Rigorous Rental Memory Management for Embedded Systems

JINKYU JEONG, HWANJU KIM, and 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 these systems increase the idle time of their integrated devices, and eventually decrease the utilization of their reserved memory.

In this article, we propose a scheme to minimize the memory inefficiency caused by the memory reservation. The memory space reserved for a device can 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 as 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, memory reservation

ACM Reference Format:
Jeong, J., Kim, H., Hwang, J., Lee, J., and Maeng, S. 2013. Rigorous rental memory management for embedded systems. ACM Trans. Embedd. Comput. Syst. 12, 1, Article 43 (March 2013), 21 pages.
DOI = 10.1145/2435227.2435239 http://doi.acm.org/10.1145/2435227.2435239

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 re-delegate the task of multimedia decoding from CPU to

This work was partly supported by the IT R&D program of MKE/KEIT [10041244, SmartTV 2.0 Software Platform] and by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MEST) (No. 2012-0000148).
Author's addresses: J. Jeong, H. Kim, J. Hwang, and S. Maeng, Computer Science Department, Korea Advanced Institute of Science and Technology; email: jinkyu@calab.kaist.ac.kr; J. Lee, School of Information and Communication Engineering, Sungkyunkwan University.

DOI 10.1145/2435227.2435239 http://doi.acm.org/10.1145/2435227.2435239

an integrated hardware decoder because this delegation reduces the CPU load along with 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 of these 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 these systems. The main reason for this memory reservation is to provide physically contiguous memory space to an integrated device, which making 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. 2010] revealed that the time for video playback is only about 8% of the total smartphone operation time in a day, and the time taken by using a camera is also negligible. 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 smartphones and smart TVs. Most previous embedded systems, such as portable media players or MP3 players, have not experienced this problem because 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 the 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 a graphic device, making itself hard to be applied to 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 software-level. This approach, however, suffers from a long latency when the rented memory is returned back because its return operation does not consider minimizing the return time. According to our measurement, the latency is up to tens of seconds, which is not acceptable for most applications.

In this article, 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 latency to the applications that exploit the integrated devices. Additionally, 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 on 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 a task structure or a memory descriptor, is impossible since such structures are unmovable in general OSes [Schopp et al. 2005]. If such structures are in the rental memory, the device either fails to get back its reserved memory or will cause system failure by corrupting the structures. Anonymous pages, such as process stacks and heap pages, can also be an obstacle in 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 introduce two additional techniques, *lazy-migration* and *adaptive-activation*. The lazy-migration technique alleviates the decreased 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 the Android(Froyo)-based embedded system hardware named Odroid7 with the Android Linux kernel 2.6.32.9. The base platform of the target hardware is the same as that of the Samsung GalaxyS smartphone. We evaluated our scheme with various workloads. The evaluation results have shown that our scheme minimizes the return time to less than 1 second under various workloads. We believe that the return time can be hidden in the initialization time of integrated device's application and is acceptable to end-users. In addition to the minimized return time, our scheme naturally enhances the performance of the system during a device's idle period as compared to the system without our scheme.

The rest of this article is organized as follows. The ensuing section describes the background of our work and also includes 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 presents 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 article with some additional 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* throughout this article, 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 boot 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 region are usually determined at a system design phase and are statically fixed. The Android platform also supports such contiguous memory reservation in the form of *pmem*.

Table I shows 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 a video decoder, a camera device and a JPEG decoder. The sum of available memory and reserved memory is different from the total memory in each system since

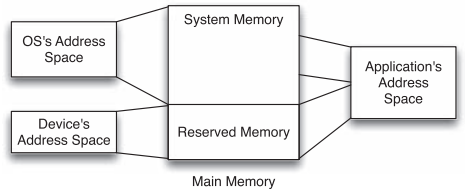


Fig. 1. The memory structure of target hardware.

Table I.
Summary of reserved memory in Android devices. 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.

we have excluded some amount of memory that is invisible for an unknown reason. Another amount of memory which is permanently used by a 3G modem device is also not taken into account. The portion of the reserved memory of each smartphone reaches from 16% to 21% of the 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 a device's reserved memory, we need to define device-idle time. To this end, 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 an `mmap()` system call to open a communication channel between the integrated device and the application. In this case, the kernel provides physical pages from 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, a multimedia player loads an encoded multimedia stream onto the mmaped pages, and an MPEG decoder device 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 the device is ending its operation and is going to be idle.

The period between (1) and (4) is an active state of the device and the reserved memory is used for the function of the device itself. We define this period as *device-active* time. Besides the device-active time, the reserved memory is not being accessed; this period is called *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 for rental memory in 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 the device

during its idle time, we excluded such pages from rental candidates in order to avoid any data corruption.

During device-active time, 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 the CPU during a device-idle time. In this case, we enable CPU caching to regain benefits from CPU caching.

2.2. Related Work

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) are requiring contiguous memory region(s) to operate. Camera devices and video decoders, for example, require contiguous memory regions of megabytes for each device, 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 the physical addresses and the 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 the 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 to be integrated with one of those 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 its 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 the 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 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 onto 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 anonymous pages (i.e., process stack and heap pages), 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 based on the importance of each page [Schopp et al. 2005]. Accordingly, 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 it takes to make room for the device. Accordingly, this approach is also limited for the same reasons as Hotplug Memory from the perspective of transparency.

3. MOTIVATION

In this section, we first explain the basic features of rental memory management. Fundamentally, rental memory management is an abstraction of OS-level support in order to minimize the inefficiency of the 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.

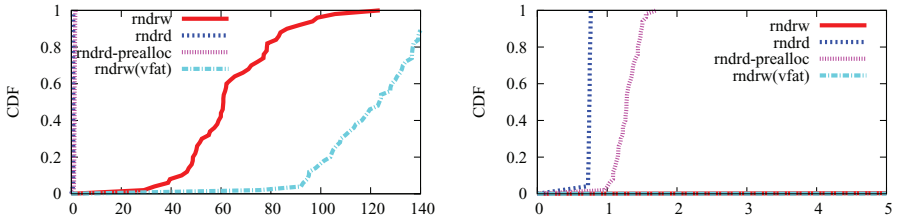
3.1. Rental Memory Management

From the perspective of the OS kernel, the kernel 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 designate this borrowed memory as *rental memory*. In order to achieve the semantics, we require interactions between the OS kernel and a device driver. For the interaction, the kernel provides two interfaces: `rent_memory()` and `return_memory()`. We denote these interfaces a rent and a return operation respectively. 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 code 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 the 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 boot 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 simple rental memory management inherits the Hotplug Memory's policy, it allows movable pages (i.e., anonymous pages and page caches) to be placed in the rental memory. The movable pages, however, could be an obstacle to prompt rental memory return. 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.



(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.

Note that the return operation in the simple management follows Hotplug Memory's unplugging procedure described in Section 2.2.2.

Our evaluation system is equipped with a 1GHz ARM cortex A8 processor with 512MB of main memory. After the system is booted and the rent operation is performed, it has about 150MB of free memory and 105MB of rental memory. To figure out the effect of page caches, we ran *Sysbench* file I/O on 100MB of files whose page caches were placed in the rental memory. We disabled a periodic `fsync()` operation to make the workload more memory-intensive than disk-intensive. During the workload running, we invoked `return_memory()` at random time, and obtained 50 return time samples for each experiment. Figure 2(a) shows the cumulative distribution functions (CDFs) of return times for various workloads. In the figure, workloads having write operations (*rndrw*, and *rndrw(vfat)*) require significantly longer time to complete the return operation as compared to the read-only workload (*rndrd*) because of the write-back operations of dirty pages generated by the workloads. These significant delays would be unacceptable to end users when they want to use the integrated devices. Under a memory insufficient condition (*rndrw-prealloc*), the return time is also increased. The detailed workload is explained in Section 4.3. The main reason for this significantly larger latency is due to the fact that the policy of Hotplug Memory is not designed to minimize the return time.

In addition to page caches, anonymous pages could also be another obstacle for the prompt rental memory return in embedded systems. In practice, most commodity embedded systems are not equipped with swap storage. 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 number of anonymous pages could be larger than the size of the normal memory if a workload is anonymous page-intensive. Once the return operation is performed, we cannot deal with the overcommitment of anonymous pages without swap storage.

4. RIGOROUS RENTAL MEMORY MANAGEMENT

In this section, we explain our rigorous rental memory management that focuses on minimizing the return time of the rental memory. We first describe our policy to control the rental memory. In conjunction with the control policy, we provide additional techniques to enhance our scheme in terms of memory efficiency.

4.1. Page Allocation Policy

In order 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, only page cache (file cache) pages are allowed to reside in the rental memory. This is referred to as a *page cache-only* policy. As described in Section 3.2, allowing anonymous pages in the rental memory can lead to

killing important tasks due to memory insufficiency after the return operation. In our experiments in Section 6.1, we observed that many background daemons and services were killed, and the system no longer became available due to the out-of-memory after the return operation. To this end, 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. Our evaluation result in Section 6.6 showed that the page cache-only policy exhibits similar performance improvement in comparison with a scheme allowing anonymous pages together in Android-based workloads. We also believe that emerging embedded software such as pocket cloudlets [Koukoumidis et al. 2011] will benefit from the increased memory specialized for page caches.

Since the rental memory only admits page caches, the normal memory could have higher utilization and a lower free page ratio than that of the rental memory when a workload intensively uses anonymous pages. For example in our evaluation in Section 6.6, launching many Android applications requires many anonymous pages. The anonymous page allocation requests evict many page caches (e.g., shared libraries, data and executables) from the normal memory because these requests can only be handled within the normal memory. When the rental memory has many free pages, especially right after the rent operation has occurred, it could be beneficial for the evicted pages to be given a second chance, thus being placed in memory for a longer time. When the evicted pages are within a part of a working set, this second chance could reduce costly re-read I/Os for the evicted pages.

We refer to giving the second chance as *lazy-migration*. When a page cache in the normal memory is a target of eviction under the condition described above, we migrate the page into the rental memory instead of evicting the page. For optimization, some pages that are read once and never read again can be filtered by borrowing ideas from previous work [Chen et al. 2005]. We, however did not implement such optimizations in this work. The detailed page migration procedure can be found in the literature [Schopp et al. 2005].

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 pages is significant in terms of the return time. Our preliminary experiment in Figure 2(a) reveals the problem of the dirty page. In the figure, *rndrw* shows tens of seconds of return time while *rndrd* (random read) shows 0.75 seconds in average with little variation. 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 for excluding dirty pages from rental memory.

First, the page lock held by its write-back operation is one of the obstacles to 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 or dirty pages are older than a specific threshold [Love 2010]. If dirty pages in the rental memory are in the middle of write-back operations, the page is neither reclaimable nor movable until the end of write-back operations.

The second 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 (i.e., free or clean 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 that many dirty pages exist in the normal memory, too. This situation inevitably requires the write-back of dirty pages. In addition to this, some file system implementation does not provide dirty page migration. For example in Linux, the dirty pages of VFAT should be written-back first, and then the pages can be migrated. This is the reason for the longer return time of *rndrw(vfat)* in Figure 2(a).

To cope with the clean-only policy, we need to modify page cache allocation paths. In most OSes, there are two methods for page cache allocation. One is through explicit file access operations such as `sys_read()` and `sys_write()`. The other method is through implicit file access using `sys_mmap()`. In explicit allocations, if the `sys_write()` requires a new page allocation, a page from the normal memory pool is provided because this path explicitly indicates that the page will be dirtied. In the `sys_read()` path, a page from the rental memory pool is provided. 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 the rental memory. We allocate a new page from the normal memory, copy data onto that 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 occurrs on the write-disabled page, it will cause a page fault exception and we apply copy-on-write before the write proceeds.

With the clean-only policy, the lazy-migration technique also works with the page evictions that are caused by dirty page allocations in the normal memory. Note that the page replacement algorithm generally chooses clean pages for eviction, and therefore the lazy-migration does not violate our clean-only policy.

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 each clean page cache.

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.

In more detail, Hotplug Memory aggressively 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. In Figure 2(b), *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. Once 13MB of free pages are filled up, the 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 the physical pages in the normal memory and reclaims some of those pages. In the end, already migrated page caches are also evicted to make room for the ensuing migrations. This page allocation contention is the main source of the increased return time.

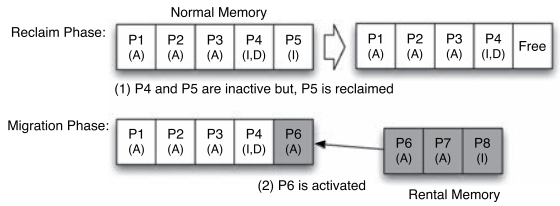


Fig. 3. Overview of the adaptive-activation. A, I, and D indicate active, inactive and dirty, respectively.

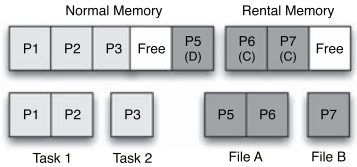


Fig. 4. An overview of rigorous rental memory usage. D and C denote dirty and clean respectively.

In our scheme, we require a more sophisticated approach to achieve both memory efficiency and prompt rental memory return. The main reason for unpredictable delay is that the unplugging procedure in Hotplug Memory does not take into consideration the state of 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 determines 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, each physical page in the normal memory is scanned once. If each physical page is inactive and reclaimable immediately, the page is reclaimed. Otherwise, the reclaim phase continues on the next physical page. If the number of reclaimed pages exceeds the number of active pages in the rental memory, the reclaim phase ends.

In the migration phase, two conditions can exist. 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 a long delay. The other condition is that an insufficient number of physical pages are reclaimed during the reclaim phase. This case indicates that additional page reclaim requires a long delay such as in write-back I/Os. Accordingly, some of the active pages in the rental memory are discarded. Partial discarding is still better than discarding all pages in terms of memory efficiency. In Figure 3, we have two inactive pages but our reclaim phase collects page P5 only. Because P4 is inactive and dirty, P4 is not reclaimed. Therefore, one active page P6 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, more active pages in the rental memory are discarded.

Since our adaptive-activation avoids memory contention within the normal memory, the time to complete the return operation is almost deterministic. 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.

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

memory pages including page caches and anonymous pages, whereas the rental memory only permits clean page caches. Although our two policies limit the usage of rental memory, we observed in Section 6.6 that the memory utilization is not seriously compromised in our experiments. The lazy-migration technique also alleviates possible memory utilization degradation by the limited usage. In addition, the upper-bound of anonymous page size is the same as that without any rental memory scheme used.

By placing only clean caches in the rental memory, the return operation becomes much simpler and less time-consuming. The adaptive-activation technique also makes the return time more predictable and also minimizes additional read I/Os due to discarding clean caches from the rental memory.

As the system gains more memory from the rental memory scheme, we can expect performance improvement by caching more data in memory than storing it on disk or in flash. In addition, the kernel's write-back heuristics reduces the frequency of task blocking for write-back operations. The detailed reason is explained in Section 6.2.

5. IMPLEMENTATION

We implemented our scheme on the Android Linux kernel 2.6.32.9. We modified memory hotplugging codes to support variable lengths and used page migration features in Linux and Hotplug Memory [Schopp et al. 2005]. For copy-on-write, we made a new interface that migrates a single page.

In Odroid7, we modified three device drivers, a MPEG decoder, a camera device, and a JPEG decoder, to support rental memory management. The sizes of reserved memory for each device are 72MB, 33MB, and 8MB; the reserved memory of the three devices is 74% of the system's total reserved memory.

The rental memory pool is managed as `ZONE_REM`, which is initially empty at boot time. A buddy system allocator, Linux's default page allocator, provides page allocation from the `ZONE_REM`. Linux also has a per-zone-based page replacement policy and manages a per-zone approximated LRU list. Our adaptive-activation functions 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 function to distinguish page cache allocation from other types of page allocation. A page cache allocation is serviced from `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 invoked in the middle of `sys_write()`-related functions. The allocation request from this function is serviced from the `ZONE_NORMAL`. For the copy-on-write scheme, when the write system call tries to update a page in `ZONE_REM`, we migrate the page into the `ZONE_NORMAL` and proceed to update the page.

6. EVALUATION

The evaluation of our scheme 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 a Securer Digital (SD) card partition. The evaluations in Sections 6.5 and 6.1 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* [Guthaus et al. 2001]. *Convert*, a prevalent image editing workload, performs rotation and normalization of a 30MB-sized bitmap image file and writes 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 three file I/O modes: random read, read/write and sequential write. The notation *sysbench300rw* indicates a Sysbench random read/write on 300MB of file set. We disabled a periodic `fsync()` operation and warmed up the input files to make the workload more memory-intensive than disk-intensive. The random read/write workload executes 30000 I/O 16KB-requests with a 1.5 read/write 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.

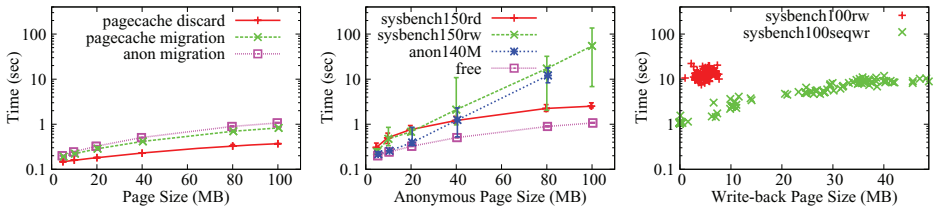
We used the reserved memory of two devices; the devices are the video decoder device and the camera device. The total size of the two reserved memory regions is around 105MB. We omitted the use of the JPEG decoder's reserved memory because the reserved memory of the two devices is sufficient to show the effect of rental memory. Except for the return time evaluation in Section 6.6, we regarded two rental memory regions as one rental memory region. The free memory after booted is about 150MB since about 130MB of memory has been occupied by kernel data, system daemons, and background services during the boot. After the rent operation, the free memory increased to 255MB even though the use of the rental memory is limited by each scheme. The evaluation is focused on measuring the following metrics.

—Trade-Off analysis. In order to minimize the rental memory return time, we adopted the page cache-only policy and the clean-only policy. We analyzed how dirty and anonymous pages adversely affect the return time.
—Performance enhancement by REM. The main reason for rental memory management is to enhance the performance of a system during the 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 the rental memory, copy-on-write overhead is inevitable. We characterized the overhead caused by copy-on-write in our evaluations.
—Effects of lazy-migration and adaptive-activation. Both 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. For the transparency of a system, the main goal of rigorous rental memory management is to minimize the return time. We measured the return times under various workloads.

6.1. Trade-Offs Analysis

Before the trade-off analysis, it is important to know the costs of page migration and of page discard from the perspective of time; anonymous pages though have no choice but to be migrated into the normal memory without swap storage. We can also infer the time costs of adaptive-activation since the technique includes page migrations. Figure 5(a) shows the average time taken either to migrate or to discard page caches and anonymous pages while varying their size. In this evaluation, the pages resided in the rental memory. In *pagecache migration* and *anon migration* cases, the normal memory had enough space to handle their page migrations. As shown in the figure, the time cost increases in all cases when the number of pages increases. Discarding page caches (*pagecache discard*) has the lowest cost in time. Page cache migrations (*pagecache migration*) require more time in which to copy page data and to re-establish page cache structures. Anonymous page migrations (*anon migration*) require additional time since re-establishing page tables is also required.

Next, we measured how anonymous pages effect the return operation. In a real system, it is not practical to guarantee that the normal memory have a sufficiently large amount of free space available. In light of this circumstance, we placed various types



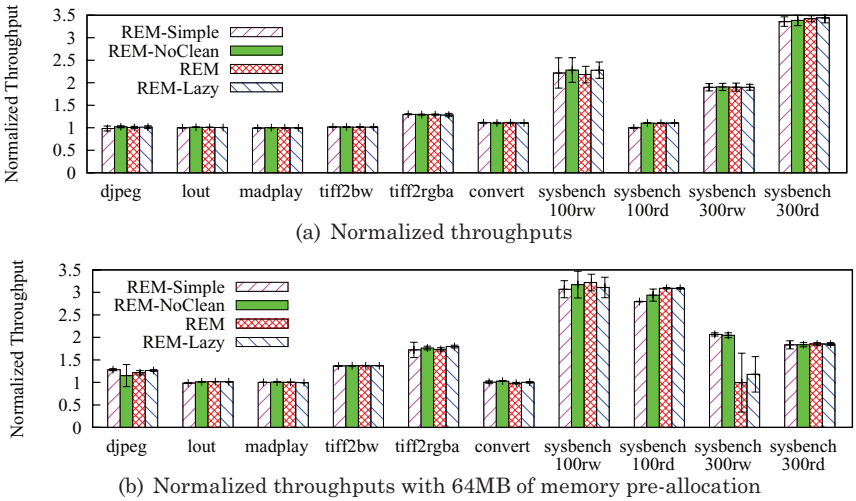
(a) Page migration and discard cost (b) Anonymous page (c) Dirty(write-back) page

Fig. 5. Trade-off analysis in terms of return time under various conditions.

of memory pages (i.e., clean caches, dirty caches, and anonymous pages) in the normal memory. Figure 5(b) shows the average, minimum and maximum of return times while varying the number of anonymous pages in the rental memory. The return operation used in this evaluation was to migrate the anonymous pages from the rental memory to the normal memory. We had three normal memory's conditions: (1) containing clean caches (*sysbench150rd*), (2) containing dirty caches (*sysbench150rw*), and (3) containing anonymous pages (*anon140M*). Note that *free* is the same value as *anon migration* in Figure 5(a). For anonymous pages, we used a simple matrix multiplication program, and adjusted the anonymous page size by varying its matrix size. In the *anon140M* case, two matrix programs were run, one on the normal memory and the other on the rental memory.

In the *sysbench150rd* case, the clean page caches in the normal memory can be evicted to make free space for the anonymous page migrations. When the size of migrating anonymous pages increases, the memory contention causes the return time to increase as shown in Figure 5(b). In the *sysbench150rw* case, making room in the normal memory takes longer time as a result of the normal memory being filled with many dirty pages. In the *anon140M* case, when the anonymous page size is smaller than or equal to 20MB, the return times are similar to those of the *free* case. When the anonymous page size is 40MB, the memory contention within the normal memory lengthens the return time to 1.2 seconds whereas the return time without the contention is 0.5 seconds. At this point, the normal memory can handle the memory requests for the migration by evicting page caches occupied by system daemons and background services. When the anonymous page size increases to 80MB, the normal memory cannot handle the memory allocations for the migrations. Eventually, the system fell into an out-of-memory state. Many system daemons and background services were killed and even the system on occasion no longer became available. From this evaluation, keeping anonymous pages in the rental memory not only increases the cost of the return operation but also causes system failure during the return operation.

As we stated in Section 4.2, write-back pages are the obstacle under the page cache-only policy. The number of write-back pages is determined by the kernel's write-back semantics, and the semantics are well-defined by the kernel developers. Accordingly, we measured the relationship between the return time and the size of write-back pages while workloads running. We ran Sysbench random read/write and sequential write on the rental memory 100 times. For each experiment, we conducted the return operation at random time. In this evaluation, no memory contention occurred during the return operation because the normal memory has a sufficient amount of space for the page migrations. Figure 5(c) plots the return time and the number of write-back pages when the return operation occurred. We note that the return time is mostly wait-time for write-back I/Os in this evaluation. As shown in the figure, waiting for the completion of the write-back I/Os requires up to tens of seconds even if the size of requested write-back pages is small around 5MB in the random read/write workload. In the



(a) Normalized throughputs

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

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

sequential write workload, the wait-time per dirty page is decreased, and the total return time is more linear than that of the random read/write workload. However, the return time is still several seconds. Note that the flash device in our evaluation system (and almost in general) performs much faster for sequential writes than for random writes.

From these evaluations, we argue that the page cache-only policy and clean-only policy are essential for minimizing the return time.

6.2. Performance

In this section, we measured how the additional memory gains throughput improvement under 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 allows for dirty page caches to be placed in rental memory, too. REM indicates that the rental memory is used under the rigorous management scheme without the lazy-migration. REM-Lazy includes the lazy-migration. Although, REM-Lazy is our scheme, four legends can help us figure out the effect of each policy in more detail. We obtained 20 samples and depicted average and standard deviation of the normalized throughput for each experiment.

As depicted in the figure, *tiff2rgba*, *convert*, and Sysbench workloads with the rental memory show throughput increases by 30%–250%. In the cases of *sysbench300rd* and *sysbench300rw*, the performance benefits mainly come from an increase in available memory; the workloads cache more data in memory. The other workloads, *tiff2rgba*, *convert*, and *sysbench100rw*, gain throughput increases as a result of the Linux-specific characteristic although their working-set size can be fit in the normal memory.

The Linux-specific characteristic comes from a dirty page write-back heuristic in Linux itself. The main purpose of the write-back heuristic is to preserve promptly reclaimable pages (i.e., clean pages) over a given threshold [Love 2010]. When a system's dirty ratio, the number of dirty pages divided by the number of dirtyable (free or clean) pages, is below a specified threshold, 0.2 in our evaluation, the *pdflush* kernel thread performs write I/Os on behalf of the applications. When the ratio is

Table II. Summary of copy on write overhead: Increased CPU utilization, jiffies, and the number of copy on write. 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

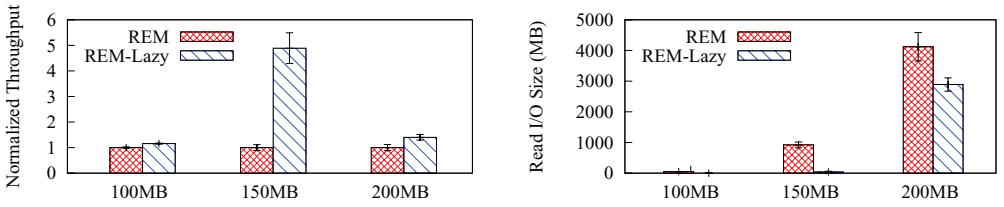
above the threshold, the kernel forces the applications to directly perform write-back I/Os. These write I/Os cause the applications to be blocked when the request queue of the block device is full. Once available memory is increased by the rental memory, the applications are less frequently blocked because the dirty ratio is less frequently exceeded. For this reason, the workloads, *tiff2rgba*, *convert*, and *sysbench100rw*, have shown throughput improvement as shown in Figure 6(a).

In embedded systems, it is difficult to have a large amount of free memory at runtime. Accordingly, many workloads could face a more memory-constrained situation. In light of this situation, we synthetically gave more memory pressure to the system. We ran a task that allocated around 64MB of anonymous pages in the background and then conducted the identical workloads. Figure 6(b) shows the normalized throughput of the same workloads with the memory pre-allocation. Two workloads *djpeg* and *tiff2bw* improve their throughputs under high memory pressure. Within these conditions, *tiff2rgba* and other Sysbench workloads show more improved throughput than 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 number of file writes is 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.

6.3. 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 the CPU utilization, the CPU cycle, and the copy-on-write size for 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 range from 5% to 12%. In the case of Sysbench 300MB with pre-allocation, the overall CPU utilization is 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 the 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 exhibited 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 through `sys_write()` system calls.



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

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

6.4. Effects of Lazy-Migration

In order to show the effect of lazy-migration in more detail, we designed the following synthetic evaluation. We warmed up the normal memory with input files and emptied the rental memory. After we started Sysbench benchmark, we ran a memory hogging task which gradually increased its anonymous page allocation of up to 128MB. Therefore, the normal memory was busy handling anonymous page requests while the rental memory had many free pages. Figure 7 shows the 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 the 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 the REM case with a 100MB file set size. In the case of the 150MB file set size, the throughput increases by 400% as compared to the REM case. While the amount of allocated anonymous page is increasing, memory contention for file caches occurred because the size of page cache was decreasing relatively. Therefore, the contention incurs frequent file data caching and eviction. Figure 7(b) shows that four times larger read I/Os occurs in the REM case than in the REM-Lazy case. The lazy-migration technique gives second chances to the page caches and thus reduces unnecessary read I/Os. In the case of the 200MB file set size, the lazy-migration increases throughput by 40%. Since both the 200MB file caches and the 128MB anonymous pages cannot be placed in the two memory regions, file caches are loaded and evicted frequently in both cases. Accordingly, the absolute read size of the 200MB case is significantly larger than that of the 150MB case as shown in Figure 7(b).

6.5. Return Memory Operation

In this subsection, we show how our scheme successfully minimizes the return time of the rental memory as compared to the simple rental memory management. Additionally, we measured how adaptive-activation brings performance improvement by migrating active page caches instead of discarding them.

Figure 8 shows the CDFs of return times of three workloads: *convert*, *sysbench100rw*, and *sysbench100rd*. REM-AA denotes REM-Lazy with adaptive-activation. The return operation of REM-Simple is the unplugging procedure of Hotplug Memory in Section 2.2.2. We varied the time at which the return operation occurred, and we measured 50 return times for each workload. In all the cases, REM-Lazy shows the shortest return time, around 0.4 seconds, in every workload because its return operation has the lowest cost, discarding all clean caches from the rental memory. REM-AA required additional time (0.2–0.3 seconds) at the cost of keeping active pages in memory. REM-Simple shows a similar return time to that of REM-AA in *convert* workload

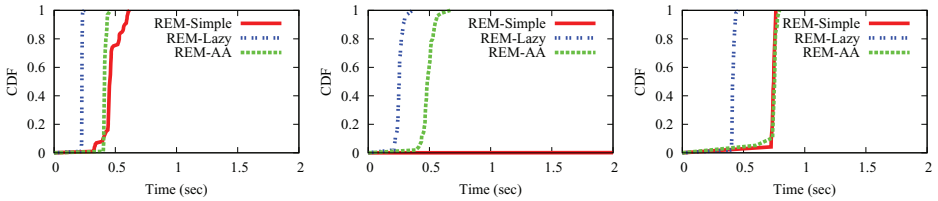


Fig. 8. CDFs of return times of three workloads.

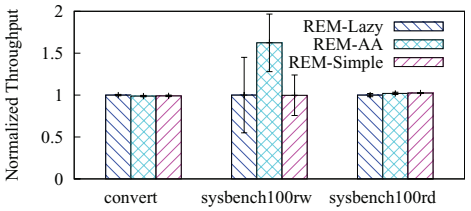


Fig. 9. Throughputs of three workloads with return operations.

and *sysbench100rd* workload because neither memory contention nor dirty pages occurs in the workload. In *sysbench100rw* workload, however, REM-Simple presents an order of magnitude larger return time in Figure 8(b) because many dirty pages (actually write-back pages) in the rental memory disturb the return operation.

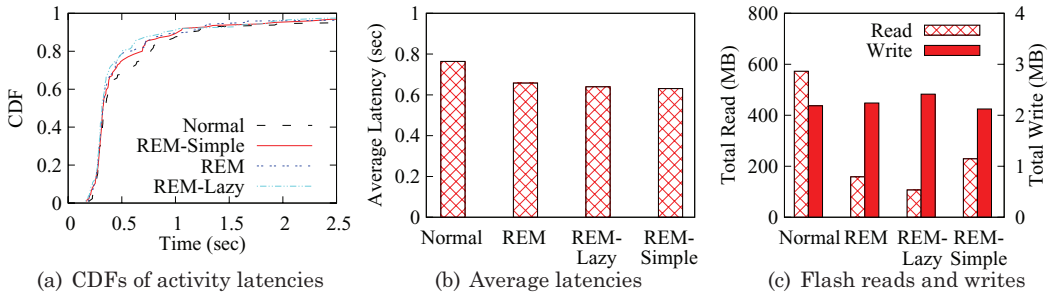
From those experiments, REM-Lazy provides the shortest return time since it does not perform any page migrations. The adaptive-activation technique requires additional costs of the page copies for page migrations, but still shows at most 0.7 seconds for the return operation.

We also measured the throughputs of the above three workloads in order to figure out the performance improvement by adaptive-activation, especially by migrating active page caches. 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 when migrating active pages in the rental memory to the normal memory. The reason for this is that the pages in the rental memory are for an image file that is read once during the workload. Therefore, active page migration has no effect. REM-Simple, however, incurs 1% of the throughput loss even though it also migrates active pages. In more detail, the activation of REM-Simple case 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 reason for the throughput loss.

The *sysbench100rw* workload gains performance benefits by 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 shows 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 wait-time 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 increase their throughputs by 2% as compared to REM-Lazy. Adaptive-activation migrates about 14,000 pages during the two sysbench workloads. In *sysbench100rd*, about 3,500 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

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

Category	Applications
Communication	email, sms, call
Browsing	browser, 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, alchemy
Maps	google map, google map(search), wingspoon
Others	launcher, vending, vending(search)



(a) CDFs of activity latencies (b) Average latencies (c) Flash reads and writes

Fig. 10. Effects of REM when running Android applications.

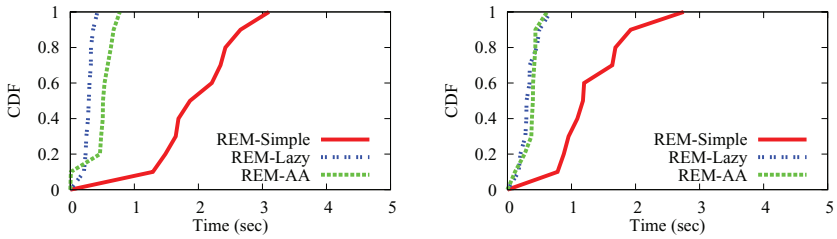
compared to the *sysbench100rw* case. In *sysbench100rw*, the adaptive-activation makes approximately 2100 requests (1.5 r/w ratios reflected) among the 30,000 requests to skip read I/Os.

6.6. Real-World Workload

In this section, we measured the performance impact of our scheme with Android real-world applications. Since there is no representative benchmark application in the Android environment, we ran popular Android applications sequentially and measured application (activity) launch latencies. The applications we used are shown in Table III. The inputs for each application are randomly generated by the *monkeyrunner* tool in the Android framework. The session time of each application varies from a few seconds to one hundred seconds.

Our performance metric, application launch latency (or activity latency), is the time between when an application's activity starts and when the activity is shown on screen [Morrill 2008]. The latency value is measured by *activity manager* in the Android framework. Normal indicates that the evaluation is conducted without the rental memory. Although we used 24 applications, the number of application launch activities is around 100, which is larger than the number of the applications since each application can have several windows showing activities. For example, a setting application has 6 activities, showing wireless settings, sound settings, and so on.

Figure 10(a) shows the CDF of latencies in four cases, and Figure 10(b) depicts the average latencies of the four cases. As depicted in the two figures, Normal shows the longest latency since it has the smallest available memory. REM-Simple shows the shortest average latency (82.5%, normalized value to the Normal case), REM-Lazy the second (84%), and REM the third (86%). Because the OS kernel can make its best decisions on whether to keep page caches or to keep anonymous pages, REM-Simple shows the shortest average latency. The two schemes, REM and REM-Lazy, however, use the rental memory for page caches, and reduce read I/Os as shown in Figure 10(c).



(a) MPEG decoder's rental memory (b) Camera device's rental memory

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

The lazy-migration technique migrates on average 76MB of page caches, and 63% of migrated page caches are accessed again. Accordingly, REM-Lazy reduced the read size more than that of REM as shown in Figure 10(c).

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

From the perspective of memory utilization, the REM-Simple scheme shows 93% of the average memory utilization and the REM-Lazy scheme shows 89%. The main reason for the memory occupancy degradation comes from the limited use of the rental memory in our scheme. The REM-Lazy scheme, however, shows similar activity latency compared to the REM-Simple. This result indicates that the cost of the limited use can be compensated by caching more data such as shared libraries, executables and file data.

Our final evaluation is the measurement of return times as a realistic scenario. We inserted 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 on 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 memories 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. We argue that these delays are severe in terms of user perceived latency because this time is a completely additional delay in running the media player or the camera application. REM-Lazy shows latencies of at worst 0.4 and 0.7 seconds for the two cases respectively. REM-AA also shows that the time for adaptive-activation requires at worst an additional 0.77 seconds (4 times shorter than REM-Simple) prior to launching the multimedia player and 0.6 seconds (4.6 times shorter than REM-Simple) before launching the camera application. The main reason for the long return time of the REM-Simple case is the memory contention in the normal memory when the return operation occurs. The adaptive-activation technique, however, does not increase the return time under the insufficient memory condition.

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 systems 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 memory inefficiency due to the increased ways of the use of these systems and thus causing an increase in the device's idle time.

In this article, 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 have applied the page cache-only policy and the clean-only policy to the management policy of the rental memory. These policies are the keys to minimizing the rental memory return time in our scheme for the transparency.

Our prototype demonstrates that simple rental memory management is improper to minimize the rental memory return time. On the contrary, our scheme shows minimized return time in synthetic and Android-based workloads. Our scheme also provides comparable performance improvement to the simple management policy except for highly write-intensive workloads; we believe such highly write-intensive workloads are rare in real-world embedded systems. The results also have shown that lazy-migration reduces unnecessary read I/Os caused by our page placement policy and always provides better or the same performance as compared to that without the lazy-migration. During the return operation, the adaptive-activation preserves active page caches being placed in memory instead of discarding them. It, however, offsets the minimized return time by about 40% compared to that without the adaptive-activation. We argue that whether to use the adaptive-activation is up to the end user or the manufacturer of embedded systems based on their demands for transparency quality.

REFERENCES

Abramson, D., Jackson, J., Muthrasanallur, S., Neiger, G., Regnier, G., Sankaran, R., Schoinas, I., Uhlig, R., Vembu, B., and J. W. 2006. Intel virtualization technology for directed I/O. *Intel Technol. J. 10*, 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 OLS '07*. 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 SIGMETRICS '05*. 145–156.

Chiodo, M., Giusto, P., Jurecska, A., Hsieh, H., Sangiovanni-Vincentelli, A., and Lavagno, L. 1994. Hardware-software codesign of embedded systems. *IEEE Micro 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 EuroSys '11*. 243–256.

Dong, Y., Yu, Z., and Rose, G. 2008. SR-IOV networking in Xen: architecture, design and implementation. In *Proceedings of WIOV '08*. USENIX Association, Berkeley, CA, 10–10.

Falaki, H., Mahajan, R., Kandula, S., Lymberopoulos, D., Govindan, R., and Estrin, D. 2010. Diversity in smartphone usage. In *Proceedings of MobiSys '10*. 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 *Proceedings of WWC '01*. 3–14.

Hansen, D., Kravetz, M., and Christiansen, B. 2004. Hotplug memory and the Linux VM. In *Proceedings of OLS '04*.

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 *Proceedings of ISCE '04*. 385–389.

Intel. 2005. Intel dynamic video memory technology (DVMT) 3.0. http://download.intel.com/design/chipsets/applnots/30262305.pdf.

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 ASPLOS '11*. 171–184.

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

LOVE, R. 2010. *Linux Kernel Development* 3rd Ed. Addison Wesley.

MORRILL, D. 2008. Inside the Android application framework. *Google I/O*.

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

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 *Proceedings of OLS '05*.

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. *Proc. 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 *Proceedings of USENIX ATC '08*. 15–28.

YASSOUR, B.-A., BEN-YEHUDA, M., AND WASSERMAN, O. 2010. On the DMA mapping problem in direct device assignment. In *Proceedings of SYSTOR '10*. 18:1–18:12.

Received June 2011; revised January 2012; accepted March 2012