk of contiguous memory blocks to DMA modules determines the performance of multimedia processing. The main challenge to provide physically contiguous memory blocks to DMA modules is that a physical memory space becomes fragmented into smaller pieces over time and with use. Figure 1 compares snapshots of memory states right after the booting (in Figure 1(a)) against after executing some applications (in Figure 1(b)) in a Galaxy Note 3 smartphone[1]. As shown in the figures, the dots representing the order of contiguous free pages tend to move to lower orders in Figure 1(b), which indicates that the physical memory becomes fragmented as applications keep allocating and freeing physical memory over time. When a system suffers such memory fragmentation [Matias et al. 2011; Udayakumaran et al. 2006], an *operating system* (OS) is not able to provide sufficient contiguous memory chunks to DMA modules, which in turn leads to the delay of the response time of launching and playing multimedia content because an OS requires to invoke a page reclamation routine to secure sufficient contiguous memory space to accommodate the requests.

An *input-output memory management unit* (IOMMU) is used to mitigate the performance degradation caused by the page reclamation [Amit et al. 2011, 2010]. The IOMMU allows to access physically scattered pages in a similar manner to accessing

[1]Refer to Section 4 for the specification of the smartphone used in this experiment.

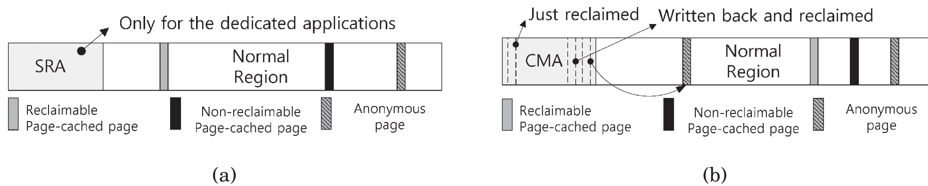


Fig. 2. A memory layout for (a) SRA and (b) CMA schemes.

contiguous pages with the aid of *virtual-to-physical* (VA-to-PA) address translation. To this end, a device driver for IOMMUs needs to manage mapping information required by VA-to-PA address translation for a physically scattered memory pool assigned to the IOMMU. Due to the resource and latency overheads caused by the address translation, using an IOMMU in a system can degrade the performance, which can be further exacerbated as the amount of memory managed by an IOMMU increases.

A *statically reserved area* (SRA) is used to complement the drawback of using an IOMMU. The main idea of the SRA-based physical memory management scheme (hereinafter, we will refer to it as the SRA scheme) is to reserve a large contiguous physical memory chunk at boot time which can be used only for dedicated applications that can be predetermined by designers in design time [Jeong et al. 2013]. The SRA scheme simply segregates physical memory into two regions, that is, an SRA and a normal region, as shown in Figure 2(a). Each application that needs to allocate pages in the SRA should have its own subregion in the SRA region. The SRA scheme is one of the most widely used solutions for allocating contiguous memory because of its instant access to the dedicated physical memory space allocated for predetermined applications. However, it inevitably degrades the memory usage efficiency when we run only applications that are not configured to use the SRA region, because the SRA scheme leaves the SRA region idle when applications that utilize the SRA region are not executed [Jeong et al. 2013, 2012]. This hinders the chance to achieve utmost performance improvement that can be achieved by fully utilizing entire physical memory capacity.

To overcome the SRA scheme's inefficient memory utilization, the *contiguous memory allocator* (CMA) was developed [CMA 2012]. Figure 2(b) shows the memory layout of the CMA scheme. As in the SRA scheme, in the CMA-based scheme, there is a special region (CMA region) and a normal region of memory. Compared to the SRA scheme, the key change in the CMA-based scheme is that any application can write into the CMA region. When applications that are configured to utilize the CMA region demand the CMA region, it is cleared by reclaiming pages (if they are dirty, write-back to the secondary storage is necessary) and moving anonymous pages, if any, into the normal region. The advantage of this system is that it gives all applications access to the full memory space. The drawback is that clearing the CMA region can take some time, and therefore the CMA region's targeted application experiences long delay [Jeong et al. 2012; CMA 2012].

In order to further improve the memory utilization efficiency while maintaining the benefit obtained by the SRA scheme, we propose a novel physical memory management scheme that utilizes a special memory region, named *page-cache-preferred area* (PCPA) region (hereinafter, we will refer to it as the PCPA scheme). The PCPA region is the simple replacement of the SRA region. Compared to the SRA region, the PCPA region is allowed to allocate page-cached pages of all running applications when it is not used by a set of predetermined applications that need to secure big chunks of contiguous anonymous pages (i.e., ones utilizing the SRA region in the SRA scheme). This enables the proposed PCPA scheme to reduce the execution time of memory- and IO-intensive workloads that use abundance of page-cached pages by facilitating additional

physical memory space to allocate more page-cached pages without memory reclamation. Furthermore, page-cached pages are backed by files and normally clean, so they can be reclaimed very quickly compared to dirty page-cached and anonymous pages, as they do not require any write-back to secondary devices [Love 2010, 2004; Jeong et al. 2012]. By benefiting from the hallmark of clean page-cached pages, we can also preserve the benefit of the SRA scheme, that is, provision of big chunks of memory, by quickly reclaiming the PCPA at the moment when predetermined applications require to use it by simply returning clean page-cached pages in the PCPA to the freed page lists managed by the Linux kernel.

The remainder of this article is organized as follows. Section 2 reviews the memory allocation scheme in a state-of-the-art Android-based system. Section 3 elaborates the proposed PCPA solution. Section 4 shows the experimental results. Section 5 reviews the related work, followed by conclusions in Section 6.

2. CONTIGUOUS MEMORY ALLOCATION IN ANDROID SYSTEMS

This section reviews a state-of-the-art solution to efficiently allocate contiguous memory space for running applications using an IOMMU in Android-based mobile devices (in Section 2.1) along with the latency analysis to secure memory space required to launch applications and VA-to-PA address translation (in Section 2.2). Then, we explain the SRA scheme in Android systems (in Section 2.3).

2.1. Contiguous Page Allocation Using an IOMMU in Android Systems

Figure 3 overviews a memory allocation scheme in state-of-the-art Android-based mobile devices (hereinafter, Android system) using an IOMMU and *ion_buffer*. An IOMMU handles the *virtual-to-physical* (VA-to-PA) translation and an *ion_buffer* is an ensemble of physically scattered memory pages connected with a big, singly linked list. An IOMMU is largely composed of a *translation lookaside buffer* (TLB) and a prefetch buffer. The TLB is a hardware cache that can reduce the VA-to-PA translation latency by eliminating the latency taken by accessing page tables in a main memory for the address translation when corresponding entries reside in the TLB. The prefetch buffer holds preloaded VA-to-PA translation information for consecutive virtual addresses so that physically scattered pages can be successively accessed without additional main memory access, as long as they are virtually consecutive [Amit et al. 2010].

The memory granularity managed by an IOMMU is the size of page block. The page table entry consists of a tuple of virtual address, that is, $V(\cdot)$, corresponding physical memory address, that is, $P[V(\cdot)]$, and the page block size. For instance, in Figure 3, with a single 1MB-sized page table entry for $V(1) \rightarrow P[V(1)]$, the virtual address space ranging $V(1) \sim V(1) + 2^8 \times 4\text{KB}$ is mapped to the contiguous physical address space ranging $P[V(1)] \sim P[V(1)+2^8 \times 4\text{KB}]$. Like the modern ARM architecture [ARM 2014], there are four supported page block sizes: 16MB (2^{12} pages), 1MB (2^8 pages), 64KB (2^4 pages), and 4KB (1 page). In general, the page table entry for a 16MB-sized page block is not used because it is practically difficult to allocate consecutive 2^{12}-sized pages at a time, as shown in Figure 1(b).

The units of consecutive page blocks managed by a Linux kernel should be commensurate with those in an IOMMU, that is, 2^8-, 2^4-, and 2^0-sized page blocks, to seamlessly support the IOMMU. The contiguous physical memory allocation in the Linux kernel is governed by a *buddy algorithm* that is devised to mitigate the external fragmentation issue. It uses a Linux kernel function *alloc_pages*(n) which allocates contiguous 2^n-sized physical pages, where, n is an integer value, typically ranging $0 \sim 10$ [Love 2010]. Using the *alloc_pages*(n) function, all the required physical pages for an IOMMU are allocated in the heap. In Android systems, for easy maintenance, an IOMMU connects all the scattered physical pages in *ion_buffer* that consists of three, small linked lists, each

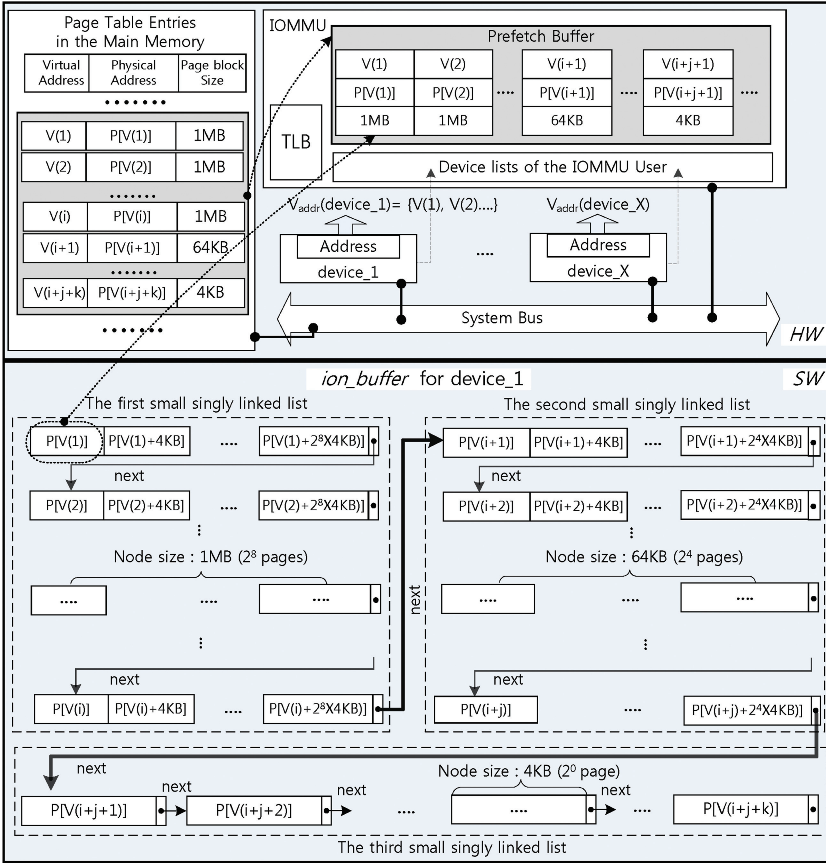


Fig. 3. Overview of the memory allocation scheme with the aid of IOMMU and *ion_buffer* in state-of-the-art Android systems.

of which contains a number of identically sized nodes of 2^8-, 2^4-, and 2^0-sized page blocks, respectively. The first small linked list, whose node size is 2^8 pages, ranges from $P[V(1)]$ to $P[V(i) + 2^8 \times 4\text{KB}]$; the second list whose node size is 2^4 pages is from $P[V(i+1)]$ to $P[V(i+j)+2^4 \times 4\text{KB}]$; the last list is from $P[V(i+j+1)]$ to $P[V(i+j+k)]$ with one page-sized node where i, j, and k represent the number of nodes in each small linked list, respectively. Each node corresponds to a block of contiguous pages allocated by *alloc_pages*(n). Thus the number of *alloc_pages*(n) calls is equal to the number of links (connections by *next* pointer) in each small linked list.

When a device[2] has 2^{total} pages in its *ion_buffer*, we can express the size as:

$$2^{total} = i \times 2^8 + j \times 2^4 + k. \tag{1}$$

Based on this, we can derive the total number of *alloc_pages*(n) calls to construct the *ion_buffer* as follows [Amit et al. 2010; LWNnet 2012]:

$$N_{alloc} = i + j + k. \tag{2}$$

[2]Note that multiple devices can share an IOMMU. In Figure 3, hardware devices, ranging from *device_1* to *device_X*, can use the IOMMU after each device's *ion_buffer* is registered to the IOMMU. For simplicity, only *device_1*'s *ion_buffer* is presented in Figure 3 among multiple IOMMU users.

Table I. Average Execution Time for Allocating 2^8-, 2^4-, and 2^0-Sized Page Blocks

Page block size	4 KB (2^0 pages)	64 KB (2^4 pages)	1 MB (2^8 pages)
normal_alloc	3.4μs	35.4μs	383.7μs
slow_path_alloc	NA	4.6ms	103ms

Note: Test results are based on a Galaxy Note 3.

2.2. Memory Allocation Overhead under Memory Fragmentation

The memory fragmentation affects the performance of running applications when they are launched as well as executed. The launch time is seriously degraded, especially when an application needs a big chunk of contiguous memory (larger than a few MB) as it results in the increase of the number of page allocation calls, as presented in Eq. (2). On the other hand, the performance degradation during the execution is mostly affected by address translation overhead caused by increased TLB misses due to the increase of the number of page table entries. In this section, we first analyze the impact of the increased page allocation calls (in Section 2.2.1), and then explain the impact of the increased TLB miss rate (in Section 2.2.2).

2.2.1. Increase of Allocation Time. There are two kinds of allocation patterns in *alloc_pages*(n) function, that is, *normal_alloc* and *slow_path_alloc*. The *normal_alloc* is invoked when the Linux kernel has a sufficient freed page-list to allocate a requested-sized page block, while the *slow_path_alloc* is invoked when the Linux kernel does not have sufficiently large contiguous physical pages in its freed page-lists. In case of *slow_path_alloc*, it takes longer than the other because the Linux kernel needs to secure enough freed pages to accommodate the size of the requested page block by reclaiming page-cached pages and/or page migrations. Note that a 2^0-sized page is not the case for *slow_path_alloc* as we always have at least more than one 2^0-sized pages. Table I shows the average execution time for allocating 2^8-,2^4-, and 2^0-sized page blocks in the two allocation patterns, which is measured on a state-of-the-art smartphone. In case of *normal_alloc*, the average execution time increases as the page block size increases. Note that the execution time for finding a 2^8-,2^4-, and 2^0-sized page block is same because the procedure of finding a page block in each freed page-list is identical apart from the size [Love 2010]. However, after being detached from the freed page-lists, the found page block needs to pass through a check-up routine identifying that all the pages in it are adequate, without any influence on the Linux kernel.

During the establishment of *ion_buffer*, the Linux kernel first tries to find 2^8-sized page blocks in the freed page-lists, as many as possible. Then, if there are no more 2^8-sized page blocks left, 2^4-sized page blocks in the freed page-lists are searched. Note that, for the case of securing a 2^8-sized page block in *ion_buffer*, *slow_path_alloc* is not executed because it takes too long as shown in Table I. Different from the case of 2^8-, if there are no more 2^4-sized page blocks in the freed page-lists, the *slow_path_alloc* is executed. After the *slow_path_alloc* execution, if no more 2^4 page blocks are found, 2^0-sized pages are added to the *ion_buffer*.

Here we provide an example of how such a memory allocation scheme affects memory allocation time when we establish a 100MB *ion_buffer*. With the values in Table I, if a 100MB *ion_buffer* can be established with only 2^8-sized page blocks, it takes only 38.37ms(=383.7μs $\times$100) while the times are increased to 56.45ms and 87.04ms when only 2^4- and 2^0-sized page blocks are used, respectively, which are situations when memory fragmentation becomes severe. Needles to say, the time when *slow_path_alloc* is executed increases exponentially. We can also find out that the system has small values of i and j when a system suffers memory fragmentation, thereby resulting in spending a lot of time for establishing *ion_buffer* due to the increased N_{alloc}. Due to this

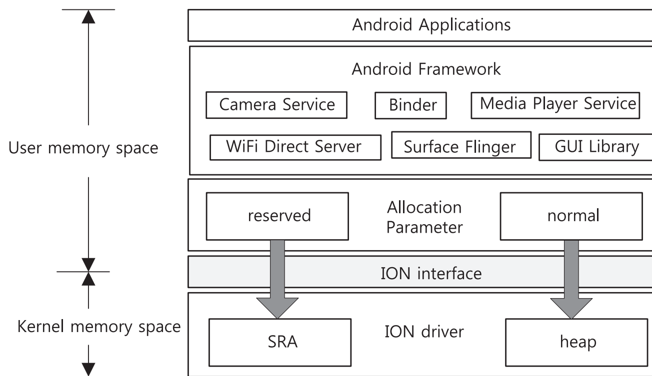


Fig. 4. ION interface in an Android system.

page allocation overhead in using an IOMMU, the responsiveness of an application can be deteriorated.

2.2.2. Increase of TLB Miss Ratio. The number of page table entries of an IOMMU in the main memory is equal to N_{alloc} in Eq. (2). The sizes of TLB and prefetch buffer are limited. Thus, if N_{alloc} is large, many page mapping lists compete to acquire entries in the TLB and the prefetch buffer. In such a case, the TLB miss rate and the number of copying page table entries from the main memory to the prefetch buffer increase, which leads to the increase of the VA-to-PA translation time that affects performance drop when using an IOMMU. Depending on system environment, the time spent for a TLB miss and a prefetch buffer refill can be up to a few tens of microseconds, so if an application uses an IOMMU with large N_{alloc}, it can experience a great deal of replacement overhead. Due to this overhead, the performance of running the application can be degraded.

2.3. SRA Scheme in Android Systems

As explained in Section 2.2, the increased N_{alloc} delays the launching and execution time of applications when using an IOMMU. To reduce the latency, Android system developers pre-allocate a physically contiguous memory space, that is, SRA, then access the memory space with an ION interface that enables to access SRA as well as heap regions with low latency according to an allocation parameter, that is, *reserved* or *normal*, as shown in Figure 4 [LWNnet 2012; Linaro 2013]. In general, when we use an ION interface, requests of several megabytes from a dedicated application are allocated to the SRA while requests of several pages are mapped to the heap.

To further clarify usage of the ION interface, we take an example of a camera application. In Figure 4, *Camera Service* interacts with a device driver that implements functions required to operate a camera hardware [Google 2014c], a *Binder* [Google 2014b] that is a mechanism of *inter-process communication* (IPC), and a *Surface Flinger* that is a system server performing rendering and image synthesis to the frame buffer [Google 2014d]. *Camera Service* and *Surface Flinger* request contiguous pages required to run a camera application in a user memory space through the ION interface, followed by passing parameters to an *ION driver* in the Linux kernel. According to the allocation parameters, either the SRA or the heap is selected. *Camera Service* sets the allocation parameter to *reserved* while *Surface Flinger* and *Binder* set the parameter to *normal*. Based on this, large-sized buffers required for *Camera Service* can be allocated in the SRA while relatively small-sized pages requested by *Surface Flinger* and *Binder* can

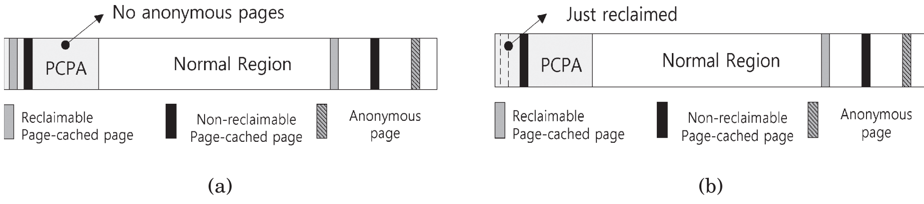


Fig. 5. Memory layout with an example of allocated pages in the PCPA scheme: (a) usual case in the PCPA scheme and (b) clearing the PCPA when an SRA-dependent application needs a contiguous memory block.

be allocated in the heap with a unified ION interface. For this reason, the camera application in an Android system can provide fast launch time.

However, the SRA is only dedicated to some applications, that is, a camera application in this context of the explanation. It is prohibited to be used by any other applications even when the camera application is not activated. Thus, the memory is not efficiently utilized in the SRA scheme, which can affect the performance of non-camera applications in a negative way.

To overcome the inefficiency while maintaining the benefit of the SRA scheme, we propose a novel contiguous memory allocation scheme, called the *page-cached-preferred region* (PCPA) scheme. The following section explains the proposed scheme in detail.

3. PAGE-CACHE-PREFERRED-AREA-BASED PHYSICAL MEMORY MANAGEMENT

In this section, we show the memory layout of our target system. We then technically treat the proposed scheme, followed by PCPA sizing.

3.1. Target System Definition

Figure 5 shows a physical memory layout of the proposed PCPA scheme. The physical memory space is largely divided into two regions: a PCPA region and a normal region. The PCPA region is a memory region where we allocate only page-cached pages of applications, except for some specified ones that previously use an SRA region in the SRA scheme. In such exceptional applications, we allocate both anonymous and page-cached pages as in the SRA scheme. There are two buddy systems, each of which handles one of the memory regions. We denote sets of pages in the PCPA and the normal regions by P^{pcpa} and P^{nor}, respectively. The relationship of the two sets of pages can be expressed as follows:

(1) N is the total number of pages in the system
(2) $P^{pcpa} = \{p_1, p_2, p_3, \ldots p_A\}$, where A[3] is the number of pages in the PCPA
(3) $P^{nor} = \{p_{A+1}, p_{A+2}, p_{A+3}, \ldots p_N\}$, where $N - A$ pages in the normal region
(4) $P^{nor} \cup P^{pcpa}$ corresponds to the total pages in the system

According to page types that are allocated in the PCPA, the state of the PCPA is set to either *ANON_ALLOC* or *PCACHE_ONLY*: only page-cached pages are allowed to be allocated in the PCPA when the state is set to *PCACHE_ONLY* while all the page types can be allocated in *ANON_ALLOC*. Based on the usage of the PCPA region, we classify applications largely into two sets: SRA-dependent and non-SRA-dependent types. A set of SRA-dependent applications is a type of application requiring the SRA in the original SRA scheme, for example, camera; otherwise, non-SRA-dependent.

[3]There is an optimal PCPA size to maximize the effectiveness of the proposed PCPA scheme. We will address this issue in Section 3.3.

ALGORITHM 1: Pseudocode for page allocation

Input: *order* (Contiguous memory order, 2^{order} sized contiguous pages)
Data: app_{cur} (Current application executing *NORMAL_alloc* function)
APP_{SRA} (A set of SRA-dependent applications)
$order_{min}$ (Minimum memory order in the PCPA)
$state_{pcpa} \in \{ANON_ALLOC, PCACHE_ONLY\}$ (PCPA state, initially set to *PCACHE_ONLY*)
Output: *pg* (Allocated page address)

```
1:  Function alloc_pages(order, type)              /* modified version of alloc_pages */
2:  if type = page_cached page then
3:     pg ← allocate 2^0 pages in the PCPA                /* request for page-cached page */
4:     if pg = NULL then
5:        pg ← allocate 2^0 pages in the normal region
6:     end if
7:  else
8:     pg ← NORMAL_alloc(order)                         /* request for anonymous pages */
9:     if pg = NULL then
10:       pg ← NORMAL_alloc(order)
11:       if pg = NULL then
12:          pg ← slow_path_alloc()       /* follow the path in the original Linux kernel */
13:       end if
14:    end if
15: end ifreturn pg
16: Function NORMAL_alloc(order)
17: pg ← allocate 2^order pages in the normal region     /* try in the normal region first */
18: if pg = NULL and order ≥ order_min and app_cur ∈ APP_SRA then
19:    if state_pcpa = PCACHE_ONLY then
20:       Quick_Reclaim(P^pcpa)                      /* clearing the PCPA in Algorithm 2*/
21:    else if state_pcpa = ANON_ALLOC then
22:       pg ← allocate 2^order pages in the PCPA
23:       ANON_ALLOC_flag ← TRUE
24:    end if
25: end ifreturn pg
```

3.2. The Proposed Scheme

The main idea of the proposed scheme is to differentiate the usage of the PCPA according to the application type. For SRA-dependent applications, we allocate both the anonymous and the page-cached pages while we only allow to allocate page-cached pages of non-SRA-dependent applications in the PCPA region. As we allocate simply page-cached pages for non-SRA-dependent applications in the PCPA, we can quickly allocate contiguous anonymous pages for an SRA-dependent application by simply discarding reclaimable page-cached pages in the PCPA, as presented in Figure 5(b).

The proposed PCPA scheme is composed of three components: (1) page allocator, (2) quick memory-reclaimer, and (3) PCPA_timer. They are described in Algorithm 1, Algorithm 2, and Algorithm 3, respectively. In Algorithm 1, allocation of pages in the PCPA and the normal regions is described. Algorithm 2 represents how the page-cached pages in the PCPA region are reclaimed as quickly as possible. Algorithm 3 shows how the best moment to change the state of PCPA is decided.

3.2.1. Page Allocation. Algorithm 1 shows how we allocate contiguous pages in the proposed PCPA scheme. To reuse the most part of memory allocation framework in Android, we augmented the *alloc_pages* function explained in Section 2.1. Accordingly,

we can allocate pages accounting for the application type, that is, SRA-dependent versus non-SRA-dependent, and memory request type, that is, anonymous and page-cached pages (lines 1∼16). First, we check the memory request type (in line 2). If the memory request type is page-cached pages (line 2), we allocate page-cached pages, first in the PCPA region then in the normal region if it is not available in the PCPA (lines 3∼6). In the Linux kernel, page-cached pages are usually file-backed and handled as the unit of 2^0-sized page. When the memory request type is anonymous pages, we allocate the pages using *NORMAL_alloc* functions (lines 8∼15).

In the *NORMAL_alloc* function (lines 17∼27), we first try to allocate 2^{order}-sized pages in the normal region (line 18). If the memory space in the normal region is insufficient (i.e., *pg=NULL*), we first check whether the requested application is in a set of SRA-dependent applications (i.e., $app_{cur} \in APP_{SRA}$). If so, we also check whether the requested memory order is larger than a minimum contiguous memory order of the PCPA (i.e., $order \geq order_{min}$) where $order_{min}$ is a system-specific parameter defined as the smallest size of nodes in *ion_buffer* except for 2^0-sized pages, that is, 4 in our target system. It is checked in order to minimize the memory fragmentation in the PCPA. If all the conditions are met and the state of the PCPA is in *PCACHE_ONLY*, we clear the PCPA region by calling the *Quick_Reclaim* function in Algorithm 2 to secure more memory space in the PCPA (lines 20∼21).

When the state of the PCPA region is in *ANON_ALLOC*, we allocate 2^{order}-sized pages in the PCPA region and set the *ANON_ALLOC_flag* to *TRUE*. The *ANON_ALLOC_flag* is used to determine how long the $state_{pcpa}$ is kept in *ANON_ALLOC* and explained in Algorithm 3. As we simply clear the PCPA region when an SRA-dependent application needs to allocate anonymous pages in the PCPA that was previously used to host only page-cached pages for non-SRA-dependent applications, that is, *PCACHE_ONLY*, we should call the *NORMAL_alloc* function twice (lines 8 and 10). The first call is for the trial to allocate anonymous pages in the normal region (line 8). If we have sufficient contiguous memory space in the normal region, we allocate anonymous pages in this region. Otherwise, we simply clear the PCPA region by calling the *Quick_Reclaim* function to secure contiguous memory space in the PCPA region and change the state of the PCPA to *ANON_ALLOC*. Then, we actually allocate anonymous pages in the PCPA at the second call (line 10). When we cannot find any memory space to handle the request even after clearing the PCPA region, we then invoke *slow_path_alloc* (explained in Section 2.2) to make more memory space by reclaiming pages in the normal region (line 12).

3.2.2. Quick Memory-Reclaiming. Algorithm 2 shows the proposed quick reclaiming solution with the two scanning threads. As finding reclaimable pages requires scanning all the pages in the PCPA, it takes longer time as the size of the PCPA is configured to be larger. To help improve reclaiming speed, we propose a solution utilizing two independent scanning threads. One thread checks from lower addresses moving upward while the other scans from higher addresses downward. When *Quick_Reclaim* is called, we initialize some parameters (lines 2∼7). States of all pages in the PCPA are initially set to *NULL* (line 7), and then updated to *CHECKED* as a corresponding page is scanned (lines 16∼26).

The scanned pages are examined to identify whether *reclaimable* or *non-reclaimable*. A reclaimable page indicates one that is not shared by any other processes and not a dirty page, that is, no need to write-back; otherwise, non-reclaimable. The first scanning thread finding reclaimable pages by moving downward is invoked by sending a wake-up signal (line 8). Then, the second scanning thread moving upward is started (line 9). The two scanning threads independently search reclaimable pages in the PCPA and store them in S until one of them reaches a page that has already been checked by

ALGORITHM 2: Pseudocode for quick memory-reclaiming

Input: $P^{pcpa} = \{p_1, p_2, p_3, \dots p_A\}$
Data: S (set of reclaimable pages), p_i *(state)* (The status of ith page in P^{pcpa})
Require: Create *RT_Thread_Quick_Reclaim* thread before the first run of *Quick_Reclaim*

```
1:  Function Quick_Reclaim(P^pcpa)
2:  S ← φ
3:  FINISH_FLAG ← FALSE
4:  state_pcpa ← ANON_ALLOC
5:  ANON_ALLOC_flag ← FALSE
6:  set PCPA_timer                                          /* start the timer */
7:  ∀i, p_i (state) ← NULL
8:  wake_up_process(RT_Thread_Quick_Reclaim)          /* send a wake-up signal */
9:  Reclaimable_Page_Check(1, A)      /* check from lower addresses moving upward */
10: Wait until FINISH_FLAG is changed to TRUE.
11: free pages (∀ pg ∈ S)       /* return the reclaimed pages to the freed page-lists */

12: Function RT_Thread_Quick_Reclaim(P^pcpa)
13: Sleep and wait for wake up signal
14: Reclaimable_Page_Check(A, 1)          /* scan from higher addresses downward */
15: FINISH_FLAG ← TRUE

16: Function Reclaimable_Page_Check(From, To)        /* find the reclaimable pages */
17: for i ← From to To do
18:    if p_i (state) = CHECKED then
19:       break
20:    else
21:       p_i (state) ← CHECKED
22:       if P_i is reclaimable then
23:          p_i (state) ← CHECKED, S ← S ∪ P_i
24:       end if
25:    end if
26: end for
```

the other (lines 18∼19). Once the condition is met, we set *FINISH_FLAG* to *TRUE* to indicate the completion of the scanning (line 10), and then free the pages in S (line 11).

3.2.3. PCPA_timer. To further maximize the utilization of the PCPA for non-SRA-dependent applications, we develop a solution to control the state of the PCPA region. After an SRA-dependent application has secured several large contiguous page blocks and makes no further memory allocation request, the state of the PCPA can return back to be *PCACHE_ONLY* from *ANON_ALLOC* and remains in this state unless an SRA-dependent application is restarted. The earlier the state goes back to *PCACHE_ONLY*, the more we can possibly utilize the PCPA to allocate page-cached pages for non-SRA-dependent applications. Thus, we need to develop a mechanism to figure out the earliest possible moment for such a state transition.

To achieve this goal, we use a timer and its callback function as shown in Algorithm 3. We set *PCPA_timer* with a timeout value ($T_{timeout}$) and *ANON_ALLOC_flag* to *FALSE* when *Quick_Reclaim* is called (lines 4∼6 in Algorithm 2). While the *PCPA_timer* is ticking, we check whether there is a request for large contiguous pages (2^{order}-sized pages, where $order \geq 4$). If so, we set the *ANON_ALLOC_flag* to *TRUE* (line 24 in Algorithm 1); otherwise, the *ANON_ALLOC_flag* remains *FALSE*. When the *PCPA_timer* is expired and *ANON_ALLOC_flag* is *TRUE*, it means that there was a large contiguous memory allocation request during the previous $T_{timeout}$ interval. Thus, we maintain the state

ALGORITHM 3: Pseudocode for *PCPA_timer*

```
1: Function PCPA_timer_callback()
2: if ANON_ALLOC_flag = TRUE then
3:   ANON_ALLOC_flag ← FALSE
4:   set PCPA_timer
5: else
6:   state_pcpa ← CACHE_ONLY
7: end if
```

of the PCPA with *ANON_ALLOC* as it has higher probability that large contiguous memory requests will occur in the next time interval (line 4 in Algorithm 3).

When the *PCPA_timer* goes off with the *ANON_ALLOC_flag* being *FALSE*, the state of the PCPA region is set back to *PCACHE_ONLY*. In addition, when SRA-dependent applications are finished, all the anonymous pages used by the SRA-dependent applications in the PCPA region are automatically freed because they are not required any more, as they are the heap memory of the applications. Consequently, the state of the PCPA returns to *PCACHE_ONLY* and the freed region can be used to allocate page-cached pages of non-SRA-dependent applications.

We should carefully determine $T_{timeout}$ as it strongly affects the effectiveness of the PCPA scheme. A too-large value will delay the state transition from *ANON_ALLOC* to *PCACHE_ONLY*, thereby losing a chance to allocate more page-cached pages in the PCPA. On the contrary, a too-small value can result in the case where the state of the PCPA returns to *PCACHE_ONLY* too early before all the necessary allocations of sufficient contiguous page blocks are done. Through extensive experiments, we have found that $T_{timeout} = 0.5$ (sec) is most appropriate in our target system.

3.3. Sizing the PCPA

In the proposed PCPA scheme, the size of the PCPA strongly affects performance of SRA-dependent applications, especially when they are required to secure a large memory block at start-up. When the size of the PCPA is configured to a small value, the launch time is high as the probability to secure contiguous memory space with and without *Quick_Reclaim* is low, thereby requiring to invoke *slow_path_alloc* (explained in Section 2.2.1). As explained in Algorithm 1, we first try to allocate anonymous pages in the non-SRA-dependent region. As it can be highly fragmented because we allow to allocate even 2^0-sized pages in this region, the probability that we cannot find the required contiguous memory in the *normal* region is high. Then, we need to perform *Quick_Reclaim* as an attempt to allocate the requested anonymous pages in the PCPA. However, despite *Quick_Reclaim*, the probability of finding the contiguous memory in the PCPA is still low as the PCPA size is small. Consequently, we end up running a slow reclaiming process, that is, *slow_path_alloc* in line 12 of Algorithm 1.

On the contrary, if the size of the PCPA is large, the probability of securing large memory chunks in the PCPA after *Quick_Reclaim* is high. Thus, we can manage to allocate a large contiguous memory request with low overhead. At the same time, however, it takes longer time to clear the PCPA with *Quick_Reclaim* as the amount of the memory space that requires to be scanned is increased. Furthermore, the launch time of SRA-dependent applications can be further degraded as *Quick_Reclaim* is called more frequently, since the probability of failure to allocate anonymous pages in *normal* region is increased as the size of *normal* region is smaller. The preceding explanation can be formally represented as follows. First, let us define the existence probability of freed 2^n-sized pages in a certain region "R" as $Prob(2^n, \text{R})$. With this definition, we can derive the following.

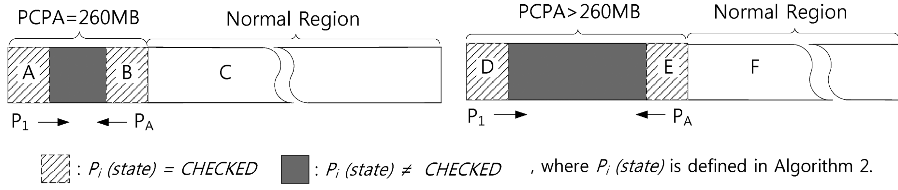


Fig. 6. Snapshot of processing *Quick_Reclaim* when PCPA size = 260MB and PCPA size > 260MB.

(1) In Figure 6, $Prob(2^n, A)$, $Prob(2^n, B)$, $Prob(2^n, D)$ and $Prob(2^n, E)$ are identical, where n is 4 or 8.
(2) As described in Figure 1(b), $Prob(2^8, C) \approx Prob(2^8, F) \approx 0$.
(3) Assuming that C and F are fragmented, $Prob(2^4, C) > Prob(2^4, F)$ because $C > F$.
(4) $Prob(2^8, A+B+C) + Prob(2^4, A+B+C) > Prob(2^8, D+E+F) + Prob(2^4, D+E+F)$.

Note that the performance of non-SRA-dependent applications is independent of the size of the PCPA because the only difference for such applications is the location of page-cached pages while the amounts of memory space for anonymous and page-cached pages are identical, even though the size of the PCPA is varied.

Thus, in order to maximize the effectiveness of the proposed PCPA scheme, we need to find the optimal PCPA size in the design time. Note that the optimal point can be varied according to target systems. As explained in Section 2.2, N_{alloc} is obtained after establishing *ion_buffer* of an SRA-dependent application. We should consider N_{alloc} and execution time of *Quick_Reclaim* simultaneously to find the optimal PCPA size. Since such a trade-off analysis is a complex problem, we depend on an engineering approach. In our experiments, we chose 260MB as the optimal PCPA size, and details of the analysis will be described in Section 4.4.

4. EXPERIMENTS

In this section, we explain the experiments for evaluating the performance improved by the proposed scheme. We first describe our experimental setup. We then show and analyze the results of benchmark tests and real-life Android application tests. Finally, we present the trade-off and overhead analysis of our approach.

4.1. Experimental Setup

We performed experiments on a Galaxy Note 3 smartphone with octa-core Exynos 5420 that consists of four ARM Cortex-A15 and four ARM Cortex-A7 cores. The Galaxy Note 3 also has an AMOLED 1920x1080 display, a 13-megapixel camera, 3GB of RAM, and 32GB-sized eMMC flash device. We used Android 4.3 with Linux kernel 3.4.39. In the original target system, various applications are configured to use *statically reserved area* (SRA), such as camera, WiFi Direct, Codec, etc. Among them, the camera application is dominant in terms of allocated SRA size. The size of SRA for the camera application amounts to 260MB, which corresponds to 88.7% of the total size of the SRA. Therefore, to ease the experiments without losing any accuracy of the results, we simply considered the subregion of the camera as the entire SRA while ignoring the others.

4.2. Benchmark Test

As a metric for evaluating performance improvement, we used the average execution time of benchmarks. We measured it under the SRA scheme and our proposed scheme. To demonstrate the effectiveness of increased available memory space, we chose memory- and IO-intensive benchmarks instead of CPU-intensive ones. To see whether the abundance of page-cached pages had an effect on execution times of

Table II. Memory Allocation Test Using *Sysbench*

Test block size	128KB		100MB	
Scheme	SRA	PCPA	SRA	PCPA
Free Mem. (MB): before	54.4	320.4	54.3	318.1
Free Mem. (MB): after	53.9	320.2	134.1	318.0
Page-cached Mem. (MB): before	713.2	714.6	711.4	712.9
Page-cached Mem. (MB): after	713.2	714.6	641.5	712.9
Execution time	135.9ms	136.4ms	801μs	730μs

benchmarks, we checked the amount of page-cached pages after each test. In these experiments, we used multiple benchmark suites: *Sysbench* [Sysbench 2014] and *IOzone*[IOzone 2006][4].

To reflect a reproducible memory-constrained environment, we configured the test conditions as follows.

(1) We disabled *low-memory killer* (LMK) [Kalkov et al. 2012] that kills background tasks to get available pages when the Android system detects memory shortage. This is because the time consumed by LMK operation can affect the performance of the system and newly obtained free pages with LMK can change the execution time of each test.
(2) Before each start of test, we imposed 1GB random memory allocation to our experimental target right after system booting in order to generate memory-constrained status.
(3) We powered-off the target after one test finished and powered-on again for the next test.

4.2.1. Memory-Intensive Cases. We used the *memory* test mode in *Sysbench* to validate the effectiveness of the proposed PCPA scheme in memory-intensive cases. Table II shows the average execution time of running the *memory* test mode by setting the block size for each access to 128KB and 100MB, the number of threads to 32, and the total access size to 3GB. In the table, *before* and *after* indicate the memory state before and after executing *Sysbench*, respectively. In the case of 128KB, there is almost no memory status change between before and after in both schemes, so there is almost no difference in execution time, that is, 135.9 versus 136.4ms. However, for the case of 100MB, the SRA scheme has just 54.3MB of free memory before the execution, so the remaining required pages are secured by reclaiming mostly page-cached pages (711.4MB $\rightarrow$ 641.5MB). On the contrary, the PCPA scheme does not need memory-reclaiming because the free memory space is enough for a 100MB-sized allocation request (318.1MB). Consequently, there is around 9% of performance improvement in terms of the execution time in the memory-intensive test.

4.2.2. IO-Intensive Cases. To validate the effectiveness of the proposed PCPA solution in an IO-intensive scenario, we used two benchmarks: *fileio* test mode of *Sysbench* and *IOzone* [IOzone 2006]. In each benchmark suite, there are several types of benchmark tests. For the case of *Sysbench*, all the types of tests are done individually, thus each test has no influence on others. Contrary to *Sysbench*, all the types of benchmark tests of *IOzone* are executed sequentially, thus each test is correlated to the previous test.

Figure 7 shows the results of the *fileio* test mode of *Sysbench* when we set the block size as 128KB and 4KB. We configured the prefetch size to 256KB. Each test created

[4]We also used memory-intensive benchmarks (*bzip2*, *mcf*, etc.) in the SPEC CPU2006 benchmark suite [SPECCPU2006 2014] in our evaluation. However, they use memory spaces in an iterative way and not in a bulky manner. Thus we did not show the results in this article.

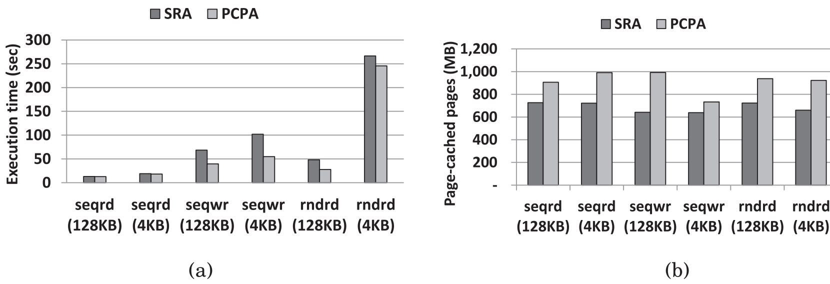


Fig. 7. File I/O test using *Sysbench*: (a) average execution time of *Sysbench* and (b) average size of page-cached pages after each test.

16 files. The *fileio* test mode in *Sysbench* is composed of three types of benchmark tests, that is, *sequential read* (i.e., seqrd), *random read* (i.e., rndrd), and *sequential write* (i.e., seqwr). In Figure 7(a), the x- and y-axis represent the test type and the average execution time, respectively.

In the *sequential write* test, the execution time was reduced by 42% and 46% compared to the SRA scheme when the block size was 128KB and 4KB, respectively. While *sequential write* is running, page-cached pages are used for the buffer to keep the *write* data before actual flash write I/Os happen. Thus, the more memory space for page-cached pages is provided, the less flash write I/O happens. If there is no more memory space for page-cached pages, some pages should be reclaimed, and the dirty pages which are reclaimed need write-back to the flash. As shown in Figure 7(b), the amount of page-cached pages under the PCPA scheme was increased by 35% and 13% when the block size was 128KB and 4KB, respectively. In *sequential write* test, the PCPA scheme benefited from the increased page-cached pages. In the *random read* test, the execution time was reduced by 42% and 8% compared to the SRA scheme when the block size was 128KB and 4KB, respectively. As the size of page-cached pages in the PCPA scheme was increased by 23% and 28%, more data were cached in the PCPA region. Thus many file reads were hit in the page-cached pages.

On the other hand, the *sequential read* test resulted in almost the same execution time. This can be explained as follows. The file I/O transaction is interfaced with three SW layers, consisting of a user layer, a file system layer, and a block I/O layer. For read operation, a user-layer function (by the system call from *Sysbench*) seeks for page-cached pages in the file system layer. If page-cached pages have the right data for that system call, that is, hit case, the data are returned immediately. However, in the case of a miss, the block I/O layer checks whether the request type is sequential or random [Fengguang et al. 2008], and then the eMMC is accessed using the IO scheduler [Love 2004]. Once the request type for read operation turns out to be sequential access in the block I/O layer, a *read_ahead* operation is activated, and then more data than requested are read from the eMMC to the memory space for page-cached pages. Thus, the first 128KB-sized *read* access created miss state, but from the second access, hit state was recorded. This is because an additional 128KB was already read from the eMMC to page-cached pages during the prefetch of the first access [Fengguang et al. 2008]. Even if the system is little short of free memory space to create page-cached pages, during the previous access, through reclaiming memory, the required amount of page-cached pages is provided. Therefore the time spent for memory-reclaiming can be hidden.

Figure 8 shows the test results of the *IOzone* benchmark when we configured the block size of each file I/O transaction to 128KB (Figure 8(a)) and 4KB (Figure 8(b)), respectively. In this test, we ran *IOzone* 10 times and averaged the test results. The x- and y-axis represent the test sequence and the normalized execution time, respectively.

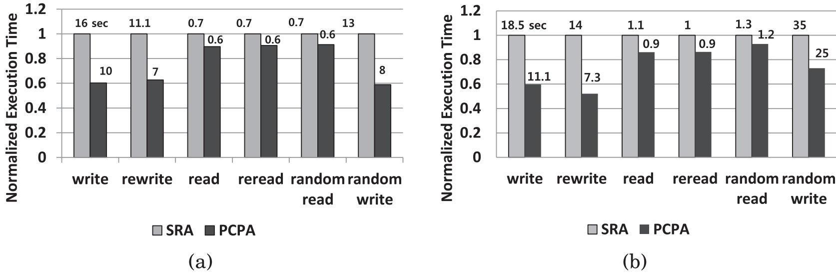


Fig. 8. Normalized execution time of *IOzone*: (a) block size = 128KB and (b) block size = 4KB.

Table III. Average Size of Page-Cached Pages after *IOzone*

File I/O Transaction Size	SRA	PCPA
4KB	126.9MB	245.9MB
128KB	124.1MB	382.1MB

The *IOzone* benchmark makes a single file and it is used in each test step represented in the x-axis. As we set the total transaction size as 512MB, the same single 512MB-sized file was used in each test.

As depicted in the figure, the average improvement of the *IOzone* benchmark was 24.7% and Table III shows the average amount of page-cached pages after the *IOzone* test in each scheme. In the case of *write*, the execution time of our scheme was reduced by 40% compared to the SRA scheme in both cases of transaction block sizes. We used the same condition with the *memory* test mode in *Sysbench*, so the amount of free pages at the initial stage of each test was the same as that of Table II. Our proposed scheme had more free pages, approximately 260MB, than the SRA scheme and these free pages were used as the buffer before actual *write* to the eMMC. However, as the SRA scheme had less free pages than our scheme, it should reclaim page-cached pages to make new free pages for the *write* buffer. Due to the page reclamation, the SRA scheme spent more execution time in *write*. During the first *write* test, most free pages were used for page-cached pages. As described in Table III, our scheme can hold more page-cached pages through the PCPA region than the SRA scheme. As a result, in *rewrite* test, our scheme experienced more write-hit cases than the SRA scheme. However, as the SRA scheme held fewer page-cached pages, it triggered page reclamation for the *rewrite* and flash write I/Os for the reclaimed pages. Thus the execution time was increased by 38% and 48% in each case of block size. As for the case of *rewrite*, in the three consecutive *read* tests, more read-requested pages were found in the memory with the aid of the PCPA region. Thus the execution times of three *read* tests were reduced by 8%$\sim$13% compared to the SRA scheme.

4.3. Real-Life Android Applications

To quantitatively validate the effectiveness of increased page-cached pages through the PCPA region in real-life Android applications, we evaluated the performance of non-SRA- and SRA-dependent applications. We chose the launch time of these applications as our performance metric. The application launch time is an important performance metric, especially for Android system users, since they dynamically switch a foreground application from time to time. However, it is not straightforward to measure an accurate application launch time. We therefore measured the timing information of applications through the *Activity Manager* [Google 2014a] module in the Android framework. The key of the proposed PCPA scheme is to maximize the utilization of the PCPA so as

to improve the launch time of non-SRA-dependent applications while maintaining the launch time of SRA-dependent applications.

To reflect the memory status shown in Figure 1(b), different from the random memory allocation used in Section 4.2, we used the *monkeyrunner* tool [Google 2014e] to generate memory-constrained status of real-life Android systems. This tool can impose randomly generated user inputs as general smartphone users touch the screen to operate applications.

4.3.1. Non-SRA-Dependent Applications. We selected and ran the most widely used non-SRA-dependent applications in Android systems: *gallery*, *youtube*, *calendar*, *settings*, *video*, *music*, *google search*, *myfiles*, and *voicenote*. To reflect the realistic memory status of Android systems, we first ran the *monkeyrunner* tool to generate random events created when Android applications are starting and running. We then randomly launched the aforementioned non-SRA-dependent applications multiple times while the tool was running. We recorded all the events of the target system in a log file. We extracted the launch time of the target applications from the log file and averaged them. In this test, we also disabled *low-memory killer* (LMK) [Kalkov et al. 2012] as in Section 4.2 to remove the influence on the launch time by newly obtained free pages.

To see the effect of the page-cached pages on the application launch time, we examined the size of available page-cached pages. We also measured the amount of flash read and write I/O, since it is directly related to the size of page-cached pages.

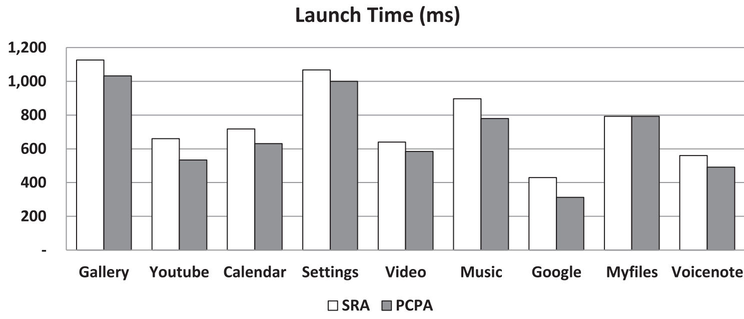
Figure 9(a) plots the average launch times of the applications under the SRA and the PCPA schemes. As expected, the launch time under the proposed scheme was effectively reduced by 9.2% on average. This result comes from the increased amount of page-cached pages by the PCPA region as shown in Figure 9(b). In the figure, we can see that 13.1MB difference in page-cached pages between before and after executing each application under the SRA scheme. It means the SRA scheme needed to reclaim the existing page-cached pages to create new free pages for anonymous pages or page-cached pages, and it also affected the launch times of applications. In contrast, in our proposed scheme, the change in amount of page-cached pages was only 540KB. This is because many file accesses were hit in page-cached pages, so there was no need to reclaim page-cached pages.

Figure 9(c) shows that 65% of average flash read and write I/Os are reduced due to the increased page-cached pages. Thus the reduction of flash read and write I/O improved the launch time.

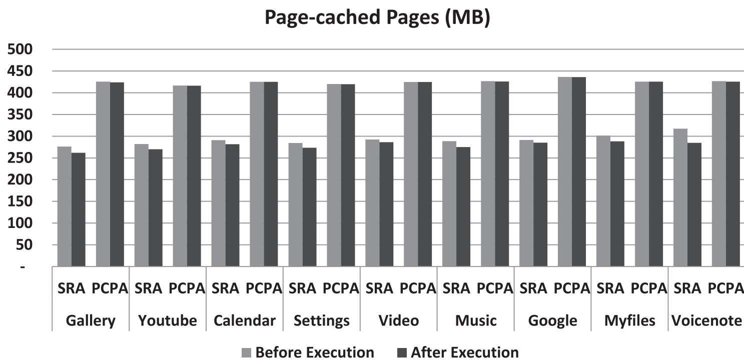
Note that the launch-time difference of *myfiles* was not noticeably reduced, unlike other non-SRA-dependent applications. This is because *myfiles* generates a small number of file accesses when it launches. Instead, it creates many file accesses when users open a directory or search for a file after launching the application. As shown in Figure 9(c), *myfiles* recorded the smallest number of flash read and write I/Os among the non-SRA-dependent applications in both schemes. As previously described, this application is less I/O-intensive than others. Thus the increased size of page-cached pages does not benefit *myfiles*, different from the others.

For further analysis of the launch time, we discuss how the *Quick_Reclaim* additionally incurred by our approach affects the launch time of non-SRA-dependent applications. In order to explain this, we first classify physical pages using the notations in Section 3.1. Let $P^{total} = P^{nor} \cup P^{pcpa}$ and $p_i \in P^{total}$. Then, there exists three cases for a page-cached page p_i before the camera application is launching:

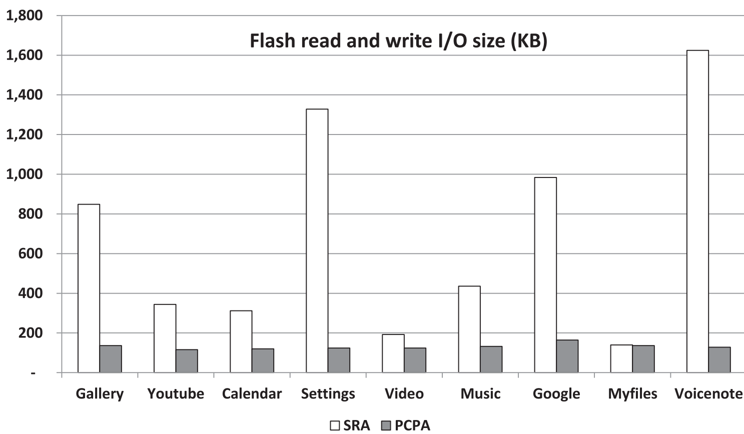
(1) $p_i \in P^{nor}$;
(2) $p_i \in P^{pcpa}$, where p_i is a dirty page or a page shared by other processes;
(3) $p_i \in P^{pcpa}$, where p_i is a clean page-cached page.



(a)



(b)



(c)

Fig. 9. Effects of the PCPA scheme when running real-life non-SRA-dependent Android applications: (a) average launch time, (b) average page-cached pages before and after starting each application, and (c) average flash read and write I/O size.

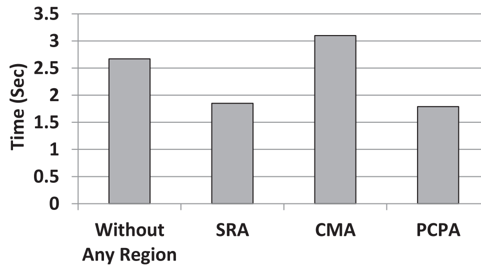


Fig. 10. Average launch time of the camera application.

Consider a non-SRA-dependent application which uses p_i as a page-cached page after the camera application finishes its execution. In case of (1) and (2), p_i is not reclaimed so the file access is hit in the page-cached page. Thus the relaunch time of the application is not delayed. To the contrary, in the case of (3), p_i is reclaimed by *Quick_Reclaim* and the relaunch time of the application can deteriorate. Contrary to intuition, our scheme outperforms the SRA scheme in average launch time of non-SRA-dependent applications as shown in Figure 9(a). This can be explained as follows. The size of P^{nor} is 10.53 times larger than that of P^{pcpa} so the existence probability of $p_i \in P^{nor}$ is much larger than that of $p_i \in P^{pcpa}$. In the case of (2), even if p_i was in P^{pcpa}, it is excluded from the execution of *Quick_Reclaim*. In the case of (3), p_i is evicted by *Quick_Reclaim* and the PCPA region is quickly refilled with page-cached pages of all applications after the camera application finishes. At this point in time, the first relaunch of the application is delayed and p_i resides in P^{nor} again. From the second relaunch of the application, as the PCPA scheme has larger space to keep the page-cached pages, the probability of $p_i \in P^{total}$ is higher than that of the SRA scheme. Consequently, from the viewpoint of average launch time, case (3) does not damage the launch time of non-SRA-dependent applications.

4.3.2. SRA-Dependent Applications. As an SRA-dependent application, we chose the camera application. To reflect the realistic memory status of Android systems, we first ran the *monkeyrunner* tool as in the non-SRA-dependent application tests. We then randomly launched the camera application multiple times while the tool was running. We measured the average launch time of the camera application under our target system with four different configurations: (1) without any special memory region, (2) with the SRA region, (3) with the CMA region, and (4) with the PCPA region. Figure 10 shows the result of each configuration.

First, we configured the target system not to have any special memory region and used it as a baseline to compare with other schemes. We can see that the average launch time was 2.67 seconds in the figure, the second-worst result among the configurations. Since there is no special memory region for the camera application, all the contiguous page blocks required for launching the camera should be provided by the operating system. In addition, as the randomly generated events made the memory status as in the case of Figure 1(b), the number of 2^4- or 2^8-sized page blocks required for *ion_buffer* was small, as explained in Section 2.2. As a result, memory allocation time for establishing *ion_buffer* was increased and affected the launch time.

Under the SRA scheme, we can see in the figure that the launch time was improved compared to the aforementioned first test configuration. Since the predefined SRA region is always ready for launching the camera application in the SRA scheme, there was no need to establish *ion_buffer*. In the figure, we can see that it took 1.85 seconds to launch, nevertheless it had no overhead in provisioning contiguous pages. This is because, as explained in Section 2.3, Android framework modules, that is, *Surface Flinger*

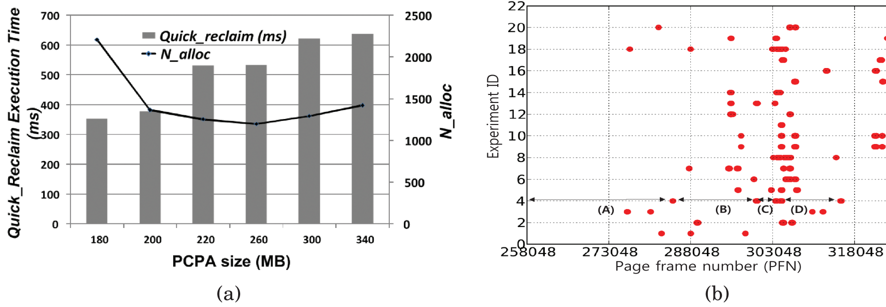


Fig. 11. Trade-off and overhead analysis: (a) average execution time for *Quick_Reclaim* and average measured *N_alloc* when running an SRA-dependent application (i.e., camera) with varying the PCPA size; (b) distribution of available pages in the PCPA after *Quick_Reclaim*.

and *Binder*, also seek for contiguous memory space during the launching process. Accordingly, we can know that if Android systems are in a memory status as shown in Figure 1(b), provision of contiguous pages for the Android framework modules can affect the launch time of the camera application.

The CMA scheme resulted in the worst launch time compared to others. Under the CMA scheme, any application can allocate anonymous pages as well as page-cached pages in the CMA region. When the camera application launches, this region should be cleared by reclaiming page-cached pages or migrating anonymous pages. When we migrated 200MB-sized pages from the CMA region to the normal region, this alone took 1.66 seconds in our target system. What was worse, write-back of dirty pages to the eMMC incurred additional delay.

In spite of the overhead in memory reclamation and establishing *ion_buffer*, the PCPA scheme represented the best launch time. The first reason is that the PCPA scheme keeps more page-cached pages than the SRA scheme. Thus many file accesses of the camera application could be hit in the page-cached pages. Second, as the page blocks requested by *Surface Flinger* and *Binder* can be allocated in the PCPA region, the memory allocation latency induced by provisioning contiguous pages for Android framework modules was reduced.

4.4. Trade-off and Overhead Analysis

In order to analyze the trade-off between the PCPA size and memory allocation latency, we measured the execution time of *Quick_Reclaim* while varying the PCPA size. At the same time, we measured N_{alloc} according to the PCPA size to find the optimal point which brings about the smallest number of calls for *alloc_pages*(n). To clear the PCPA region with *Quick_Reclaim*, we ran the camera application. Figure 11(a) shows the dependency between N_{alloc} and the size of the PCPA (in the x-axis) in our target system. When the PCPA size is smaller than 180MB, N_{alloc} drastically increases. In the case of PCPA larger than 340MB, *Quick_Reclaim* execution time is over 700ms and, due to the reduced size of the normal region, LMK [Kalkov et al. 2012] can be triggered more often. Therefore we ruled out the cases when the PCPA size is smaller than 180MB or larger than 340MB in this experiment.

As shown in the figure, N_{alloc} is minimized when the PCPA size is set to 260MB. When we compare the PCPA sizes at points 200MB and 260MB, the *Quick_Reclaim* execution time at 200MB is faster than the case of 260MB by 140ms. However, we choose 260MB as the optimal PCPA size (N_{alloc} is smaller by 200) by jointly considering the TLB miss ratio and the memory allocation latency for overall system overhead. To minimize *Quick_Reclaim* execution time while maintaining the smallest N_{alloc}, we used

multithreaded page reclaiming as described in Algorithm 2. In addition, we excluded dirty pages in scanning to eliminate the additional time for write-back. As a result, we can achieve 520ms execution time in *Quick_Reclaim*.

In addition to the execution time, we examined non-reclaimable pages after executing *Quick_Reclaim*. As the launch time can be varied according to the distribution of non-reclaimable pages, we performed 20 runs in total. Figure 11(b) shows the memory state of the PCPA after *Quick_Reclaim*. The y-axis denotes the experiment ID and the x-axis represents the *page frame number* (PFN) of the PCPA. The start- and end-points of the x-axis correspond to $P_1 \in P^{pcpa}$ (PFN = 258048) and $P_A \in P^{pcpa}$ (PFN = 325119), respectively. The dots in Figure 11(b) indicate the PFNs of non-reclaimable pages after execution of *Quick_Reclaim*. As an example, in the experiment ID #4, the memory space, region (A), has contiguous free pages ranging from P_1 (PFN 258048) to P_{26452} (PFN 284500), meaning around free 103MB-sized contiguous pages. Regions (B), (C), and (D) also represent tens of megabytes, respectively. As shown in the figure, we can vastly reduce the number of *alloc_pages*(0) calls in constructing *ion_buffer* while increasing the number of calls for *alloc_pages*(8) and *alloc_pages*(4), that is, small *N_alloc*.

5. RELATED WORK

Studies on provisioning of contiguous memory spaces have mainly focused on two categories: developing a good memory allocation and a free-space management policy and page migration based on exchanging the position of used and freed pages.

5.1. Allocation and Free-Space Management

Johnstone and Wilson [1998] insisted that well-designed allocators can reduce fragmentation and also showed that a segregated fit-policy-based good-fit policy performs well in terms of memory fragmentation. Based on that work, Masmano et al. [2004] proposed the TLSF (*two-level segregated fit*) memory allocator, which manages a large group of segregated free blocks using a two-level array of free blocks, and gave a solution to the fragmentation problem in dynamic memory allocation. Ramakrishna et al. [2008] proposed a smart dynamic memory allocator to provide effective utilization of memory. Free-lists (linked list for freed pages) are subdivided into short-lived and long-lived objects, and one side of the memory is allocated with long-lived objects while the other side is filled with short-lived objects. Thus, one side for free-lists assigned to short-lived objects can be used as the non-fragmented region. The hoard allocator, [Berger et al. 2000] manages one global heap and per-processor heaps to bound blowup to a constant factor, where blowup is an indicator of memory fragmentation. To eliminate a large amount of time for defragmentation activities followed by the invocation of *malloc*() and *free*(), a defrag-dodging approach was developed and benefited Web-based applications [Inoue et al. 2009].

5.2. Page Migration

When the Linux kernel fails to get free contiguous pages, it moves the currently used pages to the free region at higher physical address and increases the free region at lower physical address if the flag for compaction, *CONFIG_COMPACTION*, is set [Corbet 2010]. Therefore many consecutive free pages are found at lower physical address in memory. Rental memory management [Jeong et al. 2013] also uses a special region as the SRA scheme and this special region always tries to keep clean page-cached pages. Whenever dirty page-cached pages are found, they are moved to the normal region. By doing so, an empty memory chunk can be found after eviction of contents in that special region. The advanced version of rental memory management, the eCache-based scheme [Jeong et al. 2012], uses a special region, called eCache, for page-cached pages. The least accessed file-mapped caches are moved to the eCache. Thus the probability

of existence of write-back pages in the eCache is very low. Therefore, by evicting page-cached pages in the eCache, an empty memory space can be given. Rental memory management and the eCache-based scheme are good solutions for the system which does not support IOMMU and better than the CMA-based scheme in clearing that special region which should be used for allocating contiguous pages.

Our PCPA scheme, however, does not focus on clearing all the page-cached pages in that special region as do eCache and the PCPA. Instead, the PCPA scheme is most effective in the system which supports an IOMMU, thus, we focused on how to acquire as many quickly freed contiguous pages as possible, even though they are not totally consecutive.

6. CONCLUSION

With the increased size of multimedia contents, allocating a large amount of physically contiguous memory becomes an important issue. To cope with such an issue, the SRA-based and CMA-based schemes have been introduced, but neither supports efficient memory usage and fast memory allocation time simultaneously. In addition, although an IOMMU is supported, if the required contiguous memory space is large, embedded system designers are reluctant to use an IOMMU due to its memory allocation latency in operating the IOMMU.

In this article, we proposed a new physical memory management policy, named PCPA scheme, which overcomes the inefficiency of the SRA scheme's memory utilization and late response time of the CMA-based scheme in securing a physically contiguous memory region. Using a quick memory-reclaiming mechanism with a special region named PCPA, we could rapidly allocate a number of contiguous memory blocks, thereby overcoming the induced page management overhead in using the IOMMU. Our PCPA scheme is almost equivalent to the SRA scheme when we compare the launch time of an application which uses the SRA region in the SRA scheme. To point out the inefficient memory usage induced by the SRA, we compared the PCPA scheme with the SRA scheme using benchmarks and real-world Android applications. Our evaluation showed that, compared to the SRA scheme, the average execution time of I/O-intensive benchmarks was improved by 24.7% and the average launch time was reduced by 9.2% in sampled real-world Android applications with the aid of enhanced memory utilization.

REFERENCES

Nadav Amit, Muli Ben-Yehuda, Dan Tsafrir, and Assaf Schuster. 2011. vIOMMU: Efficient IOMMU emulation. In *Proceedings of the Annual USENIX Technical Conference (USENIXATC'11)*.

Nadav Amit, Muli Ben-Yehuda, and Ben-Ami Yassour. 2010. IOMMU: Strategies for mitigating the IOTLB bottleneck. In *Proceedings of the International Conference on Computer Architecture (ISCA'10)*. 256–274.

Roberto Ammendola, Andrea Biagioni, Ottorino Frezza, Francesca Lo Cicero, Alessandro Lonardo, Pier Stanislao Paolucci, Davide Rossetti, Francesco Simula, Laura Tosoratto, and Piero Vicini. 2013. Virtual-to-physical address translation for an FPGA-based interconnect with host and GPU remote DMA capabilities. In *Proceedings of the International Conference on Field-Programmable Technology (FPT'13)*. 58–65.

ARM Architecture. 2014. Cortex-A9 technical reference manual. http://infocenter.arm.com/help/topic/com.arm.doc.ddi0388i/DDI0388I_cortexa9_r4p1_trm.pdf.

Emery D. Berger, Kathryn S. Mckinley, Robert D. Blumofe, and Paul R. Wilson. 2000. Hoard: A scalable memory allocator for multithreaded applications. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'00)*. ACM Press, New York, 117–128.

CMA. 2012. A deep dive into CMA. http://lwn.net/Articles/486301/.

Jonathan Corbet. 2010. Memory compaction (2010). http://lwn.net/Articles/368869/.

Google. 2014a. ActivityManager. http://developer.android.com/reference/android/app/ActivityManager.html.

Google. 2014b. Android.os.Binder. http://developer.android.com/reference/android/os/Binder.html.

Google. 2014c. Camera HAL overview. https://source.android.com/devices/camera/camera.html.

Google. 2014d. Graphics. https://source.android.com/devices/graphics.html.

Google. 2014e. Monkeyrunner. http://developer.android.com/tools/help/monkeyrunner concepts.html.

Jean-Pierre Henot, M. Ropert, Julien Le Tanou, Jean K, and Thomas Guionnet. 2013. High efficiency video coding (HEVC): Replacing or complementing existing compression standards. In *Proceedings of the IEEE International Symposium on Broadband Multimedia Systems and Broadcasting (BMSB'13).*

Hiroshi Inoue, Hideaki Komatsu, and Toshio Nakatani. 2009. A study of memory management for Web-based applications on multicore processors. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'09).* ACM Press, New York, 386–396.

IOzone. 2006. IOzone filesystem benchmark. http://www.iozone.org/.

Jinkyu Jeong, Hwanju Kim, Jaeho Hwang, Joonwon Lee, and Seungryoul Maeng. 2012. DaaC: Device-reserved memory as an eviction-based file cache. In *Proceedings of the International Conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES'12)*, Ahmed Jerraya, Luca P. Carloni, Vincent John Mooney III, and Rodric M. Rabbah, Eds. ACM Press, New York, 191–200.

Jinkyu Jeong, Hwanju Kim, Jeaho Hwang, Joonwon Lee, and Seungryoul Maeng. 2013. Rigorous rental memory management for embedded systems. *ACM Trans. Embedd. Comput. Syst.* 12, 1.

Mark S. Johnstone and Paul R. Wilson. 1998. The memory fragmentation problem: Solved? In *Proceedings of the 1st International Symposium on Memory Management (ISMM'98).* ACM Press, New York, 26–36.

Igor Kalkov, Dominik Franke, John F. Schommer, and Stefan Kowalewski. 2012. A real-time extension to the Android platform. In *Proceedings of the 10th International Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES'12).* ACM Press, New York, 105–114.

Myung-Jin Lee, Joo-Yong Oh, and Soon-Ju Kang. 2011. Design of multimedia stream channel arbiter in home network gateway. *IEEE Trans. Consumer Electron.* 57, 4, 1661–1669.

Linaro. 2013. ION interface and memory allocator. https://wiki.linaro.org/BenjaminGaignard/ion.

Robert Love. 2004. I/O scheduler. http://www.linuxjournal.com/article/6931.

Robert Love. 2010. *Linux Kernel Development*, 3rd ed. Addison Wesley.

Lwnnet. 2012. The Android ION memory allocator. http://lwn.net/Articles/480055/.

Miguel Masmano, Ismael Ripoll, Alfons Crespo, and Jorge Real. 2004. TLSF: A new dynamic memory allocator for real-time systems. In *Proceedings of the 16th Euromicro Conference on Real-Time Systems (ECRTS'04).* 79–88.

Rivalino Matias, Tais Ferreira, and Autran Macedo. 2011. An experimental study on user-level memory allocators in middleware applications. In *Proceedings of the IEEE International Conference on Systems, Man, and Cybernetics (SMC'11).* 2431–2436.

M. Ramakrishna, Jisung Kim, Woohyong Lee, and Youngki Chung. 2008. Smart dynamic memory allocator for embedded systems. In *Proceedings of the 23rd International Symposium on Computer and Information Sciences (ISCIS'08).* 1–6.

SPEC CPU2006. 2014. SPEC's benchmarks and published results. http://www.spec.org/benchmarks/html.

Sysbench. 2014. SysBench: A system performance benchmark. https://launchpad.net/sysbench.

Sumesh Udayakumaran, Angel Dominguez, and Rajeev Barua. 2006. Dynamic allocation for scratch-pad memory using compile-time decisions. *ACM Trans. Embedd. Comput. Syst.* 5, 2, 472–511.

Guibin Wang and Wei Song. 2011. Communication-aware task partition and voltage scaling for energy minimization on heterogeneous parallel systems. In *Proceedings of the 12th International Conference on Parallel and Distributed Computing, Applications and Technologies (PDCAT'11).* 327–333.

Fengguang Wu, Hongsheng Xi, and Chenfeng Xu. 2008. On the design of a new Linux readahead frame-work. *SIGOPS Oper. Syst. Rev.* 42, 5, 75–84.

Chia-Hao Yu, Chung-Kai Liu, Chih-Heng Kang, Tsun-Hsien Wang, Chih-Chien Shen, and Shau-Yin Tseng. 2007. An efficient DMA controller for multimedia application in MPU based SOC. In *Proceedings of the International Conference on Multimedia and Expo (ICME'07).* 80–83.

Received January 2015; revised April 2015; accepted May 2015