Rigorous Rental Memory Management for Embedded Systems

Jinkyu Jeong[*] Hwanju Kim[†] Jeaho Hwang[‡]
KAIST KAIST KAIST
Joonwon Lee[§]
Sungkyunkwan University

Seungryoul Maeng[**]
KAIST

CS/TR-2011-349

July 15, 2011

K A I S T
Department of Computer Science

[*] jinkyu@calab.kaist.ac.kr
[†] hjukim@ calab.kaist.ac.kr
[‡] jhhwang@calab.kaist.ac.kr
[§] joonwon@skku.edu
[**] maeng@calab.kaist.ac.kr

Rigorous Rental Memory Management for Embedded Systems

Jinkyu Jeong, Korea Advanced Institute of Science and Technology
Hwanju Kim, Korea Advanced Institute of Science and Technology
Jeaho Hwang, Korea Advanced Institute of Science and Technology
Joonwon Lee, Sungkyunkwan University
Seungryoul Maeng, Korea Advanced Institute of Science and Technology

Memory reservation in embedded systems is a prevalent approach to provide a physically contiguous memory region to its integrated devices, such as a camera device and a video decoder. Inefficiency of the memory reservation becomes a more significant problem in emerging embedded systems, such as smartphones and smart TVs. Many ways of using the systems increase the idle time of their integrated devices, and eventually decrease the utilization of their reserved memory.

In this paper, we propose a scheme to minimize the memory inefficiency caused by the memory reservation. The memory space reserved for a device may be rented for other purposes when the device is not active. For this scheme to be viable, latencies associated with reallocating the memory space should be minimal. Volatile pages are good candidates for such page reallocation since they can be reclaimed immediately when they are needed by the original device. We also provide two optimization techniques, lazy-migration and adaptive-activation. The former increases the lowered utilization of the rental memory by our volatile page allocations, and the latter saves active pages in the rental memory during the reallocation.

We implemented our scheme on a smartphone development board with the Android Linux kernel. Our prototype has shown that the time for the return operation is less than 0.77 seconds in the tested cases. We believe that this time is acceptable to end-users in terms of transparency since the time can be hidden in application initialization time. The rental memory also brings throughput increases ranging from 2% to 200% based on the available memory and the applications' memory intensiveness.

Categories and Subject Descriptors: D.4.2 [**Operating Systems**]: Storage Management—*Main memory*; D.4.7 [**Operating Systems**]: Organization and Design—*Real-time systems and embedded systems*

General Terms: Design, Management

Additional Key Words and Phrases: memory management, memory hotplugging, SoC, SiP

1. INTRODUCTION

Recently, embedded systems are increasingly versatile leveraging a feature of hardware integration in the form of System-on-a-Chip or System-in-a-Package. Smartphones and digital televisions now redelegate the task of multimedia decoding from CPU to an integrated hardware decoder because this delegation reduces the CPU load along with also reducing energy consumption. Meanwhile, an integrated device, such as a video decoder or a camera device, usually requires tens of megabytes of memory for its computation [Chiodo et al. 1994; Tseng et al. 2005; Corbet 2011]. While some devices have their own memory areas, which are separated from the main memory of the system, many devices reserve a part of the main memory to reduce cost and space [Hu et al. 2004; Corbet 2011]. Recent smartphones reserve about 80-154MB of memory for their integrated devices. The portion of the reserved memory ranges from 15% to 22% of the total memory in systems. The main reason for the memory reservation is to provide physically contiguous memory space to an integrated device, which makes programming easier.

The main problem of this memory reservation is memory inefficiency. When an integrated device is idle, its reserved memory is also idle and wasted during a device's idle time. Falaki et al. revealed that the time for video playback is only about 8% of a total smartphone operation time in a day, and the time taken by using camera is also negligible [Falaki et al. 2010]. Accordingly, the time during which the reserved memory is used is less than 10% of the total time. The memory reservation, however, prevents a

system from exploiting the wasted memory during the idle period of the devices. This phenomenon stems from the fact that such an embedded system is becoming a general-purpose system such as smartphone and smart TV. Most previous embedded systems, such as a portable media player or an MP3 player, have not experienced this problem since they are single-purposed. Modern general purpose systems provide various usages, such as web-browsing, reading news, communications, and games. As a result, the waste of the memory becomes more serious due to the increased device-idle time in resource constrained embedded systems.

Hardware-level support is one of ways to minimize or eliminate the inefficiency of memory reservation. Intel DVMT [Intel 2005] and AGP Gart (graphic address translation table) support on-demand memory allocation and I/O address mapping for graphic devices. IOMMU (I/O memory management unit) is a more general form of on-demand I/O address mapping. These approaches, however, are limited to a special device, such as graphic device, making itself hard to be applied for other cases. Many IOMMU implementations are also limited to high-end server systems [AMD 2011; Abramson et al. 2006; Dong et al. 2008].

Hotplug Memory [Schopp et al. 2005] is one of possible software-level approaches although its main purpose is to add a physical memory package into a system on the fly. When an integrated device is going to be idle, its reserved memory can be virtually hot-plugged into a system. When the device is activated, its reserved memory is virtually unplugged from the system and returned back to the device. During the idle time, the OS kernel uses its virtually hot-plugged memory. We will denote this virtually hot-plugged memory as *rental memory* since the OS kernel borrows the reserved memory during the device-idle time. The main advantage of this concept is that neither hardware redesign nor increase in unit cost is needed because all work is done at the software-level. This approach, however, suffers from a long latency when the rented memory is returned back due to its complex operations. According to our measurement, the latency is up to tens of seconds, which is not acceptable for most applications.

In this paper, we propose a rigorous rental memory (REM) management scheme. The main goal of this rigorous management is to minimize the time taken for the return operation, or *return time* for brevity, by carefully controlling the rental memory. This goal is important in terms of the transparency of a system because the minimized return time will provide unrecognizable application launch latency to an end user. While providing the transparency, a system can achieve better performance due to increased available memory during a device-idle period. Once the device is activated, the system shows the same performance as the system without our scheme.

In order to ensure rigorous rental memory management, it is crucial to make careful decisions on which data to be placed in the rental memory, and how to control the rental memory. Though there are many ways of using the rental memory in the OS kernel, an improper use of the rental memory will incur significant delay during the return operation. For example, placing dirty data in the rental memory requires write-back I/Os during the return operation. Placing kernel metadata, such as task structure or memory descriptor, is impossible since such structure is unmovable in general OSes [Schopp et al. 2005]. If such structures are in the rental memory, the device fails to get back its reserved memory, or the device will cause system failure by corrupting the structures. Anonymous pages, such as process stacks and heap pages, also can be an obstacle of minimizing the return time in swapless embedded systems. Briefly, our main idea is to place volatile data in the rental memory. Volatile data means a memory page whose content can be promptly discarded without corrupting the correctness of any computation. In addition, we introduced two additional techniques, *lazy-migration* and *adaptive-activation*. The lazy-migration technique alleviates the

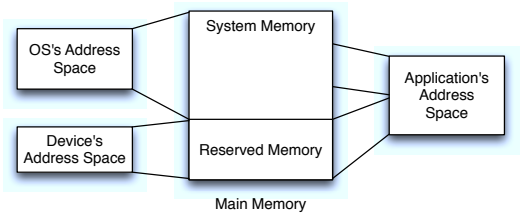


Fig. 1. The memory structure of target hardware

Table I. Summary of reserved memory in four Android smartphones and one Android development board. Two ratios of reserved memory are calculated by dividing a reserved memory size by a total memory size and an OS available memory size (*MemTotal* in `/proc/meminfo`) respectively.

Device	Total	Available	Reserved	Usage
Nexus One	512MB	385MB	81MB(16%/23%)	MDP, video decoder, camera
GalaxyS	512MB	334MB	115MB(22%/34%)	video decoder, camera, etc.
NexusS	512MB	386MB	105MB(21%/27%)	video decoder, camera, etc.
GalaxyS2	1024MB	816MB	154MB(15%/19%)	video decoder, camera, JPEG, etc.
Odroid7(our target)	512MB	283MB	149MB(29%/52%)	video decoder, camera, JPEG, etc.

lowered utilization of the rental memory caused by the volatile page placement policy. The adaptive-activation technique preserves active pages in the rental memory when the return operation is performed without hurting the transparency of the system.

We implemented our scheme on Android(Froyo)-based embedded system hardware named Odroid7 with the Android Linux kernel 2.6.32.9. The target hardware is the same platform as the Samsung GalaxyS smartphone. We evaluated our scheme with various workloads. The evaluation results have shown that our scheme minimizes the return time to under 1 second in various workloads. We believe that the time can be hidden in the application initialization time and is acceptable to end-users. In addition to the minimized return time, our scheme naturally enhances the performance of the system during the idle period of a device compared to the system without our scheme.

The rest of this paper is organized as follows. The next section describes the background of our work and also states related work. Section 3 depicts the rental memory management, the base of our scheme, and also provides the motivation of this work with preliminary evaluation results. Section 4 shows the detail of our rigorous rental memory management. We provide implementation issues for our prototype in Section 5. Section 6 presents the evaluation results. The last section concludes this paper with some discussion of this work.

2. BACKGROUND AND RELATED WORK

2.1. Background

Our target system generally splits its main memory into two regions. One is a system memory region, which is referred to as *normal memory* in the rest of this paper, for the OS kernel and the other is a reserved region for integrated device(s) as shown in Figure 1. The kernel reserves a certain amount of memory for integrated device(s) at booting time since it may not be possible to allocate the required contiguous memory space during runtime. The amount and the address of a reserved memory space is usually determined at a system design phase and is statically fixed. The Android platform also supports such contiguous memory reservation in the form of *pmem*.

Table I shows the state-of-the-art Android-based smartphones with their reserved memory sizes. All smartphones in the table are equipped with various integrated devices, such as video decoder, camera device and JPEG decoder. The sum of available

memory and reserved memory is different from total memory in each system since we excluded some amount of memory that is invisible for an unknown reason. Another amount of memory which is permanently used by 3G modem device is also not taken into account. The portion of each reserved memory reaches from 16% to 21% of a total memory; Odroid7 is excluded in this calculation because it is a development board. Compared to available memory, the reserved portion increases reaching up to 34% of OS available memory.

In order to use device's reserved memory, we need to define a device-idle time. We first show how an integrated device interacts with its reserved memory.

(1) When an application, which provides a function of an integrated device, is started, it opens a device file of the device to begin an interaction in Unix-like OSes.
(2) The application makes a `mmap()` system call to open a communication channel between the integrated device and the application. In this case, the kernel provides physical pages of the device-reserved memory space to the mmap-requested address range as shown in Figure 1.
(3) The application communicates with the device through the mmaped pages. For example in a multimedia player, the application loads an encoded multimedia stream onto the mmaped pages, and an MPEG decoder hardware decodes the multimedia stream and passes decoded data to either the application or to a video frame buffer directly.
(4) When the application is exiting, it closes the opened file descriptor. This indicates that that the device ends its operation and is going to be idle.

The period between (1)-(4) is an active state of the device and the reserved memory is used for a function of the device. We define this period as a *device-active* time. Besides the device-active time, the reserved memory is not accessed by anyone, and this period is called a *device-idle* time.

A device driver is in charge of renting its reserved memory to the OS kernel. Since a device driver knows a semantic of its device's behavior, it knows which physical pages can be a candidate of rental memory in a device-idle time. In some cases, a device driver stores device-specific data in some physical pages in its reserved memory. For example in our implementation, a video decoder puts firmware codes in a few physical pages in its reserved memory. Although such physical pages are not accessed by a device during idle, we excluded such pages from rental candidates in order to avoid any data corruption.

During a device-active period, the kernel usually disables CPU caching of a reserved memory region for memory consistency because a CPU and an integrated device access the memory simultaneously. If our scheme is used, however, the reserved memory region is solely accessed by a CPU during a device-idle time. In this case, we enable CPU caching again to gain benefits from CPU caching.

2.2. Related work

In this section, we first describe previous approaches that provide a contiguous memory region to a device. Then, we explain the details of Hotplug Memory [Schopp et al. 2005].

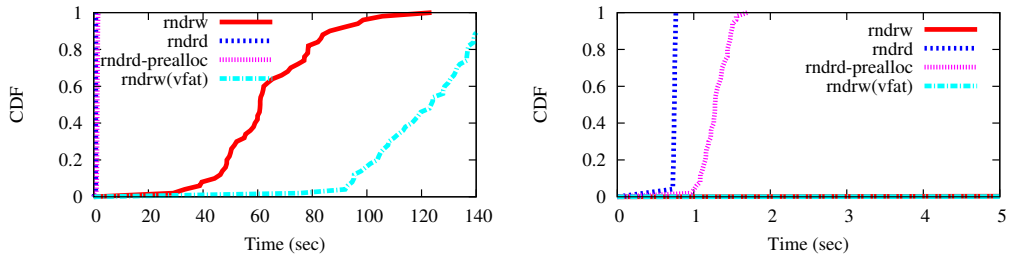
2.2.1. Device Memory Provisioning. Many devices in embedded systems do not provide scatter-gather DMA or I/O mapping support while the device(s) require contiguous memory region(s) to operate. Camera devices and video decoders, for example, require contiguous memory regions of megabytes, and the reservation scheme is a simple and easy answer to it, at the expense of low memory utilization.

In order to eliminate the inefficiency of memory reservation, many graphic devices, such as AGP Gart and Intel DVMT [Intel 2005], allow on-demand memory allocation. The discontiguity of allocated pages is covered by I/O mapping between their physical addresses and device's I/O addresses. IOMMU is a general form of supporting such on-demand I/O mapping for other devices. The problem of this I/O mapping approach is that they are hardware-dependent. Graphic devices are specially designed to support I/O mapping, and IOMMU approach requires platform-specific hardware such as Intel VT-d [Abramson et al. 2006], AMD's IOMMU [AMD 2011], or PCI standard-SR-IOV devices [Dong et al. 2008]. Therefore, a system needs to be redesigned in order to be integrated with I/O mapping features. Many state-of-the-art smartphones still do not provide such I/O mapping features for video decoding devices or camera devices as shown in Table I. Another problem of the I/O mapping approach is high CPU overhead for mapping management. Ben-Yehuda et al. have revealed that IOMMU requires an additional CPU time of from 15% to 30% for network I/O [Ben-Yehuda et al. 2007]. While various techniques could minimize such CPU overhead [Yassour et al. 2010; Willmann et al. 2008], the costs for additional hardware and system redesign will probably increase unit cost and the time-to-market of a system.

A software-level approach to minimize the memory inefficiency is Contiguous Memory Allocation (CMA) [Corbet 2010]. It provides a customizable and modular framework to manage device memory allocation. The main difference from memory reservation is that CMA allows sharing of reserved memory regions. For example, a video decoder requires 10MB of contiguous memory and a camera device requires 8MB of contiguous memory. If both devices never operate concurrently, they can share one physical memory whose size is 10MB; a simple reservation approach requires 18MB of contiguous memory in this case. Therefore, CMA provides a flexible memory reservation for various types of systems and can reduce the amount of reserved memory in a system. This work is complementary to our work since the 10MB of the reserved memory can also be rented to the OS kernel while both devices are idle.

2.2.2. Hotplug Memory. The concept of rental memory is very similar to Hotplug Memory [Hansen et al. 2004; Schopp et al. 2005] since both approaches increase usable memory from the perspective of software. When a physical memory package is added on a system board, the OS kernel initializes a kernel memory allocator to service page allocation from the hot-added physical pages. In order to be aware of physical memory removal (or unplugging), the hot-added region is only given to movable pages. A movable page can be migrated to another physical page by copying its content and redirecting its page table mapping. Movable pages include process stack, heap, and page cache pages [Schopp et al. 2005]. When the hot-added memory is unplugged, the movable pages in its region are either migrated into another memory region or dropped if possible based on the importance of each page [Schopp et al. 2005]. Active pages are forced to be migrated while inactive pages are written-back (if dirty) and discarded. Since Hotplug Memory is not designed to minimize the time taken for unplugging, it shows long unplugging latency. We will show this result in the next section.

A later version of CMA [Nazarewicz 2010; Corbet 2011] is a convergence of CMA and Hotplug Memory. In this version, a reserved memory region is serviced by the kernel page allocator for movable or reclaimable pages when the region is not used by any devices; reclaimable pages store reclaimable kernel structures such as inode caches. When the device wants to use the memory, the pages in the region are moved and/or evicted to make room for the device. This, however, is also not designed to minimize the time to make room for the device. Accordingly, this approach is also limited by the same reasons as Hotplug Memory from the perspective of the transparency.



(a) X-axis from 0 to 140 second

(b) X-axis from 0 to 5 second

Fig. 2. CDFs of return times of various workloads

3. MOTIVATION

In this section, we first explain our rental memory management. Fundamentally, it is an abstraction of OS-level support to minimize the inefficiency of memory reservation. Then, we provide simple rental memory management which follows the policy of Hotplug Memory. We also show the problem of the simple rental memory management with preliminary evaluations.

3.1. Rental Memory Management

From the perspective of the OS kernel, it borrows a device's reserved memory while the device is idle. When the device is being activated, the kernel is fully in charge of vacating the borrowed memory region and returning the region to the device. We denote this borrowed memory as rental memory. In order to achieve this semantic, we require interactions between the OS kernel and a device driver. For the interaction, the kernel provides two interfaces, which are `rent_memory()` and `return_memory()`; two interfaces are denoted as a *rent operation* and a *return operation*, respectively. As we described in Section 2.1, a device driver calls `rent_memory()` when its device file is closed. Contrary to this, a device driver invokes `return_memory()` when the device file is opened. As the role on the device driver side is relatively simple, our prototype requires about 10 lines of additional codes per driver to support rental memory management. We believe that the cost to apply our scheme to other legacy or new device drivers is minimal.

When the kernel is given the information of rental memory region(s) through the rent operation, it first puts the physical pages in the rental memory region(s) into a rental memory pool, which is initialized as empty at booting time. The pool is separated from the pre-existing pool (also denoted as a normal pool) in order to differentiate page allocation policies between the two pools. When the return operation is invoked, the kernel vacates all physical pages which are specified in the function. Since the physical address of the rental memory is fixed in the device driver, the kernel has to free all physical pages specified by the function. After the specified rental memory is freed, the kernel returns the memory back to the device.

3.2. Problems of Simple Rental Memory Management

Since the return operation requires reclaiming all physical pages in the rental memory, we need to make a page allocation policy to place reclaimable pages in rental memory. Hotplug Memory admits movable pages in their hot-plugged memory. This is a good starting point to establish rental memory management. Therefore, the simple rental memory management follows the Hotplug Memory's policy described in Section 2.2.2. The return operation in this simple management also follows the same mechanism as in Hotplug Memory as described in Section 2.2.2.

In order to reveal the problem of simple rental memory management in terms of return time, we conducted a simple measurement on our evaluation system. The term 'return time' indicates the time taken to complete the `return_memory()` function. The evaluation system is equipped with a 1GH ARM cortex A8 processor. Main memory has around 150MB of free memory, and the size of the rental memory is 105MB. We ran Sysbench file I/O on 100MB of files. We disabled a periodic fsync() operation to make the workload more memory-intensive than disk-intensive. We intentionally made all page caches in order for the files to be placed in the rental memory. During the workload running, we invoked `return_memory()` at random time, and measured the return time 50 times. Figure 2(a) shows the CDFs of various workloads. In the figure, *rndrw*, random reads/writes took an average of 64 seconds for return operations due to the write-back operations of dirty pages; other workloads are also explained in Section 4.2 and 4.3. The main reason for this significantly large latency is due to the fact that the policy of Hotplug Memory is not designed to minimize the return time.

4. RIGOROUS RENTAL MEMORY MANAGEMENT

In this section, we explain our rigorous rental memory management. We first describe our policy which controls the rental memory in order to satisfy our goal. We then provide additional techniques to enhance our scheme in terms of memory efficiency.

4.1. Page Allocation Policy

Since our main goal is to minimize the return time, carefully establishing a page allocation policy is crucial because the characteristic of allocated pages in the rental memory will directly determine the return time. In our scheme, page cache (file cache) pages are the only page type to reside in the rental memory. This is referred to as a page cache-only policy. While Hotplug Memory (and simple rental memory management) allows anonymous pages along with page caches, we exclude anonymous pages from our page allocation policy. This policy is specialized to swapless embedded systems. Although many researches have studied flash-based swap systems [Saxena and Swift 2010; Jung et al. 2005; Park et al. 2004], most commodity embedded systems have no swap storage in practice. Without swap storage, placing anonymous pages in rental memory could lead to out-of-memory when the return operation is performed. Since the rent operation increases available memory size, the amount of anonymous pages could be larger than the size of the normal memory if a current workload is anonymous page-intensive. Once the return operation is performed, we cannot deal with the overcommitted anonymous pages without swap storage. In order to avoid this phenomenon in advance, we need to limit the amount of anonymous pages under the normal memory size. Therefore, our policy is to disallow anonymous page allocation from the rental memory.

The importance of page cache is well known in high-end servers where the occupancy of page cache is above 50% of total memory [Lee et al. 2007; Ding et al. 2011]. Our main concern is whether page cache is still important in embedded systems. We believe that the emerging embedded software such as pocket cloudlets [Koukoumidis et al. 2011] will benefit from the increased memory specialized for page caches. Our evaluation result also showed that the page cache-only policy brings performance improvement in Android-based workloads in Section 6.5.

Since the rental memory only admits page caches, the normal memory could have higher utilization, lower free page ratio, than that of the rental memory when a workload is anonymous page-intensive. For an example scenario, before the rent operation is called, the normal memory is almost full of page caches. After the rental memory is added, a workload now requires only anonymous pages. In this case, the normal memory is busy reclaiming physical pages to handle the anonymous page allocation

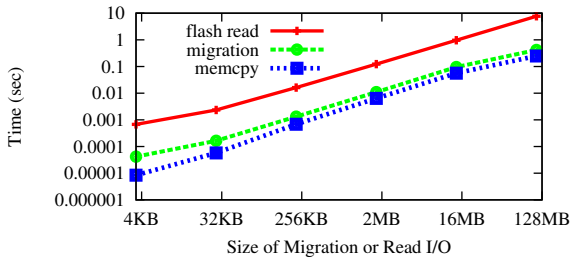


Fig. 3. The costs of page migration and read I/Os with varying the amount of pages to be migrated or to be read from flash storage

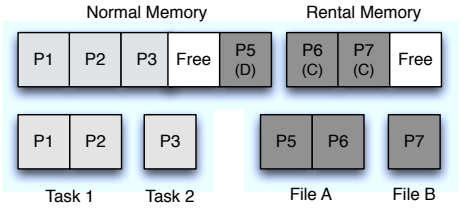


Fig. 4. An overview of rigorous rental memory usage. D and C denote dirty and clean respectively. Task 1 and 2 have their stack and heap pages in the normal memory. File B has its page in the rental memory while File A has two pages in two memory regions, one clean page P6 in the rental memory region and one dirty page P5 in the normal memory region.

while the rental memory has many free pages. Thus, the page cache-only policy incurs biased memory utilizations of two regions. At worst, some page caches in the normal memory should be evicted to handle a new page allocation and then will probably be reloaded if the pages are still in a part of the working set.

To resolve this biased utilization problem, we adopt a *lazy-migration* technique. When a page cache in the normal memory is a target of eviction for anonymous page allocation, we migrate the page into the rental memory instead of evicting the page. Note that the page replacement algorithm generally chooses clean pages for eviction. If a dirty page is the target of eviction, the algorithm delays the dirty page eviction until the page is written back. The detailed page migration procedure is described in the literature [Schopp et al. 2005]. In the example scenario, the lazy-migration eliminates the additional read I/Os and instead requires a small number of memory operations for migration. The migration costs for a group of pages are about 13 times lower than that of read I/Os for the same amount as shown in Figure 3. Some pages that are read once and never read again are improper targets of this lazy-migration. Such pages can be filtered by borrowing ideas in previous work [Chen et al. 2005], but we did not implement such optimizations in this work.

4.2. Placement Policy

The page cache-only policy is insufficient for minimizing the return time of rental memory since the page cache itself has two states, clean and dirty. Although the two states are interchangeable, the effect of dirty page is significant in terms of the return time. Our preliminary experiment in Figure 2(a) revealed the problem of the dirty page. In the figure, rndrw has shown tens of seconds of return time while *rndrd* (random read) have shown the time under 1 second. Accordingly, we also restrict the page placement in the rental memory to clean page caches. We denote this policy as a clean-

only policy. The following two paragraphs detail the two reasons to exclude dirty page from the rental memory.

First, the page lock held by its write-back operation is one of the obstacles of minimizing the return time. The OS kernel tries to write-back dirty pages if the number of dirty pages is beyond a pre-defined threshold [Love 2010]. If dirty pages in the rental memory are in the middle of write-back operations, the return operation is delayed until the write operation is completed. Note that any operation on the writing-back page should wait until the I/O is completed.

The other reason is the dirty page itself. In some cases, dirty pages in the rental memory can be migrated into the normal memory if write-back operations for the dirty pages are not yet issued. It, however, cannot be guaranteed that enough room, free or clean physical pages, in the normal memory exists to place the dirty pages at any given time. If the system is write-intensive at some time, many dirty pages in the rental memory probably indicate many dirty pages existing in the normal memory too. This situation inevitably requires write-backs of dirty pages. In addition to this, some file system implementation does not provide dirty page migration. For example in Linux, dirty pages of VFAT should be written-back first, and then the pages can be migrated. This is the reason of the longer return time of *rndrw(vfat)* in Figure 2(a).

The clean-only policy also could cause a biased utilization problem if a system is highly write-intensive at a given time. In order to cope with this problem, we added a new condition to the lazy-migration. While the rental memory has free pages, a target of eviction in the normal memory also can be migrated into the rental memory if the target is a clean page cache.

An overview of the rigorous rental memory management is shown in Figure 4 with anonymous page and page cache examples.

To cope with the clean-only policy, we need to modify page cache allocation paths. In most OSes, there are two ways for page cache allocation. One is through explicit file access operations such as `sys_read()` and `sys_write()`. The other is through implicit file access using `sys_mmap()`. In the explicit allocations, if the `sys_write()` requires a new page allocation, we provide a page from the normal memory pool because this path explicitly indicates that the page will be dirtied. In the `sys_read()` path, we have no idea whether the page will be dirtied or not. Accordingly, we provide a page from the rental memory pool. When the `sys_write()` is invoked upon a page which is already read and in the page cache, we adopt a copy-on-write scheme to satisfy the clean-only policy on rental memory. We allocate a new page from the normal memory, copy data onto the page, and continue the write system call on the new page. Memory mapped file I/Os bypass the above two system call paths. To handle this case, we disable write permissions for the pages which are mapped by the `sys_mmap()` system call. When a write occurred on the write-disabled page, it will cause a page fault exception and we apply copy-on-write to the written page.

4.3. Adaptive Activation

In order to minimize the return time, discarding all clean pages in the rental memory is our basic return operation. It requires only a few memory instructions to release clean page caches.

This, however, could result in memory inefficiency if active pages in the rental memory are discarded. After the return operation, the active pages may be read again from backing storage. Hotplug Memory migrates active pages in the hot-plugged memory into the normal memory during unplugging [Schopp et al. 2005]. Although this activation increases overall system efficiency, it stands against our main goal because a large number of migration operations will probably increase the return time.

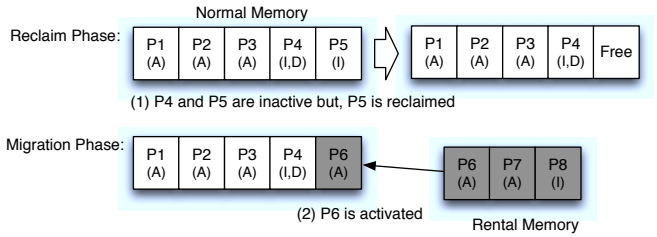


Fig. 5. Overview of the adaptive-activation. A, I, and D indicate active, inactive and dirty, respectively. The normal memory has two inactive pages P4, P5. The reclaim phase evicts P5, and the migration phase moves P1 into the free page.

In more detail, Hotplug Memory migrates all active pages in the hot-plugged memory into the normal memory even if the normal memory has insufficient room to place the active pages. Therefore, memory contention in the normal memory could cause a large number of page reclamations. Figure 2(b) is the same figure as Figure 2(a), but the x-axis is scaled. In the figure, *rndrw-prealloc* indicates that random read/write operations are performed with 128MB of anonymous page pre-allocation. Due to the pre-allocation, normal memory has around 13MB of free memory. In this case, the return operation tries to migrate 66MB of active pages from the rental memory to the normal memory. Since the normal memory has insufficient free pages, the migration increases memory allocation contention in the normal memory. After 13MB of free pages are filled up, following page migrations still need to allocate 53MB of pages in the normal memory. The page allocation requests continue until the return operation is finished. Due to the flood of page allocation requests, a page replacement algorithm repetitively scans physical pages in the normal memory and reclaims the pages. In the end, already migrated page caches are also evicted to make room for the following migrations. This page allocation contention is the main source of the increased return time.

In our scheme, we require a more sophisticated approach to achieve both memory efficiency and our goal. The main reason for unpredictable delay is that the unplugging in Hotplug Memory does not consider the state in the normal memory. To alleviate this unpredictable delay, we propose the *adaptive-activation* technique that is performed prior to discarding all pages in the rental memory. The main advantage of this technique is that it foresees the state of the normal memory and decides how many migrations are appropriate.

In more detail, the adaptive-activation consists of two phases, a reclaim phase and a migration phase. In the reclaim phase, it scans each physical page in the normal memory once. If each physical page is inactive and can be reclaimed immediately, it reclaims the page. Otherwise, it continues to the next physical page. If the number of reclaimed pages exceeds the number of active pages in the rental memory, it ends the reclaim phase.

In the migration phase, we can have two conditions. One is that the number of reclaimed physical pages exceeds the number of active pages in the rental memory. This is an optimal case, and all active pages in the rental memory can be migrated to the normal memory without long delay. The other condition is an insufficient number of physical pages reclaimed during the reclaim phase. This case indicates that additional page reclaim requires a long delay such as write-back I/Os. Accordingly, we discard some of the active pages in the rental memory. Partial discarding is still better than discarding all pages in terms of memory efficiency. In Figure 5, we have two inactive pages but our reclaim phase collects page N5 only. Because N4 is inactive and dirty,

N4 is not reclaimed. Therefore, one active page R1 in the rental memory is moved into the normal memory. When the available room in the normal memory is decreased due to a sudden page allocation requests, we discard more active pages.

Since our adaptive-activation avoids memory contention in the normal memory, it takes an almost deterministic time to complete its operations. The reclaim phase needs to scan every physical page once, and requires memory operations to release reclaimed physical pages. This is bounded by the maximum number of physical page reclamations in the normal memory. In addition, the migration phase works with clean page caches. Accordingly, this is also bounded by the maximum page migrations in the rental memory.

5. IMPLEMENTATION

We implemented our scheme on Android Linux kernel 2.6.32.9. We modified and used codes for memory hotplugging and page migration in Linux and Hotplug Memory [Schopp et al. 2005]. Linux has a memory hotplugging feature for the system providing memory sections; a memory section is a unit of memory (e.g., 16MB for PowerPC and 1GB Itanium) and also a unit of memory hotplugging. Since device reserved memory regions are not aligned to a memory section and have various sizes, we modified the memory hotplugging code to support the variable lengths of memory chunks for plugging and removal. For copy-on-write, we made a new interface that migrates a single page.

In Odroid7, our target platform, we modified three device drivers to support rental memory management. The drivers are for a MPEG decoder, a camera device, and a JPEG decoder. The sizes of reserved memory for each device are 72MB, 33MB, and 8MB; the three devices' reserved memory is 74% of system's total reserved memory. In our implementation, the three drivers require a significantly lower engineering effort (around 10 lines of code addition per driver) to support our rental memory management. We also believe that the manufacturers of embedded systems could easily modify their device drivers to support the rental memory management.

The rental memory pool is managed as `ZONE_REM` that is initialized as empty at booting time. A buddy system allocator, a Linux default page allocator, provides page allocation on the `ZONE_REM`. Linux also has a per-zone-based page replacement policy and manages a per-zone approximated LRU list. Our adaptive-activation works with the active and inactive pages in the two zones.

In Linux, a page cache page is populated by a single function `__page_cache_alloc()` which eventually calls the buddy system allocator, `alloc_pages()`. We modified the page cache allocation function to specify a flag of page cache allocation to the buddy system allocator. In the allocator, a new page request with the specified flag is serviced on the `ZONE_REM` first. If the `ZONE_REM` is full and the `ZONE_NORMAL` has free pages, then the request is forwarded to the `ZONE_NORMAL`.

For our copy-on-write scheme, we added a new page cache allocation function `__page_cache_alloc_write()` that is called in the middle of `sys_write()` related functions. The allocation function skips a new page allocation from the `ZONE_REM`. For the copy-on-write scheme, when the write system call tries to update a page in `ZONE_REM`, we migrate the page to the `ZONE_NORMAL` and proceed to update the page.

6. EVALUATION

Our evaluation is performed on the Odroid7 Android system, which is equipped with the Samsung S5PC110 platform. The target system has a 1GHz ARM cortex A8 processor, 512MB RAM and 4GB T-Flash storage. We conducted our evaluation on the VFAT file system, which is the default file system of an sdcard partition. Some evaluations in Section 6.4 were performed on the EXT3 file system.

We evaluated our prototype with various embedded system workloads and synthetic workloads. Five workloads, djpeg, lout, madplay, tiff2bw, tiff2rgba are from *mibench* embedded system benchmark [Guthaus et al. 2001]. *Convert*, a prevalent image editing workload, performed rotation and normalization of 30MB-sized bitmap image file and stored an output file in a JPEG format. One synthetic workload is *Sysbench file I/O* [Kopytov 2004]. In Sysbench, we varied the file set size to adjust its working set size and used two file I/O modes: random read and random read/write. The notation *sysbench300rw* indicates a Sysbench workload with random read/write on 300MB of file set. We disabled a periodic fsync() operation and warmed up page caches with the input files before every execution in order to make the workload more memory-intensive than disk-intensive. The random read/write workload executes 30000 I/O 16KB-requests with 1.5 reads/writes ration. The random read workload performs 1.6 million read requests in 90 seconds. At the end of this section, we used various Android real world applications to figure out the effect of our scheme in a real environment.

Our system has around 150MB of free memory. We actually used two devices' reserved memory; the devices are the video decoder device and the camera device. The total size of the two reserved memory regions is around 105MB. We omitted the use of the JPEG decoder's reserved memory because the two devices' reserved memory is sufficient to show the effect of rental memory. Except for Section 6.5, a rental memory region (or the rental memory) indicates both reserved memory.

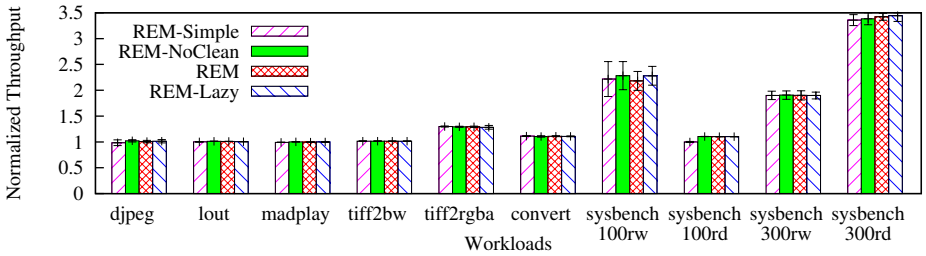
The evaluation is focused on measuring the following metrics.

— Performance enhancement by REM. The main reason for rental memory management is to enhance the performance of a system during a device-idle period. Accordingly, performance improvement is one of the main concerns of this evaluation.
— Overhead of copy-on-write. Due to the clean-only policy, when a write occurred on a physical page in rental memory, copy-on-write overhead is inevitable. We characterized the overhead caused by copy-on-write in our evaluations.
— Effects of the lazy-migration and the adaptive-activation. Two techniques address some performance losses caused by the clean-only policy and the return operation respectively. We measured the effects of the two techniques.
— Return time. The main goal of rigorous rental memory management is to minimize the time taken for the `return_memory()` operation. This is important for the transparency of a system. We measured return times under various workloads.

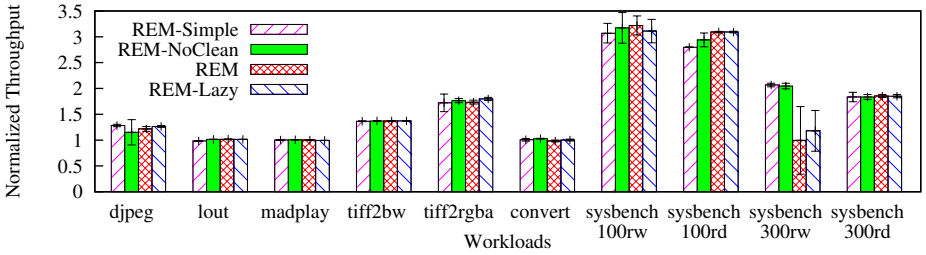
6.1. Performance

Firstly, we measured the throughput of various workloads. Obviously, memory-intensive workloads will show better performance by exploiting an additional memory. Figure 6(a) shows the throughputs of all workloads. All throughputs are normalized to those without rental memory. REM-Simple indicates that the rental memory is used under the simple rental memory management policy. REM-NoClean denotes that the rental memory also places dirty page caches too. REM indicates that the rental memory is used under the rigorous management scheme without the lazy-migration. REM-Lazy includes the lazy-migration. From the four legends, we can figure out the effect of each policy in more detail. Each value is an average of 20 samples.

As shown in the figure, tiff2rgba, and Sysbench workloads have shown throughput increases by 30% - 250% with the rental memory. In cases of Sysbench with the 300MB file set, the performance benefits mainly come from an increase of available memory; the workloads cache more data in memory. The other workloads, tiff2rgba, convert, and sysbench100rw, gain the throughput increase because of a different reason. Although they are memory intensive, their working-set size can be fit into the free pages in the normal memory.



(a) Normalized throughputs



(b) Normalized throughputs with 64MB of memory pre-allocation

Fig. 6. Performance impact of the rental memory with using four schemes

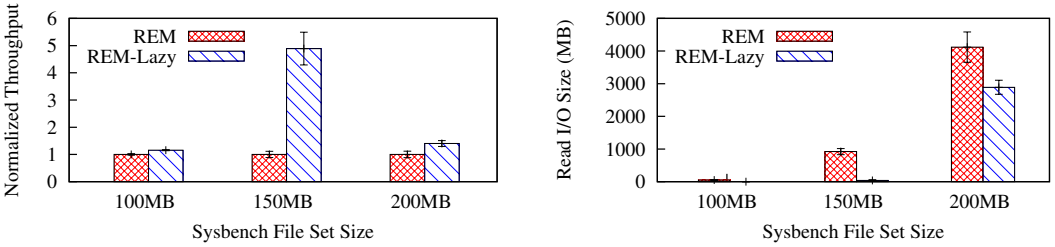
The throughput improvement comes from the characteristic of a dirty page write-back heuristic in Linux. It keeps a dirty ratio below a specified threshold, 0.2 in our evaluation. The ratio is the number of dirty pages divided by the number of dirtyable pages, which consist of clean pages and free pages. If the dirty ratio is above the threshold, *pdflush* kernel thread writes dirty pages back to a backing storage. The main purpose of this heuristic is to preserved some amount of promptly reclaimable pages [Love 2010]. When a page cache is clean, the kernel can reclaim the page immediately. If dirty, the kernel cannot do that at once.

Once available memory increased due to the rental memory, the write-back thread is less frequently woken up because it is given additional dirtyable pages. Therefore, the applications that perform file writes have shown throughput improvement as shown in Figure 6(a).

In a general purpose embedded system, it is not easy to have a large amount of free memory in runtime. Accordingly, many workloads could face a more memory-constrained situation. In order to reflect such an environment, we synthetically give more memory pressure to the system. We ran a task that allocates around 64MB of anonymous pages in the background and then conducted the same workloads. Figure 6(b) shows the normalized throughput of the same workloads with the memory pre-allocation. Two workloads, djpeg and tiff2bw, improved their throughput under high memory pressure. tiff2rgba and other sysbench workloads have shown more improved throughput than that without the memory pressure. The sysbench100rd workload also has shown increased throughput since the reduced free memory in the normal region cannot cover its working set. Notable results are REM and REM-Lazy cases in the sysbench300rw workload. In the REM case, the throughput is the same as the normal case (without the rental memory). Since the normal memory is only dirtyable, a large amount of writes are bounded by the memory contention in the normal memory. The REM-Lazy case increases its throughput by 17% because the lazy-migration reduces the read I/Os for the page caches that are evicted from the normal memory by the memory contention.

Table II. Summary of copy on write overhead: Increased CPU utilization, jiffies, and the number of copy on write with the clean-only policy. A value in parenthesis is the difference between REM-NoClean and REM.

	CPU utilization (%)		CPU cycle (jiffies)		CoW size (MB)
	REM-NoClean	REM	REM-NoClean	REM	
sysbench100rw	10.9	11.6 (0.7)	626	698 (72)	82
sysbench100rw w/ prealloc	9.1	9.9 (0.8)	910	976 (66)	81
sysbench300rw	9.0	9.4 (0.4)	1706	1789 (82)	60.21
sysbench300rw w/ prealloc	7.3	5.5 (-1.8)	2201	2465 (264)	59.41



(a) Normalized throughput (b) Additional flash read size

Fig. 7. Effects of lazy-migration in Sysbench workloads with varying file set size

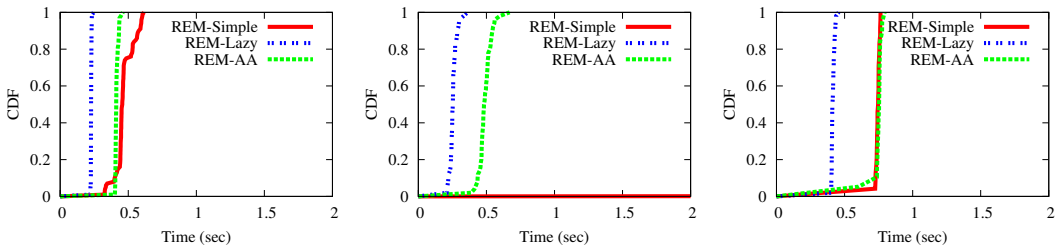
6.2. Copy-on-write Overhead

Since copy-on-write occurs when the page cache page is written after read in the rental memory, we measured the additional costs caused by copy-on-write in Sysbench random read/write workloads. Table II shows CPU utilization, CPU cycle, and the copy-on-write size occurred in each workload. The value in each parenthesis is the difference between REM-NoClean and REM. The additional CPU costs in utilization are 5%-7%. The additional CPU costs in cycles are also 5%-12%. In case of Sysbench 300MB with pre-allocation, the overall CPU utilization has been decreased even though copy-on-write occurred. This result comes from the fact that the writeable page cache size is significantly smaller than the file set size. Therefore, frequent write-backs make CPU wait for the write-back completions, and the CPU utilization is decreased. The CPU cycle is, however, increased by 12% in this case.

Other workloads such as convert and the workloads in mibench, have shown little copy-on-write overhead as shown in the previous two figures. Since an output file creation is dominant writes in the workloads, the page caches for the writes are allocated from the normal memory.

6.3. Effects of Lazy-migration

Although the throughput improvement by the lazy-migration has been shown in the sysbench300rw workload in Figure 6(b), the workload does not give all of the advantages of the lazy-migration. In order to show the effect of lazy-migration in more detail, we designed the following synthetic evaluation. We made the normal memory warmed up with input files, and the rental memory empty. After we started Sysbench benchmark, we ran a memory hogging task which gradually increases its anonymous page allocation up to 128MB. Therefore, the normal memory is busy handling anonymous page requests while the rental memory has many free pages. Figure 7 shows the



(a) Convert (b) sysbench100rw (c) sysbench100rd

Fig. 8. CDFs of return times of three workloads

throughputs of these workloads in both REM and REM-Lazy cases while varying the file set size. The throughput value is normalized to that of REM case.

In the case of the 100MB file set size, while anonymous page allocation is increasing, the page caches are evicted and reloaded into the rental memory in the REM case. REM-Lazy, however migrates the page caches from the normal memory to the rental memory. Therefore, no additional read I/O is needed in REM-Lazy as shown in Figure 7(b). The increase in throughput of REM-Lazy is about 15% over that of REM case in 100MB file set size. In case of 150MB file set size, the throughput is increased by 400% compared to the REM case. While the amount of allocated anonymous page is increasing, the memory contention for file caches occurs because the size of page cache is relatively decreasing. Therefore, the contention incurs frequent file data caching and eviction. Figure 7(b) shows that the required read I/O in REM case is about 4 times larger than that in REM-Lazy case. We believe that this case represents one of the worst cases of the biased-memory utilization. The lazy-migration technique alleviates the biased-memory utilization, and thus make unnecessary read I/Os reductions. In the case of the 200MB file set size, the lazy-migration gains throughput increase by 40%. Since both 200MB of file caches and 128MB of anonymous pages cannot be placed in the two memory regions, file caches are loaded and evicted frequently in both cases. We can figure out that the absolute read size of the 200MB case is significantly larger than that of the 150MB case.

6.4. Return Memory Operation

In this evaluation, we measured the return times of three rental memory management schemes: REM-Simple, REM-Lazy, and REM-AA. REM-Simple provides Hotplug Memory's scheme. REM-Lazy discards all pages in the rental memory. REM-AA is REM-Lazy with the adaptive-activation.

Figure 8 shows the CDFs of return times of three workloads: convert, sysbench100rw, and sysbench100rd. We varied the time at which the return operation is conducted. We measured 50 samples of return operation times. In all cases, REM-Lazy shows the shortest return time in each workload. REM-AA requires more time to perform the adaptive-activation. Figure 8(a) shows that the return time of REM-Simple is comparable to that of REM-AA since the convert workload has enough room in the normal memory to place active pages in the rental memory. The workload also did not generate dirty pages in the rental memory. In Sysbench workloads, REM-Simple requires an order of magnitude larger time for the return operation and its graph line is near the bottom of Figure 8(b). The average return time is 64 seconds in REM-Simple. Dirty pages generated by the workload are the main source of this large delay. In REM-Lazy, the return operation time has shown under 0.4 seconds in all workloads. Our adaptive-activation technique additionally increases the return operation time

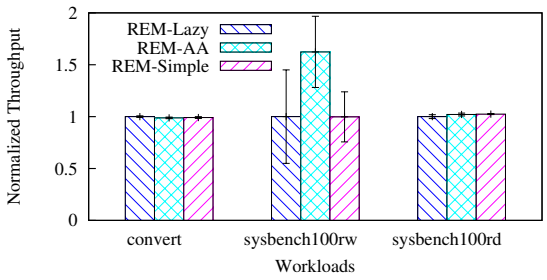


Fig. 9. Throughputs of three workloads with the return operations

at most 0.3 seconds. In sysbench100rd, REM-AA and REM-Simple have shown much similar return operation times as depicted in Figure 8(c).

In order to figure out the effect of the adaptive-activation, we also measured the throughput of all three workloads. Figure 9 shows the normalized throughputs; each value is normalized to the throughputs of REM-Lazy cases. In convert workload, REM-AA and REM-Simple have no throughput gain for migrating active pages in the rental memory to the normal memory. In REM-AA case, the pages in the rental memory are for an image file that is read once in the workload. Therefore, active page migration provides no effect. In the case of the REM-Simple, it presented 1% of throughput loss. The activation of this simple rental memory management migrates about 30MB of anonymous pages from the rental memory to the normal memory. We believe that the CPU cache compulsory misses by migrating stack and heap pages are the main source of this 1% of throughput loss.

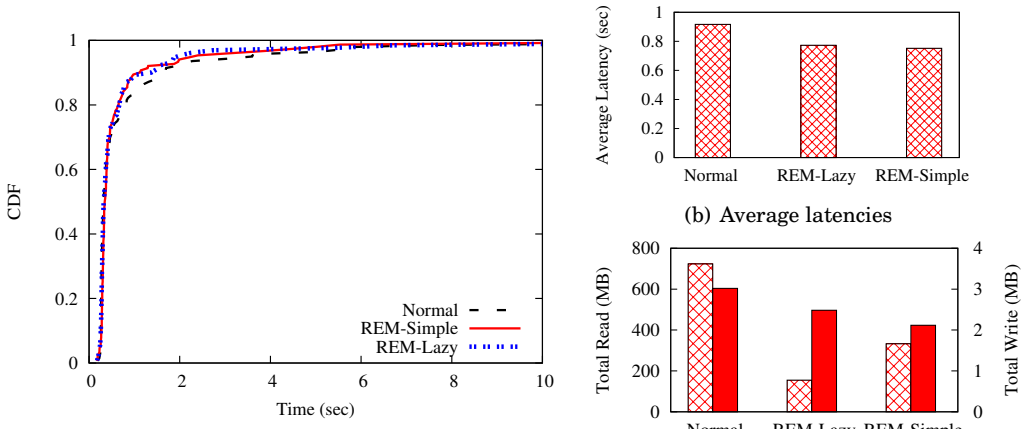
In sysbench100rw workload, we gain performance benefits by the adaptive-activation, since the normal memory has enough room to place the working set of the workload (100MB of file caches). Though REM-Simple also migrates active pages during the return operation, it showed no performance benefit. Our analysis of this phenomenon is that the return operation in REM-Simple issues more page writes than REM-AA and REM-Lazy. Any read or write operation requires waiting due to the lock held by write-back operations. This delay offsets the advantage of the active page migrations of REM-Simple's return operation. In the sysbench100rd workload, REM-Simple and REM-AA both increased their throughputs by 2% compared to the REM-Lazy. The adaptive-activation migrates about 14000 pages during the two sysbench workloads. In sysbench100rd, about 3500 16KB-requests among the 1.6 million requests take advantage of unnecessary I/O reductions. This is the reason that the throughput increase is relatively small compared to the sysbench100rw case. In sysbench100rw, the adaptive-activation makes approximately 2100 requests (1.5 r/w rations reflected) among the 30,000 requests to skip read I/Os.

6.5. Real-world workload

In this section, we measured the performance impact of our scheme with Android applications. This is a challenging experiment for us because we can figure out the effect our page cache-only policy in anonymous page intensive workloads; Android applications are well known for anonymous page intensity due to their java environment. Since there is no representative benchmark application in the Android environment, we ran popular android applications sequentially and measured application launch latencies. The applications we used are shown in shown in Table III. The inputs for each application are randomly generated by *monkeyrunner* in the Android framework. The session time of each application varies from a few second to one hundred seconds.

Table III. A list of Android applications for each category. The categories are from [Falaki et al. 2010].

Category	Applications
Communication	email, sms, call
Browsing	browser, firefox, facebook, twitter, daum (portal site)
Media	gallery, mediaplayer, camera, melon music player
Productivity	alarm, calendar, adobe reader
System	settings, alyac virus scanner
Games	angrybird, quake, archemist
Maps	google map, google map(search), wingspoon
Others	launcher, vending, vending(search)



(a) CDFs of application launch latencies

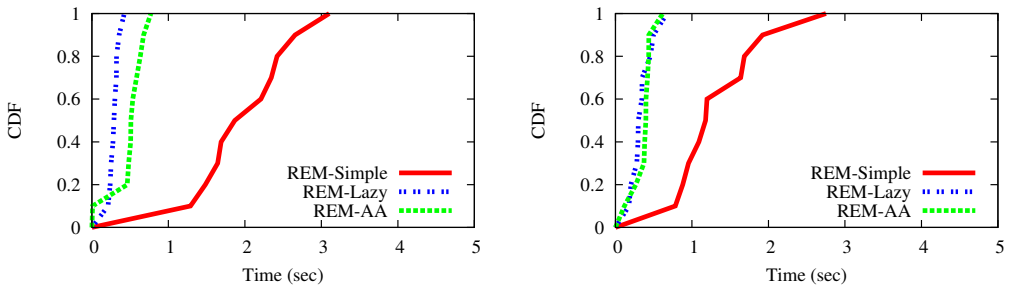
(b) Average latencies

(c) Flash reads and writes

Fig. 10. Effects of REM in launching Android applications

Our performance metric, an application launch latency, is the time between when an application's activity started and when the activity was shown on screen. The latency value is measured by *activity manager* in the Android framework. We sequentially ran the applications. Normal indicates that the evaluation is conducted without the rental memory. REM-Simple and REM-Lazy indicate that the rental memory follows Hotplug Memory's scheme and our scheme respectively. Although we used 24 applications, the number of application launch activities is larger than that because each application has several windows showing activities. For example, a setting application has 6 activities, which are showing wireless settings, sound settings, and so on. We had around 100 activities in each case.

Figure 10(a) shows the CDF of latencies in three cases, and Figure 10(b) depicts the average latencies of the three cases. As shown in the two figures, Normal shows the longest latency since it has the smallest memory among the three cases. In Figure 10(a), about 90% of the latency samples of REM-Lazy and REM-Simple are under 1 second while Normal has about 82% of samples whose latencies are under 1 second. In Figure 10(b), REM-Simple shows the shortest average activity latency since it can use all memory for anonymous pages and for page caches. Therefore, the OS kernel will make its best decisions of whether to keep page cache or to keep anonymous pages. REM-Lazy showed 25 milliseconds of slightly increased latency though. The increase is only 3% of the REM-Simple's average latency. Figure 10(c) shows the total sizes of reads during the experiments. Since REM-Lazy and REM-Simple have more available memory than Normal does, Normal performed more read and write I/Os. In



(a) Video rental memory (b) Camera rental memory

Fig. 11. CDFs of return times of video and camera rental memory regions

the figure, REM-Lazy significantly reduced reads since our scheme can cache a great deal of file data. REM-Lazy performed more write I/Os than REM-Simple since it has a smaller number of dirtyable pages than REM-Simple has. From this experiment, we believe that page caches are also important even though the workload is anonymous page intensive.

Another notable result in this experiment is that real world workload has few write I/Os. The absolute size of reads is around 200 times larger than write size. It indicates that the impact of the clean-only policy is minimal on this real world workload. In this experiment, only 208 pages are performed copy-on-write; the size is 832KB, 33% of total writes in the REM-Lazy case.

Our final evaluation is the measurement of return times as a realistic scenario. We insert a media player application and a camera application in the middle and at the end of the application launch workload, respectively. While the application launch workload fully utilizes the rental memory, the two applications will initiate `return_memory()` on the multimedia decoder's reserved memory and the camera device's reserved memory. In our system, the multimedia player also calls `return_memory()` for the camera device's reserved memory. The size of the two reserved memorys is 72MB and 33MB respectively. Therefore, the multimedia player requires vacating 105MB of the rental memory, and the camera application requires vacating 33MB of the rental memory.

Figure 11 shows the CDFs of return times for the two applications. As you can see in the figure, the latencies are at worst 3.1 seconds and 2.8 seconds in the REM-Simple cases. REM-Lazy shows latencies of at worst 0.4 and 0.7 seconds for the two cases. REM-AA also showed that the time for the adaptive-activation requires at worst an additional 0.77 seconds prior to launching the multimedia player and 0.6 seconds before launching the camera application.

7. DISCUSSION AND CONCLUDING REMARKS

Increasing memory efficiency is a challenging issue in resource-constrained embedded systems. Memory reservation for an integrated device, such as a video decoder and a camera device, makes embedded system less memory-efficient since the reserved memory is wasted during the device's idle time. Emerging systems such as smartphones and smart TVs could worsen the memory inefficiency due to increased ways of the use of the systems and thus an increased idle time of the device.

In this paper, we propose rental memory management that is a software-level abstraction to utilize the device-reserved memory. Conceptually, the OS kernel borrows the reserved memory, which is referred to as *rental memory*, during a device-idle period. In addition, we added the rigorous management of the rental memory to achieve

transparency of the system when the system exploits the rental memory. We applied the page cache-only policy and the clean-only policy to the page placement policy of the rental memory. Those policies are the keys of minimizing the rental memory return time in our scheme for the transparency.

Our prototype demonstrated that the rental memory management increased overall memory utilization even in the device-idle time. The rigorous management policy also has shown comparable performance improvement to the simple management policy except for highly write-intensive workloads. Our main concern, which is minimizing the return time, is also achieved by rigorous rental memory management. In our prototype evaluation, our scheme requires at worst 0.77 seconds to return the rental memory under various synthetic workloads and Android-based workloads. The results also have shown that the lazy-migration reduces unnecessary read I/Os caused by our page placement policy. During the return operation, the adaptive-activation preserves as many page caches as possible to be placed in memory without hurting the return operation time.

Although we suggested the adaptive-activation scheme to preserve working sets in memory, it offsets the minimized return time. As shown in our evaluation results, all return operations with the adaptive-activation have increased their time by around 40%. We argue that whether to use the adaptive-activation is up to an end user or a manufacturer of embedded systems. In addition, our scheme provides the return time of under 1 second for all the tested cases in our prototype system. The time, however, is almost proportional to the size of the reserved memory since each return operation requires some memory instructions per page to release each physical page. Some day, if the reserved memory size increases, the return time could exceed 1 second. Eventually, the increased time may cause a perceivable latency to end-users. We recognize that this limitation is inevitable in the future.

REFERENCES

ABRAMSON, D., JACKSON, J., MUTHRASANALLUR, S., NEIGER, G., REGNIER, G., SANKARAN, R., SCHOINAS, I., UHLIG, R., VEMBU, B., AND AND, J. W. 2006. Intel virtualization technology for directed i/o. *Intel Technology Journal 10(3),* 3.

AMD. 2011. Iommu architectural specification. http://support.amd.com/us/Processor_TechDocs/48882.pdf.

BEN-YEHUDA, M., XENIDIS, J., OSTROWSKI, M., RISTER, K., BRUEMMER, A., AND VAN DOORN, L. 2007. The price of safety: Evaluating iommu performance. In *Proceedings of the 2007 Ottawa Linux Symposium*. 71–86.

CHEN, Z., ZHANG, Y., ZHOU, Y., SCOTT, H., AND SCHIEFER, B. 2005. Empirical evaluation of multi-level buffer cache collaboration for storage systems. In *Proceedings of the 2005 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*. SIGMETRICS '05. ACM, New York, NY, USA, 145–156.

CHIODO, M., GIUSTO, P., JURECSKA, A., HSIEH, H., SANGIOVANNI-VINCENTELLI, A., AND LAVAGNO, L. 1994. Hardware-software codesign of embedded systems. *Micro, IEEE 14,* 4, 26 –36.

CORBET, J. 2010. Contiguous memory allocation for drivers. http://lwn.net/Articles/396702/.

CORBET, J. 2011. A reworked contiguous memory allocator. http://lwn.net/Articles/446836/.

DING, X., WANG, K., AND ZHANG, X. 2011. Srm-buffer: an os buffer management technique to prevent last level cache from thrashing in multicores. In *Proceedings of the sixth conference on Computer systems*. EuroSys '11. ACM, New York, NY, USA, 243–256.

DONG, Y., YU, Z., AND ROSE, G. 2008. Sr-iov networking in xen: architecture, design and implementation. In *Proceedings of the First conference on I/O virtualization*. WIOV'08. USENIX Association, Berkeley, CA, USA, 10–10.

FALAKI, H., MAHAJAN, R., KANDULA, S., LYMBEROPOULOS, D., GOVINDAN, R., AND ESTRIN, D. 2010. Diversity in smartphone usage. In *Proceedings of the 8th international conference on Mobile systems, applications, and services*. MobiSys '10. ACM, New York, NY, USA, 179–194.

GUTHAUS, M., RINGENBERG, J., ERNST, D., AUSTIN, T., MUDGE, T., AND BROWN, R. 2001. Mibench: A free, commercially representative embedded benchmark suite. In *Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop on*. 3 – 14.

HANSEN, D., KRAVETZ, M., AND CHRISTIANSEN, B. 2004. Hotplug memory and the linux vm. In *Ottawa Linux Symposium*.

HU, Y., SIMPSON, A., MCADOO, K., AND CUSH, J. 2004. A high definition h.264/avc hardware video decoder core for multimedia soc's. In Consumer Electronics, 2004 IEEE International Symposium on. *Consumer Electronics, 2004 IEEE International Symposium on*, 385 – 389.

INTEL. 2005. Intel 910gml/915g/915gm/915gms/915gv and 910gl express chipsets intel dynamic video memory technology (dvmt) 3.0. http://download.intel.com/design/chipsets/applnots/30262305.pdf.

JUNG, D., SOO KIM, J., YEONG PARK, S., UK KANG, J., AND LEE, J. 2005. Fass: A flash-aware swap system. In *Proc. of International Workshop on Software Support for Portable Storage (IWSSPS*.

KOPYTOV, A. 2004. Sysbench: A system performance benchmark. http://sysbench.sourceforge.net/.

KOUKOUMIDIS, E., LYMBEROPOULOS, D., STRAUSS, K., LIU, J., AND BURGER, D. 2011. Pocket cloudlets. In *Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems*. ASPLOS '11. ACM, New York, NY, USA, 171–184.

LEE, M., SEO, E., LEE, J., AND SOO KIM, J. 2007. Pabc: Power-aware buffer cache management for low power consumption. *IEEE Transactions on Computers 56*, 488–501.

LOVE, R. 2010. *Linux Kernel Development* third Ed. Addison Wesley.

NAZAREWICZ, M. 2010. Contiguous memory allocator version 6. http://lwn.net/Articles/419639/.

PARK, C., KANG, J.-U., PARK, S.-Y., AND KIM, J.-S. 2004. Energy-aware demand paging on nand flash-based embedded storages. *Low Power Electronics and Design, International Symposium on 0*, 338–343.

SAXENA, M. AND SWIFT, M. M. 2010. Flashvm: virtual memory management on flash. In *Proceedings of the 2010 USENIX conference on USENIX annual technical conference*. USENIXATC'10. USENIX Association, Berkeley, CA, USA, 14–14.

SCHOPP, J. H., HANSEN, D., KRAVETZ, M., TAKAHASHI, H., TOSHIHIRO, I., GOTO, Y., HIROYUKI, K., TOLENTINO, M., AND PICCO, B. 2005. Hotplug memory redux. In *Ottawa Linux Symposium*.

TSENG, P., CHANG, Y., HUANG, Y., FANG, H., HUANG, C., AND CHEN, L. 2005. Advances in hardware architectures for image and video coding - a survey. *Proceedings of the IEEE 93,* 1, 184 –197.

WILLMANN, P., RIXNER, S., AND COX, A. L. 2008. Protection strategies for direct access to virtualized i/o devices. In *USENIX 2008 Annual Technical Conference on Annual Technical Conference*. USENIX Association, Berkeley, CA, USA, 15–28.

YASSOUR, B.-A., BEN-YEHUDA, M., AND WASSERMAN, O. 2010. On the dma mapping problem in direct device assignment. In *Proceedings of the 3rd Annual Haifa Experimental Systems Conference*. SYSTOR '10. ACM, New York, NY, USA, 18:1–18:12.