DaaC: Device-reserved Memory as an Eviction-based File Cache

Jinkyu Jeong
Computer Science Dept.
Korea Advanced Institute of Science and Technology
Daejeon, Rep. of Korea
jinkyu@calab.kaist.ac.kr

Hwanju Kim
Computer Science Dept.
Korea Advanced Institute of Science and Technology
Daejeon, Rep. of Korea
hjukim@calab.kaist.ac.kr

Jeaho Hwang
Computer Science Dept.
Korea Advanced Institute of Science and Technology
Daejeon, Rep. of Korea
jhhwang@calab.kaist.ac.kr

Joonwon Lee
College of ICE
Sungkyunkwan Univ.
Suwon, Rep. of Korea
joonwon@skku.edu

Seungryoul Maeng
Computer Science Dept.
Korea Advanced Institute of Science and Technology
Daejeon, Rep. of Korea
maeng@calab.kaist.ac.kr

ABSTRACT

Most embedded systems require contiguous memory space to be reserved for each device, which may lead to memory under-utilization. Although several approaches have been proposed to address this issue, they have limitations of either inefficient memory usage or long latency for switching the reserved memory space between a device and general-purpose uses.

Our scheme utilizes the reserved memory as an eviction-based file cache. It guarantees contiguous memory allocation to devices while providing idle device memory as an additional file cache called *eCache* for general-purpose usage. Since the eCache stores only evicted data from in-kernel page cache, memory efficiency is preserved and allocation time for devices is minimized. Cost-based region selection also alleviates additional read I/Os by carefully discarding cached data from the eCache. The prototype is implemented on the Nexus S smartphone and is evaluated with popular Android applications. The evaluation results show that 50%-85% of flash read I/Os are reduced and application launch performance is improved by 8%-16% while the reallocation time is limited to a few milliseconds.

Categories and Subject Descriptors

D.4.2 [**Operating Systems**]: Storage Management—*main memory, allocation/deallocation strategies*

General Terms

Management, Performance

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
CASES'12, October 7-12, 2012, Tampere, Finland.
Copyright 2012 ACM 978-1-4503-1424-4/12/09 ...$15.00.

Keywords

memory management, memory reservation, contiguous memory allocation

1. INTRODUCTION

Recently, general-purpose embedded systems such as smartphones are increasing their market shares. Gartner expects that mobile phones will overtake PCs as the most common Web access device worldwide [11]. International Data Corporation also predicts that vendors will ship one billion smartphones in 2015 [16]. Since such embedded systems have lower computing power than general PCs, many hardware-implemented features complement CPU's insufficient computing capability. For example, video playback and camera functions are usually implemented by separate hardware. This computing delegation from CPU to those acceleration hardware devices not only unburdens CPU load but also decreases power consumption.

Such an acceleration hardware device usually needs contiguous memory space. For example, decoding one full-HD frame requires at least 6MB of memory, and the required memory size can increase depending on the implementation of decoding function. In addition, the required memory should be physically contiguous since those devices do not usually support scatter/gather I/Os [8]. Since an operating system (OS) manages the whole physical memory at page-granularity, it is not easy to allocate physically contiguous large memory chunks to those devices in runtime. Therefore, the necessary memory space for a device is usually reserved statically as *static reservation.* Many state-of-the-art smartphones reserve from 15% to 22% of its main memory for such acceleration hardware devices as shown in Table 1.

While the memory reservation guarantees exclusive use of a device, it leads to inefficient memory utilization since the memory region cannot be used for other purposes even when the device is idle. Many reports reveal that video playback or taking pictures are not dominant workloads in smartphone applications [10, 25]. Accordingly, the memory space for the dedicated use of the acceleration devices is wasted.

IOMMU (input/output memory management unit) [22, 1, 3], which provides on-demand mapping between I/O addresses and physical addresses, can eliminate the low memory utilization problem by providing *on-demand memory allocation* for devices. Since IOMMU provides virtually contiguous address spaces to devices, scattered pages allocated in on-demand manner becomes contiguous on a device's I/O address space. Additional cost of IOMMU hardware, however, can increase the unit cost of system. In addition, managing address mappings for devices increases burden on CPU [5, 4]. Finally, since this approach is hardware dependent [22], its deployment is limited.

Contiguous Memory Allocation [27] and Rental Memory [18] provide software-level approaches to increase the memory utilization. The two approaches allow kernel to use reserved memory for movable pages such as stack, heap and page cache pages while acceleration devices are idle. When a device becomes active, the device's reserved region can be exclusively used by the device because the movable pages in the region can migrate into other memory regions (e.g., normal memory). We denote this operation as *on-demand reservation* since this operation still provides physically contiguous regions to devices. As these schemes are implemented at software-level, many deployed devices can take advantage of those approaches via software update.

The two approaches, however, sacrifice either end-user latency or memory efficiency. When the movable pages are placed on the reserved memory, the dirty pages among them take long time to be placed in other location when the device becomes active. This additional delay directly increases the launch time of applications which depend on the acceleration devices. Accordingly, the degree of user satisfaction can be seriously compromised. If only clean data is placed on the reserved memory, returning the reserved memory to the device becomes simple and fast (hundreds of milliseconds) by discarding the clean data [18]. The simple discard operation can be, however, inefficient because it may cause additional page faults.

In this paper, we propose a novel device-reserved memory management scheme that provides on-demand memory reservation to devices while preserving the memory efficiency of system and minimizing the on-demand reservation time for devices. Our scheme uses Device-reserved memory as an eviction-based file Cache (DaaC). By gathering the reserved memory regions for each device, we build an eviction-based file cache (*eCache*). The eCache provides contiguous memory allocation (i.e., on-demand reservation) to acceleration devices while it stores file caches in memory to increase the memory utilization. Since the management of the eCache is eviction-based and exclusive, data stored in the eCache is guaranteed to be less important than those in upper-level cache (e.g., in-kernel page cache). Therefore, the on-demand reservation comes at a minimal cost by discarding data stored in the eCache.

In addition, cost-based memory region selection during the on-demand reservation is suggested to sustain the memory efficiency (or the caching efficiency of the eCache). Since the eCache consists of many devices' reserved memory regions, one device can utilize multiple regions for its memory provisioning when not all devices are active simultaneously. Accordingly, it is important to choose a device allocation region so that minimum number of page faults occurs after the on-demand reservation. In this regard, we assign the cost for each cached data of which the cost value reflects the LRU distance in the eCache. By aggregating the cost of the cached data in each predefined allocation region, our scheme can find a minimal cost region for device memory allocation. When all devices are activated, the eCache can still handle all on-demand reservation requests without external fragmentation.

Device	Total	Avail.	Reserved	Usage
Nexus One	512	379	81 (16%/23%)	Video, Camera
Desire HD	768	N/A	157 (20%/N/A)	”
Galaxy S	512	334	115 (22%/34%)	JPEG, Video, Camera
Nexus S	512	386	105 (21%/27%)	”
Galaxy S2	1024	816	154 (15%/19%)	”

Table 1: Memory reservation in many smartphones (in MB)

Based on the proposed scheme, we implemented the prototype on Nexus S smartphone with Android Open Source Project 2.3.7_r1 and Linux kernel 2.6.35.7. The prototype is evaluated with various user workloads that comprise many well-known Android applications. When reserved memory regions, whose size is 21% of the total memory, are managed in our scheme, read I/O size is significantly reduced by 76%-85% when the workload contains only 5% of applications that depend on acceleration devices. The average application launch time is also reduced by from 8% to 16%. Even though a workload contains large portion of acceleration device-dependent applications, 50%-70% of read I/Os are still serviced by the eCache. While the eCache absorbs significant read I/O traffics, the on-demand reservation is still provided in a few of milliseconds, 10 ms for a video decoding device and 4 ms for a camera device. The two reservation time are only 2% of Movies application's launch time and 1% of Camera application's one, respectively. The device memory allocation delay is small enough that our scheme can be transparent.

The rest of this paper is organized as follows. The following section shows the background of this work and depicts the motivation. Section 3 presents the details of our scheme. Section 4 discusses implementation issues of our prototype. Section 5 shows the evaluation results of our scheme. Section 6 discusses the related work. We conclude this paper in Section 7.

2. BACKGROUND AND MOTIVATION

In this section, we first provide the fact that nontrivial portion of reserved memory could be wasted for most of the time by presenting the memory reservation status and by showing previous surveys on statistics of smartphone application usage. Then, we describe the motivation of our work with illustrating the limitation of previous approaches.

2.1 Background

Many state-of-the-art smartphones use memory reservation (static reservation) for their multimedia acceleration devices. Table 1 shows a total memory size, an OS-available memory size (in `/proc/meminfo`), and a reserved memory size in each smartphone; the reservation information is obtained by analyzing each device's source code. As shown in

Device	Region	Size	Usage
Video Decoder	mfc0(fw)	2MB	Firmware
	mfc0	34MB	Video decoding
	mfc1	36MB	Video decoding
Camera	fimc0	6MB	Taking pictures
	fimc1	9MB	Always
	fimc2	6MB	Taking pictures
JPEG decoder	jpeg	8MB	Taking pictures

Table 2: Detailed usages of reserved memory in Nexus S

the table, many smartphones reserve a number of large continuous memory regions for their multimedia acceleration devices such as a camera, a video decoder and a JPEG decoder. We note that some memory regions, which are used by other devices such as a radio device or a GPU device, are not taken into account in the table. In summary, the portion of the reservation is nontrivial, reaching from 15% to 22% compared to their total memory size.

In more detail of Nexus S [13], which is used in our prototype implementation, it has three devices that reserve six disjoint memory regions. As shown in Table 2, the video decoder (*s3c mfc*), reserves two 36MB-sized regions for its operation. The 2MB-sized region at the beginning of the first region stores the firmware code of the device. The camera device (*s3c fimc*) reserves three memory regions, whose sizes are 6MB, 9MB and 6MB, respectively. From our test, the first and the last regions are used while the camera device is working. The second region is always used regardless of whether the camera device is active; always-used regions are not our target. The JPEG decoder device (*s3c jpeg*) reserves 8MB of memory which is used when to change raw image data taken from the camera device to a JPEG-formatted file. Hence, the JPEG device works only while a picture is being taken. In summary, 70MB of memory for the video decoder is used only while the device is working (e.g., by a video playback application). The 12MB of the memory for the camera device is used only when a camera application is operating. The 8MB of the memory for the JPEG device is used only when the camera is used to take pictures.

Each device's memory region is exploited as a data exchange channel between a user-level application and its owner device. For example of the JPEG device, a camera application opens a `/dev/s3c-jpeg` file when a user presses the shutter button on the screen. After that, the application calls `mmap` system call to map the JPEG device's reserved memory region to the application's address space. The application puts raw image data taken from the camera device into the mmaped space and gets a JPEG-formatted image. After the application saves the picture into a JPEG-formatted file, the device file is closed.

2.2 Motivation

Smart devices, such as smartphones, tablets and smart TVs, are becoming general-purpose systems by adopting open platform environments. They provide not only their own functions, such as calling, sending SMS and watching TV, but also a variety of functions, such as web-browsing, taking pictures and accessing social networks. Falaki et al. [10] characterized smartphone users' application usage as shown in Table 3. Nielsen company [25] also reported statistics of smartphone applications based on more categories as shown in Table 4. From the two tables, mobile activities

Category	Popularity
Communication (email, SMS, IM, etc)	44%
Browsing (web browser, SNS apps, etc)	10%
System (file explorer, etc)	5%
Games	2%
Maps	5%
Media (pictures, music, videos, etc)	5%
Productive (office, PDF reader, etc)	5%
Others	11%

Table 3: Relative popularity of each application category in Android users in 2009 [10]

Category	Used portion	Category	Used portion
Games	65%	Sports	30%
Music	45%	Communication	25%
SNS	54%	Banking/Finance	31%
News/Weather	56%	Shopping/Retail	29%
Maps/Navi./Search	55%	Productivity	30%
Video/Movies	25%	Travel/Lifestyle	21%
Entertainment/Food	38%		

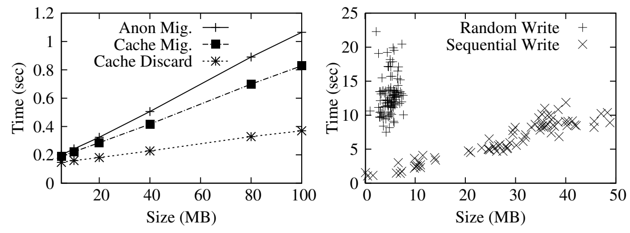
Table 4: Categories of applications used in the past 30 days [25] at Q4 2009

spread across diverse categories. The category that includes video playback or taking pictures is significantly low; the portion is less than 5% of the total application usage. Although some users could have higher usage of the multimedia applications, we believe that using smartphones only for multimedia activities is rare. Accordingly, the reserved memory regions for the multimedia acceleration devices are not fully utilized in smartphone activity.

Contiguous Memory Allocation [27, 8] reduces the reservation inefficiency by enabling the kernel to exploit reserved memory regions when devices are idle. The pages being able to reside in the reserved memory are limited to movable pages (e.g., anonymous pages, such as process stack and heap, and page cache pages). Because the movable pages can be migrated to other memory regions (e.g., normal memory), each reserved memory region is guaranteed to be reallocated to its owner device.

When on-demand reservation is available, the reserved memory is additional but temporal memory. For the memory efficiency [19], it is crucial to keep as many important (necessary) pages as possible in memory even if the reserved memory is reallocated to devices. Otherwise, additional I/Os to read the necessary pages could make system slower. The CMA approach spontaneously follows this principle by migrating all pages in the reserved region to other regions. For the page migrations, the page allocator in the kernel replaces page frames that store the least important data in the system. The reclaimed page frames are provided as destination pages of the page migrations.

The CMA approach, however, can compromise the end-user latency. The on-demand memory reservation time (or memory reallocation time for device) directly increases the launch time of applications that use the acceleration devices. Depending on the characteristics of pages in device-reserved memory, the on-demand reservation time can increase from a few seconds to tens of seconds [18]. The two upper lines in Figure 1(a) depict the time to migrate pages in a reserved memory region in the Nexus S smartphone. For example



(a) Migration and discard cost (b) Write-back cost

Figure 1: Time cost of on-demand reservation

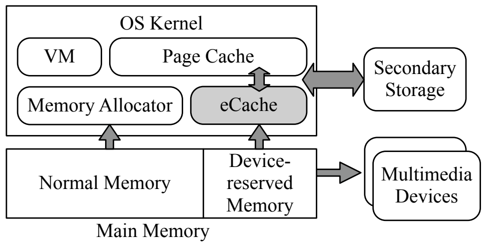


Figure 2: Memory Usage during the normal workload running

of the mfc video decoder, migrating 70MB of the memory takes from 0.6 seconds (when the region is full of clean page caches) to 0.8 seconds (full of anonymous pages). As the time to launch a video application is approximately 0.3 seconds, the migration time makes the launch time increase by a factor of from three to four due to the on-demand reservation. The migration not only requires the explicit latency of memory copies but also incurs implicit overheads such as CPU cache pollution due to the memory copy operations.

The on-demand reservation time becomes more serious when the reserved region contains dirty pages. A write-back page, a dirty page being in the middle of write-back I/O, is neither movable nor discardable until the write-back I/O for the page is finished. Accordingly, if a reserved region is filled with many dirty pages, it could take much more time to complete the on-demand reservation operation. Figure 1(b) shows the time to reclaim a reserved memory region which contains the page cache pages for Sysbench file I/O benchmark's input files. Because the number of write-back pages in the region is the main culprit of increasing reclamation time, we varied the size of write-back pages in the x-axis. We ran 100 reclamations repetitively and measured the time of each operation. As shown in the figure, when the number of write-back pages increases, the reclamation latency increases ranging from 1 second (0.5KB of sequential write pages) to 22 seconds (2MB of random write pages). This additional delay can lead to user dissatisfaction with acceleration device-dependent applications.

The Rental Memory, our previous approach, minimizes the on-demand reservation time by limiting the types of pages to clean page caches among the movable pages [18]. When on-demand reservation occurs, it discards all clean pages in the reserved memory region. Since discarding clean page caches is less costly than migrating pages as shown in Figure 1(a), the on-demand reservation time can be reduced to hundreds of milliseconds. This approach, however, can degrade the memory efficiency of the system. Since the discard operation does not consider the importance of pages, such as the pages in current working set, in the reserved region, discarding the important pages in the region will cause read I/Os after the on-demand reservation completed. Accordingly, the additional read I/O operations become the expense of achieving the short launch time of applications that depend on the acceleration devices.

In summary, providing on-demand reservation to devices increases the available memory size in the system. However, the on-demand reservation time can compromise the end-user latency if the reserved memory is not carefully used. Meanwhile, minimizing the on-demand reservation time can degrade the system's efficiency. Accordingly, we argue that the device-reserved memory should be managed differently in order to minimize the on-demand reservation time while preserving the memory efficiency of the system.

3. DEVICE-RESERVED MEMORY AS AN EVICTION-BASED FILE CACHE

The main goals of our scheme are (1) to lower the on-demand reservation cost for user's satisfaction, and (2) to provide on-demand reservation without degrading the memory efficiency by carefully managing on-demand reservation. In order to achieve our goal, we propose a scheme that uses the device-reserved memory as an eviction-based file cache. The eviction-based exclusive cache management scheme [7] is originally designed for storage cache management. The eviction-based cache resides between an upper-level cache (e.g., page cache) and lower level storage, and caches data evicted from the upper-level cache. Evicted data is data flushed from the upper-level cache by a page frame replacement. The main characteristic of the eviction-based cache management scheme is that it always stores less important data than those stored in the upper-level cache. In addition, an eviction-based cache increases the effective size of the upper-level cache by being exclusively managed [7, 6, 28].

Device-reserved memory is a good target of the eviction-based exclusive cache, which is referred to as *eCache* in the rest of this paper. The characteristic of storing less important data can contribute to preserving the memory efficiency. Since the eCache always stores less important data, on-demand reservation makes the less important data to be discarded from memory. Accordingly the on-demand reservation becomes simple, and the on-demand reservation time also can be minimized. When a device becomes idle again, the memory region provided to the device is returned to the eCache to cache evicted data. When all devices, which contribute their reserved memory regions to the eCache, become activated the system state is the same as when the static reservation approach is used. In addition, by managing the eCache exclusively, it increases the effective size of the in-memory file cache. The page frames in the eCache are still in the same memory chipset in the system. Accordingly, the cached data in the eCache can be accessed almost as fast as those in the kernel page cache. Since the kernel always synchronizes data in secondary storage with modified one in its page cache prior to evicting the data, discarding stored data in the eCache does not violate the consistency of data between the kernel page cache, the eCache and the secondary storage. Figure 2 depicts the overview of the eCache.

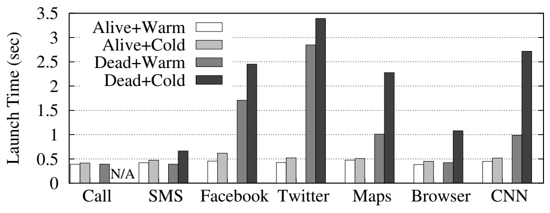


Figure 3: Slowdown of application launch time with absence of library page caches

3.1 Why File Cache?

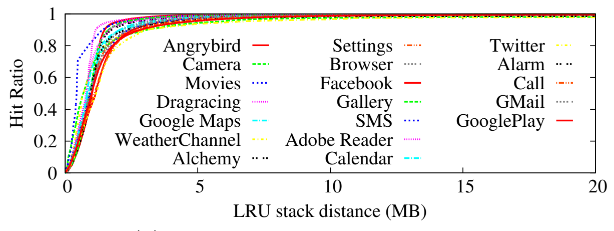
As an answer to the question of which types of data can reside in the eCache, anonymous pages are out of the target. Since many embedded systems are not equipped with swap storage, anonymous pages cannot be evicted from the kernel page cache. Without considering the eviction-based placement, placing anonymous pages in the additional memory can incur additional overhead. As we described in Section 2.2, page migration itself is overhead in terms of the on-demand reservation time. In addition, when considering that the device-reserved memory is temporal memory, keeping anonymous pages in the eCache makes anonymous pages to be overcommitted against the normal memory size. When the size of anonymous pages is larger than the size of normal memory, killing low priority tasks lowers the availability of the system. Finally, we still have a large size normal memory that can allocate anonymous pages sufficiently. Even though the additional memory cannot be used for anonymous pages, the total anonymous page footprint is the same as those with the static reservation approach.

Meanwhile, allocating page caches on the additional memory can increase the performance of the system by absorbing more read I/O traffics. Practically, many commodity OSes provide as large page caches as possible if memory is sufficient. In addition, Android applications are well-known to be tightly coupled with the libraries since the framework libraries in Android provide diverse functions and the same look and feel [14]. Accordingly, the more library files are cached, the less major faults occur while applications are starting. In order to detail the impact of page caches to the launch performance of applications, we measured the slowdown of application launch time when page caches for an application are not in the memory. In the Android system, the application (*activity* in Android) can be started from two cases: when the application (or process) is dead, and when the application is alive. We evaluated the application launch time in the two cases. As shown in Figure 3, keeping page caches in memory significantly improves launch performance by from 19% to 175% when an application is dead. Even when an application is alive, the launch performance is improved by from 6% to 36%.

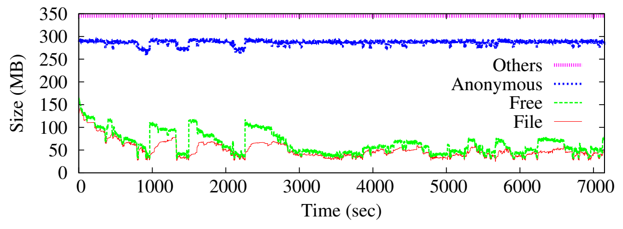
The following subsection describes how the eCache stores file data evicted from the kernel page cache and how the cached data is promoted to the page cache. The next subsection describes how a contiguous region of the eCache is provided to devices when an on-demand reservation occurs.

3.2 eCache Management

The eCache stores file data at page granularity that is the same management unit in the kernel page cache. Each file page is accessed by a unique key consisting of *file system id*, *inode* number and *offset* in the inode. During file page retrieval, the file system id and the inode number are hashed, and the corresponding hash entry points the root of a radix tree. The radix tree stores references to cached pages in the file by using each page's offset as an index.



(a) Page cache miss ratio curve



(b) Memory usage (stacked)

Figure 4: Page cache miss ratio curve and memory usage and in the Normal workload in Section 5.2

In order to cooperate with the page cache in the kernel, the eCache provides four interfaces: `put_page()`, `get_page()`, `flush_page()`, and `flush_inode()`. When a file page in the page cache is a target of page replacement in the kernel, the `put_page()` is called. The function copies the evicted data in the page cache page into a free page frame in the eCache, and establishes indexing structures to the newly stored page. The `get_page()` function is called when the kernel requires to read a file page that is not in the page cache. When the required file page exists in the eCache, its data is copied to the page cache page. Since the eCache is managed exclusively to the kernel page cache, the cached page in the eCache is discarded (or removed). `flush_page()` and `flush_inode()` are invoked when some pages in a file or a whole file is removed from a file system. The two functions flush one or all of the cached data of the specified file in the eCache, respectively. Therefore, the data consistency is not violated between the page cache, the eCache and the secondary storage.

Among many replacement algorithms, we used LRU-based page replacement in the eCache. We believe that any other page placement algorithms can also be used. All page frames in the eCache are linked to form a single LRU list. When a new file page is evicted from the page cache, the page frame at the end of the list (or at the LRU position) is freed (if needed) and stores the content of the evicted file page. Then, the page frame is moved to the beginning of the list (or MRU position). When a page is accessed via the `get_page()` function, the cached data in the corresponding page frame is discarded and the page frame is moved to the LRU position. The LRU-based page replacement is well-known to be effective when the cache employs eviction-based placement and is exclusively managed [7].

As data in the eCache is indirectly accessed, thrashing problem can happen when a file page is frequently evicted from and promoted to the page cache. In Android environment, the thrashing phenomenon unlikely occurs for two reasons. First, many Android applications have less than 10MB of file cache working set sizes. Figure 4(a) shows the miss ratio curve of page cache accesses for various Android applications. The tested workloads are described in Section 5.2. We also added some realistic behaviors (e.g., playing games) after launching each application. The memory-mapped page cache accesses are sampled at every 20 ms. As shown in the figure, the hit ratios are saturated between 5MB and 10MB. Second, the *lowmemorykiller* in Android system preserves a certain amount of page cache in system. It starts killing background applications to reclaim memory when both free memory and cached pages are under 32MB, respectively, in usual configuration. Accordingly, the page cache size in the kernel is kept around 40MB as shown in the *file* line in Figure 4(b). Accordingly, most of the pages for a foreground application can be located in the page cache.

3.3 On-demand Reservation

From the perspective of device, the eCache is a memory allocation pool when on-demand reservation is needed for devices. It is important to note that the eCache consists of contiguous page frames and the size is the sum of each device's reserved memory size. When a device requests memory allocation using the `eCache_alloc()` function, a naive approach is to statically fix the location of each device's memory region for the contiguous memory allocation. When a device requests its on-demand reservation to the eCache, all of the cached data in its fixed region are discarded and the page frames comprising the fixed region are removed from the LRU list.

This naive approach, however, could degrade the local memory efficiency (or caching efficiency). A device-fixed region is always smaller than the size of the eCache when multiple regions of devices comprise the eCache. Accordingly, a device-fixed region could contain many important pages that have higher chances to be accessed than the other pages in the rest of the regions in the eCache. Accordingly, this static region selection approach will likely incur additional read I/O operations. As an alternative approach, dynamic region selection can avoid the additional read I/Os albeit the dynamic approach could lead to a slight increase of on-demand reservation time.

For this dynamic region selection, it is also important to avoid external fragmentation in the eCache region. Since the size of the eCache is static, an improper region selection will prevent another device's on-demand reservation when the reservation requests are issued in about the same time. For example, the eCache consists of two devices' memory regions whose sizes are two and three contiguous page frames. If the former device's memory allocation is done on the second and third page frames in the eCache, remaining page frames are separately located and non-contiguous. Because of the fragmentation, a future allocation request for three contiguous pages cannot be handled.

In order to eliminate this fragmentation problem, we restrict that the on-demand reservation can be handled on one of the predefined regions (candidate regions) for each device. For example, if we have three devices (mfc, fimc and jpeg), which requires six, four and three contiguous page frames, respectively, we have 6(=3!) memory allocation cases, the permutation of the three devices, as shown in the top side of

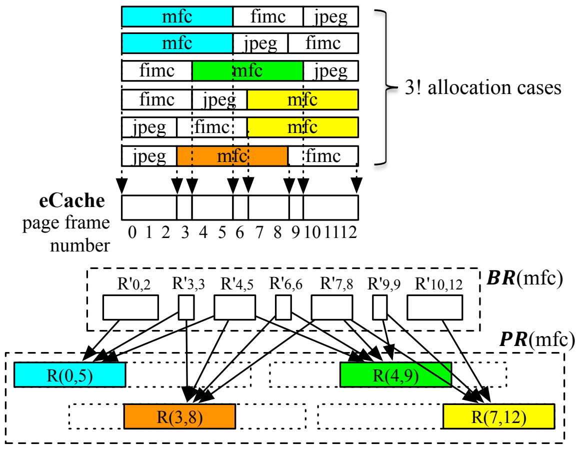


Figure 5: Cost calculation for mfc allocation case. mfc size is 6, fimc size 4 and jpeg 3.

Figure 5. We define predefined memory allocation regions for a device d as $\mathbf{PR}(d)$. $\mathbf{PR}(d)$ is a set of memory regions each of which is denoted as $R_{a,b}$ where $R_{a,b}$ denotes page frames whose indices are from a to b. Hence, the predefined memory allocation regions for the mfc device in the figure are:

$$\mathbf{PR}(\text{mfc}) = \{R_{0,5}, R_{3,8}, R_{4,9}, R_{7,12}\}$$

From the predefined regions for each device, it is important to choose a region that will expense minimal cost, when the region is allocated. In this regard, we define C_i that is a potential cost (e.g., probability for future read) when the data in the ith page in the eCache is discarded for device memory allocation. Then, the potential cost of one region $R_{a,b}$ is calculated by the following equation.

$$C(R_{a,b}) = \sum_{i=a}^{b} C_i \tag{1}$$

Accordingly, a device d's memory allocation with minimal cost can be accomplished by finding a $R_{a,b}$ with minimum $C(R_{a,b})$ in $\mathbf{PR}(d)$. In order to avoid redundant calculation for the costs of overlapped page frames, each R in $\mathbf{PR}$ is divided into disjoint subregions. We define a base region of a device d as $\mathbf{BR}(d)$ that has an element $R'_{a,b}$, which is a set of pages from ath to bth, as an element. Then, a predefined region is divided into many subregions in the base region as shown in the bottom side of Figure 5. For example,

$$\mathbf{BR}(\text{mfc}) = \{R'_{0,2}, R'_{3,3}, R'_{4,5}, R'_{6,6}, R'_{7,8}, R'_{9,9}, R'_{10,12}\}$$

$$\text{and, } R_{0,5} = \{R'_{0,2}, R'_{3,3}, R'_{4,5}\}, R_{3,8} = ...$$

Then, the equation (1) can be transformed to:

$$C(R_{a,b}) = \sum_{R'_{a,b} \in R} C(R'_{a,b}) = \sum_{R'_{a,b} \in R} (\sum_{i=a}^{b} C_i) \tag{2}$$

By calculating the $C(R'_{a,b})$ in $\mathbf{BR}(d)$, $C(R_{a,b})$ can be calculated without the redundant cost calculation. The time complexity for one device memory allocation becomes $O(n + pb + s)$. n is the size of the eCache and s is the size of requested region for a device. When m is the number of devices that require on-demand memory reservation from the eCache, p is $|\mathbf{PR}| = \sum_{i=0}^{m-1}\binom{m-1}{i}$ and b is $|\mathbf{BR}|$. The two values are dependent to m. In our prototype, m is 4, p is 8, and the maximum b is 8. In general, m is relatively small, and therefore, the term pb can be ignored. Therefore, the time complexity is $O(n + s)$.

The effectiveness of the algorithm shown above depends on how precisely C_i represents the probability to cause read I/O when the data in the page is discarded. The main property of C_i is that the important page should have larger value. An LRU distance, which is an offset from the LRU position in the LRU list, is a proper candidate for C_i. An LRU distance of ith page is denoted as l_i; this value is similar to the inverse of the stack distances in [21, 2]. When a page is recently added in the eCache, it is located on the MRU position; the recently added page has the highest probability to be accessed, and the page's LRU distance is the highest in the eCache. By using l_i, C_i can be determined as follows.

$$C_i = \begin{cases} \infty & \text{if } i\text{th page is allocated to other devices} \\ l_i & \text{otherwise} \end{cases}$$

By assigning infinity to the cost of pages that are allocated to other devices, the on-demand reservation does not choose predefined regions that already have one or more allocated pages for the other devices. Each page's LRU distance value is calculated by one traversal of the LRU list when device memory allocation is requested. Accordingly, the time complexity of device memory allocation is still linear.

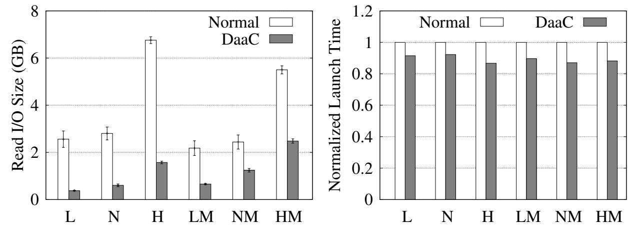
Finally, When a device becomes idle, its allocated memory region is returned to the eCache through the `eCache_free()` function. The returned page frames are inserted in to the LRU list.

4. IMPLEMENTATION

We implemented our scheme in Linux Kernel 2.6.35.7 with the Nexus S smartphone on Android Open Source Project [12]. We modified the device drivers in the kernel to support run-time position changes of the base address of each device's reserved memory. The modification requires only a few lines of codes, which are in charge of notifying the base address of reserved memory to its device. We changed the sequence of each device's reserved memory in order to make the eCache one large contiguous region. Compared to Table 2, the reservation sequence becomes *mfc(fw)*, *mfc0*, *mfc1*, *fimc0*, *fimc2*, *jpeg* and *fimc1*. We regarded fimc0 and fimc2 as one *fimc* whose size is 12MB. The eCache size is 90MB for mfc0, mfc1, fimc and jpeg.

The implemented eviction interface is similar to Clean-Cache [20]. For device memory allocation, the two interfaces, `eCache_alloc()` and `eCache_free()`, are called when a device file is opened and when a device file is closed, respectively.

When `eCache_alloc()` occurs, the selected page frames for device memory allocation are freed from the eCache in a lazy manner. Hence, we only invalidate page frames that are allocated to devices. Subsequent operations accessing the page frames are in charge of freeing the page frames actually. By using this lazy discard operation, the cost of freeing device-allocated page frames is amortized to the cost of subsequent eCache access operations. Due to the exclusive management of the eCache, the lazy discard operation incurs minimal impact on the subsequent operations. For example, `get_page()` always tries to remove the index to a target page regardless of whether the target page is in the eCache or not. Accordingly, even if the target page is invalid, the `get_page()` works the same as when the target page is valid, but returns failure instead of success.



(a) Flash read size (b) Normalized launch time

Figure 6: Flash read size and normalized application launch time in all workloads

5. EVALUATION

5.1 Environment

Nexus S, our target platform, is equipped with ARM Cortex A8 1GHz processor, 512MB RAM, and 16GB internal flash storage. We used custom firmware, which is built from Android AOSP 2.3.7_r1 with a *userdebug* configuration.

5.2 Methodology

We used six different types of workloads as described in a technical report [17]. Briefly, the light workload (L) consists of 12 essential applications and one game application. The normal workload (N) contains additional applications such as social network applications, additional games and a document reader. The heavy workload (H) includes other useful applications in addition to the normal workload. Since the on-demand reservation frequency depends on the portions of multimedia applications, such as Movies, YouTube and Camera, we added three workloads, light multimedia (LM), normal multimedia (NM), and heavy multimedia (HM), that intensively run the three multimedia applications.

We used read I/O size and application *launch time* [9] as our performance metrics. The application launch time is the time elapsed between when an *intent*, a message to invoke an application's activity, is initiated and when the application's main routine (i.e., onResume()) completes making the application window visible [23]. Only if an application is newly started, the launch time of the application is reported by *ActivityManager* through a *logcat* command. We modified a source code to make ActivityManager show the launch time even when an application is alive.

5.3 Performance

We first measured the basic performance of the eCache and the flash device in Nexus S. The performance of the eCache is the average time of completing `get_page()` in the all workloads. Flash read performance is measured by reading 4KB of page directly. Since the eCache is an in-memory cache, the average 4KB read response time in the eCache is 0.07 msec which is approximately 20 times faster than the flash read (1.51 msec).

Figure 6(a) shows the size of read I/O that occurred during each workload running. We ran each workload four times and depicted the average read I/O size. *Normal* denotes the static reservation approach for devices. *DaaC* is

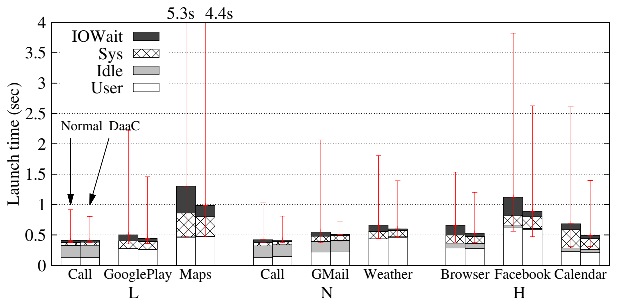
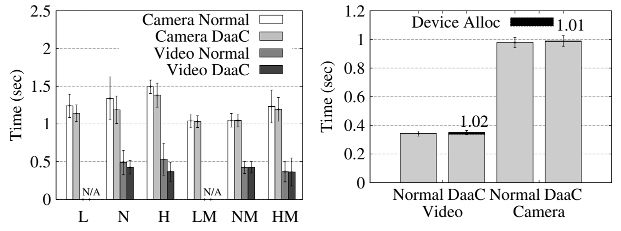


Figure 7: Breakdown of application launch time



(a) With the workloads

(b) Solo run

Figure 8: Launch time of device-dependent applications

the on-demand reservation using our scheme. As shown in the figure, our scheme significantly reduces read I/O by from 76% to 85% compared to the normal case. Since the eCache stores a large amount of file caches, read I/O traffic is largely absorbed by the eCache. The multimedia-intensive workloads (LM, NM, and HM) performed relatively larger amount of read I/O operations than non-multimedia-intensive ones (L, N, and H). Because the acceleration device-dependent applications (i.e., Camera, Movies, and YouTube) are more frequently invoked, many cached data in the eCache is frequently invalidated due to the on-demand memory reservations. The eCache, however, is quickly refilled with evicted data when device-dependent applications are finished and a subsequent application is launched.

Figure 6(b) shows the geometric mean of the normalized launch times of all applications for each workload. As our scheme reduces many read I/O operations, the launch time is decreased by 8% to 16% compared to the normal case. The main benefit of the eCache during application launch comes from read I/O reduction. In order to show the time reduction in more detail, we break down the launch time of applications into *user*, *idle*, *system*, and *I/O-wait* time. The user and idle time denote how many CPU cycles an application spends in user and idle contexts, respectively. The system time shows how many CPU cycles used at the kernel context including I/O handling. The I/O-wait time indicates how many CPU cycles an application waits for read I/O operations to complete. Since the eCache contributes to the reduction in read I/O and I/O-related kernel operations, the system time and I/O-wait time are reduced from total launch time by the eCache. We selected three applications based on the frequency of executions from the three workloads (Light, Normal and Heavy) in Figure 7. The figure also includes the minimum and maximum launch time of the applications. Except for the Call application, the applications showed reduced system and I/O-wait time by 22%-42% compared to those of the normal case. As the portion of system and I/O wait time varies across the applications, the launch time of the applications is reduced by 2%-29%.

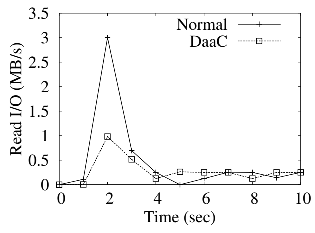


Figure 9: Read I/O occurred while the Movies application is launching

Get	Hit	Type
0.55	0.55	Dalvik Caches (framework, core, services, Gallery3D.apk, et al)
1.30	1.16	Libraries (libc, libdvm, libcutils, libcstageflight, libcaudioflinger, libOMX.SEC.AVC.Decoder, et al)
0.83	0.81	Framework Resource (framework-res.apk)
0.38	0.02	Etc. (sh, linker, mp4 file)

Table 5: The size of files hit in the eCache when Movies application is launching (in MB)

Compared to the read I/O reduction, the launch performance improvement is not significant. The reason is that only less than half of read I/O reduction contributes to the application launch performance. Approximately 60% of read I/O operations occurred during the operations other than application launches. Accordingly, the other user interactive operations, such as menu changes, loading the other screen, will also benefit from the reduced I/O operations.

Since our scheme utilizes device-reserved memory for the eCache, we have tradeoff between the gains from caching files and expenses of on-demand reservation time. In order to reveal this tradeoff, we first show the activity time of Movies and Camera applications in each workload. The launch time of the two applications includes on-demand reservation of *mfc0* and *mfc1*, and *fimc*, respectively. Figure 8(a) shows the average launch time of each application. Note that the light workload does not include the Movies application. Interestingly, the launch performance with our scheme is better than those with the static reservation approach. The reason is that the device memory allocation occurs in the middle of launching the application. Therefore, many file accesses can be hit in the eCache.

Figure 9 shows the read I/O throughput (MB/s) while the Movies application is starting. We note that the previous Movies application launch was 2,000 seconds before this Movies application launch; hence the files for the application were likely to be evicted from the page cache and even from the eCache. As shown in the figure, the normal case shows more read I/O operations while the application is launching (the time period between 1 second and 3 second). Our scheme, however, exhibits less read I/O operations because many of those are absorbed by the eCache. Table 5 shows the summary of actual read requests to the eCache when the Movie application is starting. Not only commonly used files (i.e., framework dependent files and libraries) but also application-specific files (e.g., video decoder-dependent libraries) are retrieved from the eCache.

In order to expose the pure cost of the on-demand reservation in our scheme, we ran each of two applications without any other applications. Each application is executed 100 times, and we measured the average launch time. Figure 8(b) shows the breakdown of the launch time of the Movies and Camera applications. As shown in the figure, the additional time for device memory allocation slightly increases the application launch time by 2% and 1% respectively. As the unit cost of device memory allocation is extremely low, consisting of a linked list traversal, integer calculations and setting tombstones, the device allocation time for both *mfc0* and *mfc1* is 11ms (5.8ms + 4.8ms) and the time for *fimc* is 4ms. Recall that the minimum cost to provide 70MB of memory using the on-demand reservation in the previous approaches is 0.3 seconds as shown in Figure 1(a). Accordingly, our scheme provides the on-demand reservation at least 30 times faster.

5.4 Policy Analysis

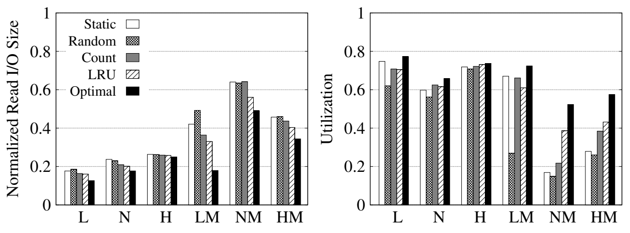
In this subsection, we analyze the LRU position-aware region selection during the on-demand memory reservation. Our policy is denoted as *LRU* that uses $C_i = l_i$ for the page cost function. For comparison, we added four different policies. *Static* fixes the memory region for each device. In our prototype, we have 24(=4!) different static allocation cases. We provided the best case result among the 24 different cases. *Random* randomly selects a region among the predefined regions for a device. *Count* assigns the same cost (1) to pages which are in use in the eCache for caching. Hence,

$$C_i = \begin{cases} \infty & \text{if } i\text{th page is allocated to other devices} \\ 1 & \text{if } i\text{th page is allocated for caching} \\ 0 & \text{otherwise} \end{cases}$$

Finally, *Optimal* shows an optimal case, which is practically impossible without knowing the future accesses to the eCache.

We evaluated each policy with trace-driven simulation in order to minimize runtime variation. A trace includes all accesses to the eCache through the interfaces described in Section 3. We gathered the trace from one of each real workload execution in the previous subsection. Figure 10 shows the normalized *read I/O size* and *average utilization* of the eCache. The read I/O size for each workload is calculated based on the number of `get_page()` misses and is represented in a normalized value to the normal case. The average utilization shows how many cached pages are in the eCache. The average utilization is the average of the number of valid pages divided by the number of page frames in the eCache at every second.

Across the workloads, multimedia intensive workloads (LM, NM, HM) have performed more read I/O operations because more frequent on-demand reservation requests for devices discard many cached files in the eCache as illustrated in Figure 10(a). This result is consistent to the result shown in Figure 6(a). The cost-based policies (count and LRU) perform less read I/O operations than static and random policies. The static and count policies show higher eCache utilization in workload L, N, and LM than the other policies in Figure 10(b). But, increasing the utilization of the eCache is not related to the efficiency of the eCache because the



(a) Normalized Read I/O (b) Average Utilization

Figure 10: Normalized read I/O size, and average eCache utilization with varying device memory allocation policy

static and count policies perform more read I/O operations than the LRU policy in Figure 10(a). Among the cost-based policies, our LRU policy performs better than the count policy by 2%-13% in terms of read I/O reduction. This result indicates that even though the LRU positions of contiguous pages are aggregated, the aggregated value is sufficient to represent the potential cost when the contiguous pages are chosen for the victim of on-demand memory reservation. Accordingly, our suggested policy performs better than the other policies in the tested workloads.

6. RELATED WORK

In this section, we discuss our work with related studies. The waste of reserved memory for device has been addressed previously. At the early stage of Contiguous Memory Allocation (CMA) [8], it tries to minimize reservation footprint by sharing reserved memory regions between devices. The rationale of this scheme is that not all devices are activated simultaneously. This scheme is orthogonal to our approach and we believe that our work can be complementary to this scheme. The later version of CMA [24, 27] increases the reserved memory's utilization by allowing kernel to use the page frames in the reserved memory region similarly to Hotplug Memory [26, 15]. Movable pages (e.g., page cache pages and anonymous pages) are allowed to be placed in the CMA region. As described in Section 2.2, this approach increases the memory efficiency of system but can sacrifice end-user latency.

In contrast, Rental Memory Management [18], our previous approach, utilizes device-reserved memory for clean page caches since keeping dirty page caches in the reserved memory causes unpredictable delay for device memory allocation. Although the additional memory is used for clean caches, its performance is almost comparable to the CMA approach while minimizing the on-demand reservation time under one second. This approach, however, compromises the memory efficiency of system as depicted in Section 2.2. We believe that our scheme will performs better than this approach for two reasons. First, both approaches have the same size of the file cache in memory. Based on the characteristics of applications depicted in Figure 4, the mostly accessed page caches are likely to be in the kernel page cache instead of the eCache because the eCache always contains the least important file caches than those in the page cache. Accordingly, the access of the eCache rarely occurs. Second, our scheme always discards the least important pages in system while this approach does not.

7. CONCLUSION

In resource-constrained embedded systems, such as smartphones, the under-utilized reserved memory is one of the major sources of inefficient resource management. On-demand reservation, which allows the system to exploit the reserved memory during an idle time of owner devices, sheds lights on the way of increasing the performance of the system while guaranteeing devices the use of reserved memory space.

We proposed a novel on-demand reservation approach, named *eCache*, that maximizes the memory efficiency of the system and minimizes the on-demand reservation time for end-user latency. By using the eCache, the system can greatly reduce the number of read I/O operations and increase the launch performance of applications by from 8% to 16%. With this performance improvement, the nature of eviction-based placement spontaneously maximizes the memory efficiency of the system. In addition, the on-demand reservation time can be minimized to millisecond level. This unrecognizable latency may make system more transparent in comparison to the system with the static-reservation approach.

8. ACKNOWLEDGMENTS

This work was 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)

9. REFERENCES

[1] D. Abramson, J. Jackson, S. Muthrasanallur, G. Neiger, G. Regnier, R. Sankaran, I. Schoinas, R. Uhlig, B. Vembu, and J. W. and. Intel virtualization technology for directed I/O. *Intel Technology Journal*, 10(3)(3), August 2006.

[2] G. Almási, C. Caşcaval, and D. A. Padua. Calculating stack distances efficiently. *SIGPLAN Not.*, 38(2 supplement):37–43, June 2002.

[3] AMD. IOMMU architectural specification. `http://support.amd.com/us/ProcessorTechDocs/48882.pdf`, March 2011.

[4] N. Amit, M. Ben-Yehuda, and B.-A. Yassour. IOMMU: Strategies for mitigating the IOTLB bottleneck. In *Proc. of WIOSCA'10*, 2010.

[5] M. Ben-Yehuda, J. Xenidis, M. Ostrowski, K. Rister, A. Bruemmer, and L. Van Doorn. The price of safety: Evaluating IOMMU performance. In *Proc. of OLS '07*, pages 71–86, July 2007.

[6] Z. Chen, Y. Zhang, Y. Zhou, H. Scott, and B. Schiefer. Empirical evaluation of multi-level buffer cache collaboration for storage systems. In *Proc. of SIGMETRICS '05*, pages 145–156, 2005.

[7] Z. Chen, Y. Zhou, and K. Li. Eviction-based cache placement for storage caches. In *Proc. of USENIX ATC'03*, pages 269–282, Jun 2003.

[8] J. Corbet. Contiguous memory allocation for drivers. `http://lwn.net/Articles/396702/`, July 2010.

[9] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones. In *Proc. of OSDI'10*, pages 1–6, 2010.

[10] H. Falaki, R. Mahajan, S. Kandula, D. Lymberopoulos, R. Govindan, and D. Estrin. Diversity in smartphone usage. In *Proc. of MobiSys'10*, pages 179–194, 2010.

[11] Gartner. Gartner highlights key predictions for it organizations and users in 2010 and beyond. `http://www.gartner.com/it/page.jsp?id=1278413`, January 2010.

[12] Google. Android open source project. `http://source.android.com/`, 2012.

[13] Google. Nexus s. `http://www.android.com/devices/detail/nexus-s`, 2012.

[14] A. Gutierrez, R. Dreslinski, T. Wenisch, T. Mudge, A. Saidi, C. Emmons, and N. Paver. Full-system analysis and characterization of interactive smartphone applications. In *Proc. of IISWC'11*, pages 81 –90, nov. 2011.

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

[16] IDC. Worldwide smartphone market expected to grow 55% in 2011 and approach shipments of one billion in 2015, according to idc. `http://www.idc.com/getdoc.jsp?containerId=prUS 22871611`, June 2011.

[17] J. Jeong, H. Kim, J. Hwang, J. Lee, and S. Maeng. Device-reserved memory as an eviction-based file cache. Technical Report CS-TR-2012-360, Korea Advanced Institute of Science and Technology, July 2012.

[18] J. Jeong, H. Kim, J. Hwang, J. Lee, and S. Maeng. Rigorous rental memory management for embedded systems. *ACM Trans. Embed. Comput. Syst.*, accepted.

[19] G. S. Joachim. Memory efficiency. *J. ACM*, 6(2):172–175, Apr. 1959.

[20] D. Magenheimer, C. Mason, D. McCracken, and K. Hackel. Transcendent memory and linux. In *Proc. of OLS'09*, pages 191–200, 2009.

[21] R. Mattson, J. Gecsei, D. Slutz, and I. Traiger. Evaluation techniques for storage hierarchies. *IBM Systems Journal*, 9(2):78–117, 1970.

[22] R. Mijat and A. Nightingale. Virtualization is coming to a platform near you. `http://www.arm.com/files/pdf/System-MMU-Whitepaper-v8.0.pdf`, 2011.

[23] D. Morrill. Inside the Android application framework. In *Google I/O*, 2008.

[24] M. Nazarewicz. Contiguous memory allocator version 6. `http://lwn.net/Articles/419639/`, December 2010.

[25] Nielsen. The state of mobile apps. `http://blog.nielsen.com/nielsenwire/online_mobile/the-state-of-mobile-apps/`, June 2010.

[26] J. H. Schopp, D. Hansen, M. Kravetz, H. Takahashi, I. Toshihiro, Y. Goto, K. Hiroyuki, M. Tolentino, and B. Picco. Hotplug memory redux. In *Proc. of OLS '05*, 2005.

[27] M. Szyprowski and K. Park. Arm DMA-mapping framework redesign and IOMMU integration. In *Proc. of Embedded Linux Conference Europe*, 2011.

[28] T. M. Wong and J. Wilkes. My cache or yours? making storage more exclusive. In *Proc. of USENIX ATC'02*, pages 161–175, June 2002.