Device-reserved Memory as an Eviction-based File Cache

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

Seungryoul Maeng[**]
KAIST

CS/TR-2012-360

July 25, 2012

K A I S T
Department of Computer Science

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

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

In resource constrained embedded systems, memory reservation, which provides physically contiguous memory allocation to devices (e.g., video decoder and camera), decreases the memory efficiency of the system. The dedicated use of reserved memory by an owner device makes itself more under-utilized when the device is less frequently used. Previous approaches support on-demand reservation to relax the dedicated use of the reserved memory by allowing the kernel to exploit the reserved memory while the owner device is idle. The previous approaches, however, can either incur significant delay to the on-demand reservation or sacrifice the memory efficiency during the on-demand reservation. Due to the delay, an end user could wait for tens of seconds when to use the owner device-dependent functions. When the memory efficiency is compromised, the system will likely incur additional read I/Os.

In this paper, we propose a scheme to use device-reserved memory as an eviction-based file cache called *eCache*. The eCache provides on-demand reservation by discarding the cached data in a contiguous memory region in eCache. From the nature of eviction-based placement policy, the memory efficiency is still preserved during the on-demand reservation because the cached data in eCache is always less important than those in an upper-level cache such as in-kernel page cache. In addition, the on-demand reservation only requires minimal time discarding cached data in eCache. When multiple reserved regions comprise the eCache, cost-based region selection improves the memory efficiency (or caching efficiency) in eCache when on-demand reservation occurs. We implemented eCache on the Nexus S smartphone and evaluated in Android workloads. The evaluation results show that the 21% of additional memory reduces read I/Os by a factor of from two to six as compared to those with memory reservation approach. The application launch performance is also improved by at most 16%. The on-demand reservation time is reduced to a few milliseconds.

Categories and Subject Descriptors

D.4.2 [**Storage Management**]: Main memory

General Terms

Management, Performance

Keywords

memory management, memory reservation

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 [12]. International Data Corporation also predicts that vendors will ship one billion smartphones in 2015 [19]. 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 [11, 32]. Accordingly, the memory space for the dedicated use of the acceleration devices is wasted.

IOMMU [28, 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 can increase the unit cost of system. In addition, managing address mappings for devices increases burden on CPU [4]. Finally, since this approach is hardware dependent [28], its deployment is limited.

Contiguous Memory Allocation [36] and Rental Memory [21] 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 takes 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 [22]. The simple discard operation can be, however, inefficient because it may cause additional page faults.

In this paper, we propose device-reserved memory management 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. By gathering the reserved memory regions for each device, we build 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 increases the memory utilization. Since the management of eCache is eviction-based and exclusive, data stored in 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.

Device	Total	Avail.	Reserved	Usage
Nexus One	512	379	81 (16%/23%)	Video, Camera
Desire HD	768	N/A	157 (20%/N/A)	”
GalaxyS	512	334	115 (22%/34%)	JPEG, Video, Camera
NexusS	512	386	105 (21%/27%)	”
GalaxyS2	1024	816	154 (15%/19%)	”

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

In addition, cost-based memory region selection during the on-demand reservation is suggested to sustain the memory efficiency (or the caching efficiency of 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 eCache. By aggregating the cost of the cached data in each possible 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.

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 in various user workloads which 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. An 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 eCache. While 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. Memory allocation times are 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 with the future direction 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.

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

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 [11]

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/Navigation/Search	55%	Productivity	30%
Video/Movies	25%	Travel/Lifestyle	21%
Entertainment/Food	38%		

Table 4: Categories of applications used in the past 30 days [32] at Q4 2009

2.1 Background

Many state-of-the-art smartphones use memory reservation (static reservation) for their multimedia acceleration devices as shown in Table 1. The table includes a total memory size and an OS-available memory size (in `/proc/meminfo`). We analyzed each device's source code [18, 34, 13] to obtain the detailed reservation information. As shown in 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% as compared to their total memory size.

In more detail of Nexus S [15], which is used in our prototype implementation, it has three devices that reserve six disjoint memory regions. As shown in Table 2, a 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. A 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. A 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 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 memory for the camera device is used only when a camera application is operating. The 8MB of memory for the JPEG device is used only when the camera is used to take pictures.

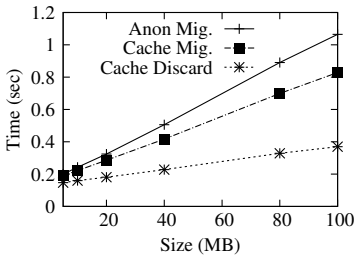
Each device's memory region is exploited as a data exchange channel between a user-level application and its owner device. For example of the JPEG device, a camera application opens a `/dev/s3c-jpeg` file when a user presses a shutter button on a screen. After that, the application calls `mmap` system 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

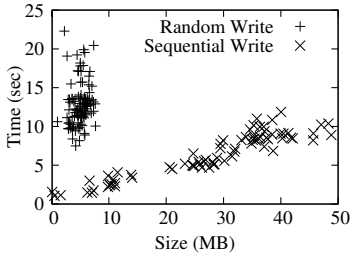
Smart devices, such as smartphones, tablets and smart TVs, are becoming a general-purpose system by adopting open platform environments. They provide not only their own functions, such as calling, sending SMS and watching TV, but also a variety of functions, such as web-browsing, taking pictures and accessing social networks. Falaki et al. [11] analyzed that smartphone users' application usage is characterized as shown in Table 3. Nielsen company [32] also reported statistics of smartphone applications based on more categories as shown in Table 4. From the two tables, mobile activities 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.

Graphic processing units (GPUs) have avoided the memory reservation problem by adopting their own memory management unit. A graphics address remapping table (Gart) in Accelerated Graphic Ports and Dynamic Video Memory Technology [20] provide address translation to make GPUs see a contiguous virtual address space. Those MMUs, however, only support graphic devices' memory address translation. General IOMMU hardware, such as SMMU in ARM, is still in the integration stage [28, 33, 30, 7]. In addition, many optimization efforts reveal that managing mapping between I/O addresses and physical addresses increases CPU load nonetheless in x86 platforms [37, 4, 39]. Finally, the IOMMU approach is hardware-dependent. Many deployed devices cannot be supported by IOMMU.

Contiguous Memory Allocation [36, 8, 31] 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 guaran-



(a) Page migration and discard cost



(b) Write-back cost

Figure 1: Time costs of on-demand reservation

teed 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 [23], 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 makes 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 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 using 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. The above two lines in Figure 1(a) depicts the time to migrate pages in a reserved memory region in the Nexus S smartphone. For example of the mfc video decoder, migrating 70MB of 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.

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 [21]. 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 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 short launch time of applications using acceleration devices.

In summary, providing on-demand reservation to devices increases the available memory size in 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 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. Eviction-based exclusive cache management scheme [6] 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 means data which is 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 [6, 5, 38]

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 leverage preserving the memory efficiency.

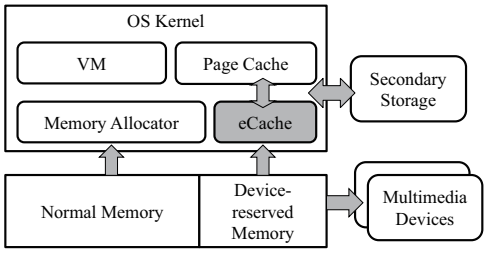


Figure 2: Memory Usage during the normal workload running

Since 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 eCache to cache evicted data. When all devices, which contribute their reserved memory regions to eCache, become activated the system state is the same as when the static reservation approach is used. In addition, by managing eCache exclusively, it increases the effective size of in-memory file cache. The page frames in eCache are still in the same memory chipset in system. Accordingly, the cached data in 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 [24], discarding stored data in eCache does not violate the consistency of data between the kernel page cache, eCache and secondary storage. Figure 2 depicts the overview of eCache.

3.1 Why File Cache?

As an answer to the question of which types of data can reside in 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 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 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 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, since Android applications are well-known to be library dependent [16], keeping more caches in memory can improve the performance of applications. Since the framework libraries in Android provide diverse functions and the same look and feel, most android applications are

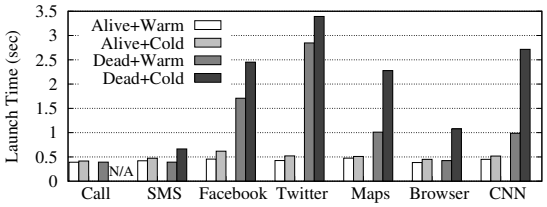


Figure 3: Slowdown of application launch time with absence of library page caches

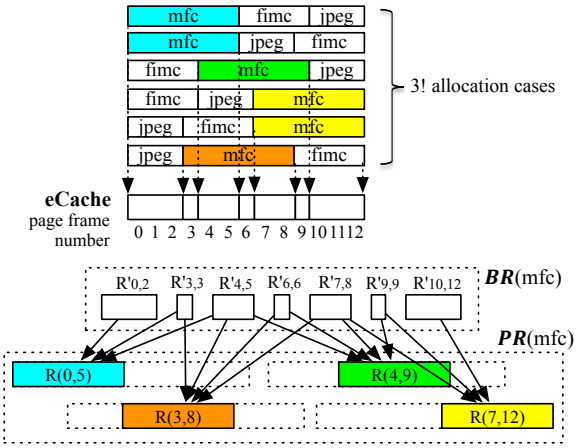


Figure 4: Cost calculation for mfc allocation case. mfc size is 6, fimc size 4 and jpeg 3.

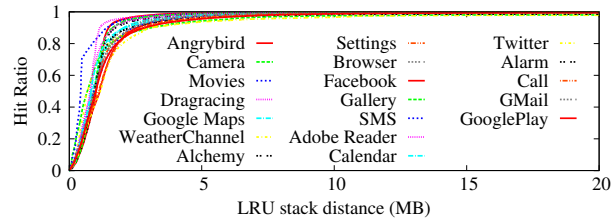
tightly coupled with the libraries. Accordingly, the more library files are cached, the less major faults occur while applications are starting. In order to detail the impact of page caches to the launch performance of applications, we measured the slowdown of application launch time when page caches for an application are not in memory. In the Android system, the application can be started from two cases, when the application 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 270% when an application is dead. Even when an application is alive, the launch performance still benefits from cached library pages.

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

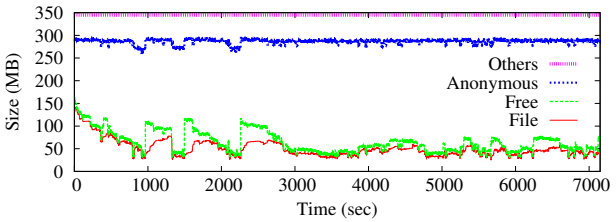
3.2 eCache Management

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, file system id and 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.

In order to cooperate with the page cache in kernel, eCache provides four interfaces: `put_page()`, `get_page()`, `flush_page()`,



(a) Page cache miss ratio curve



(b) Memory usage (stacked)

Figure 5: Page cache miss ratio curve and memory usage and in the Normal workload in Section 5.2

and `flush_inode()`. When a file page in 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 eCache, and establishes indexing structures to the newly stored page. The `get_page()` function is called when system requires to read a file page which is not in the kernel page cache. When the required file page exists in eCache, the stored data in the cached page is copied to the page cache page. Since eCache is managed exclusively to the kernel page cache, the cached page in 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 discard the deleted file's cached pages in eCache. Therefore, the data consistency is not violated between the page cache, eCache and secondary storage.

Among many replacement algorithms, we used LRU-based page replacement in eCache. We believe that any other page placement algorithms can also be used. All page frames in eCache is linked to form a single LRU list. When a new file page is evicted from page cache, the page frame at the end of the list (or LRU position) is freed (if needed) and stores the content of the evicted file page.[1] Then, the page frame is moved to the beginning of the list (or MRU position). When a page is accessed via `get_page()` function, the cached data in page frame is discarded and the page frame is moved to the LRU position. LRU-based page replacement is well-known to be effective when the cache employs eviction-based placement and is exclusively managed [6].

As data in 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 5(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 5(b). Accordingly, most of the page caches for a foreground application can be located in page cache.

[1]The term *remove* is used when the subject is cached page. The term *free* is used when the subject is a page frame in eCache. Both removes the cached data from eCache.

3.3 On-demand Reservation

From the perspective of device, eCache is a memory allocation pool when on-demand reservation is needed for devices. 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 by `ecache_alloc()`, a naive approach is to statically fix the location of each device's memory region. When a device requests its on-demand reservation to eCache, all of the cached pages in its fixed region are removed and the page frames comprising the fixed region are removed from the LRU list.

This naive approach could degrade the local memory efficiency (or caching efficiency). A device-fixed region is always smaller than the size of eCache when multiple regions of devices comprise eCache. Accordingly, a device-fixed region could contain many pages that have more chances to be accessed than the other pages in the rest regions in eCache. Accordingly, the static allocation approach will likely incur additional read I/O operations. As an alternative approach, dynamic region selection can minimize additional read I/Os from the naive approach albeit it could lead to a slight increase of on-demand reservation time. In summary, the dynamic region selection is aimed at increasing the local memory efficiency in eCache whereas the nature of eviction-based placement is aimed at preserving the global memory efficiency in system.

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

In order to eliminate this fragmentation issue, we restrict the possible memory allocation locations (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 3! memory allocation cases, the permutation of the three devices, as shown in the top side of Figure 4. We define possible 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, possible 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 possible regions for each device, it is important to choose a region that will expense minimal cost, containing less important pages, when allocated. In this regard, we define C_i that is a potential cost (e.g., probability for future read I/O) when the ith page in eCache is discarded for device memory allocation. Then, the potential cost of one region $R_{a,b}$ is calculated by the following equation.

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

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 many of $R'_{a,b}$, which is a set of pages from ath to bth, as an element. Then, a possible region is divided into many subregions in the base region as shown in the bottom side of Figure 4. 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:

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

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 eCache and s is the size of requested region by a device. When m is the number of devices that require on-demand memory reservation from 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 above algorithm depends on how precisely C_i represents the probability to cause read I/O when the page is freed. 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 [27, 2]. When a page is recently added in 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 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, possible regions having allocated pages cannot be chosen for the current on-demand reservation. 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 eCache through `ecache_free()` function. The returned page frames from a device are inserted in to the LRU list in order to cache evicted data through `put_page()` calls.

4. IMPLEMENTATION

We implemented our scheme in Linux Kernel 2.6.35.7 with the Nexus S smartphone on Android Open Source Project [13]. We modified the device drivers in the kernel to support runtime 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 physical memory to its device. We changed the sequence of each device's reserved memory in order to make eCache one large contiguous region. As 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 used eviction interface is similar to CleanCache [26]. For device memory allocation, we added two interfaces `ecache_alloc()` and `ecache_free()` that 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 are amortized to the cost of subsequent eCache access operations. Since the eCache is managed exclusively, the amortized cost is also minimal. For example, `get_page()` always tries to remove the index to a target page regardless of whether the target page is in 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.

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. The memory reservation information of a Nexus S smartphone is shown in Table 2. We used custom firmware which is built from Android AOSP 2.3.7_r1 with a *userdebug* configuration.

5.2 Methodology

We used three workloads, light (L), normal (N), and heavy (H) as shown in Table 5 and 6. The light workload consists of 12 essential applications and one game application. The normal workload contains additional applications such as social network applications, additional games and a document reader. The heavy workload includes other useful applications in addition to the normal workload.

The behavior of each application is programmed using *monkeyrunner* in Android framework [14]. The detailed behaviors of the applications are categorized in two cases. One

Category	Applications	
	L, LM	N, NM
Communication	SMS, Call	GMail
Browsing	Browser	Twitter, Facebook
System	Settings	
Games	Angrybird	Alchemy, Dragracing
Maps	Google Maps	
Media	Camera, Gallery	Movies
Productive	Calendar, Alarm, WeatherChannel	Adobe Reader
Others	GooglePlay	

Table 5: Light and Normal user workload description

Category	H, HM
Games	Angrybird, Dragracing, Alchemy
SNS	Twitter, Facebook, Kakaotalk
News/Weather	CNN, Weather
Maps/Navigation/Search	Map, Google Skymap
Browsing	Browser
Video/Movies	Movies, Youtube, Camera
Entertainment/Food	Gallery, GasBuddy
Sports	ESPN Score Center
Communication	GMail, SMS, Call
Shopping/Retail	GooglePlay
Productivity	Adobe Reader, Calendar, Alarm, AstridTask
Travel/Lifestyle	HopStop, iTriage
Others	Advanced Task Killer, Settings

Table 6: Heavy user workload description

Application	L	N	H	LM	NM	HM
Alarm	7.5	4.8	1.5	1.0	3.0	0.8
Angrybird	2.3	0.8	3.5	1.8	0.3	1.5
Browser	9.5	3.0	9.0	3.3	1.5	6.0
Calendar	6.3	5.5	0.8	4.0	1.3	0.8
Call	23.8	16.8	0.8	10.3	6.3	1.3
Camera	3.8	3.3	0.5	52.0	28.3	16.3
Maps	4.8	4.8	4.0	2.8	2.8	2.3
Picture	2.8	1.8	3.0	2.8	1.8	2.5
Setting	3.3	3.3	2.5	3.0	3.0	0.8
SMS	20.3	12.0	0.5	9.0	7.3	1.3
GooglePlay	7.5	7.5	6.0	5.3	5.3	2.5
Weather	7.5	5.0	5.8	4.3	3.5	2.0
Alchemy		0.8	6.0		0.5	1.0
Dragracing		0.8	3.8		1.0	2.3
Facebook		3.3	3.0		1.0	1.8
GMail		15.3	2.3		5.8	0.8
Movies		1.5	2.3		24.8	16.5
Twitter		3.3	4.8		0.8	1.5
Adobe			1.5			1.0
AstridTask			1.5			1.3
Avd T Killer			2.5			1.3
CNN			6.3			2.8
ESPN			4.8			2.0
GasBuddy			7.0			4.5
Skymap			5.0			2.0
HopStop			1.8			2.5
iTriage			3.0			1.0
KakaoTalk			4.3			2.5
YouTube			2.5			17.8

Table 7: The proportion of each application in each user workload

is a simple launch category. Many Android applications show many of desired information on their firstly opened screen. **WeatherChannel** shows a local weather. **CNN**'s main window depicts hot news. **ESPN Score Center** shows user-set favorite teams' scores. **Alarm** shows current alarm settings. **AstridTask** depicts To-do schedules. **Gmail** shows a current email inbox. **Google Skymap** shows celestial objects. This category also includes other applications that are simply launched since it is hard to deterministically control each application's behavior in multiple runs. For example in Angrybird, when the application is alive, it shows the last-played game screen, whereas it displays the initial screen when newly started after it is dead. Accordingly, it is hard to command each application to operate under various conditions in the monkeyrunner environment. These applications are **Angrybird** (game), **Dragracing** (game), **Alchemy** (game), **HopStop** (bus route information), **Advanced Task Killer** (an application management tool), **Call**, **SMS**, **Kakaotalk** (an instant messenger), and **Gallery** (a picture viewer).

The other application category is to add additional commands to each application. The applications in this category perform as follows. **Browser** opens randomly chosen two of the most popular ten websites [9] except for CNN, YouTube and Facebook sites. **PlayStore** opens the main window of Google PlayStore (aka Market) and searches two of six keywords. **Settings** opens two of six setting windows (wireless, applications, display, sound, security and services). **Calendar** shows its monthly schedule and daily schedule sequentially. **Google Maps** randomly shows two of six locations. **Camera** takes two pictures requiring allocations of *fimc* and *jpeg* regions. **Adobe Reader** opens two of ten PDF files; sizes of the files range from 180KB to 2MB. **Movies** plays one of three 720p high-profile MPEG4 files for 20 seconds. This workload requires device memory allocation for *mfc0* and *mfc1* regions. **Faceboook** opens news feed and scrolls down for two times. **Twitter** depicts recent tweets and scrolls down for two times. **YouTube** shows the most popular movies. We used Movies workload after this application started because we cannot make this workload to do the same video playback repeatedly. **iTriage** is a healthcare application and shows disease and medication screens. **GasBuddy** is a gas station search application. We used 'Dallas, TX' as a search keyword.

The light and normal workloads follow the application popularity in Table 3. For example in the light workload, we randomly chose a number between 1 and 100 (the sum of the popularity values). If the number is 45, we run one of the applications in the Browsing category. We repeated this procedure 400 times for each workload to describe a user behavior. Similarly, the heavy workload follows the application popularity in Table 4.[2] The actual proportion per application for each workload is shown in Table 7. Since the on-demand reservation frequency depends on the portions of multimedia applications such as Movies, YouTube and Camera, we added three additional workloads, light multimedia (LM), normal multimedia (NM), and heavy multimedia (HM), each of which increases the sum of proportions of the three applications to 50% as shown in the table.

We used read I/O size and application *launch time* [10] 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 an application window visible [29]. 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 to show the launch time even when the application is alive.

[2]Music and Banking categories are removed and a Browsing category is added with 50% of used portion

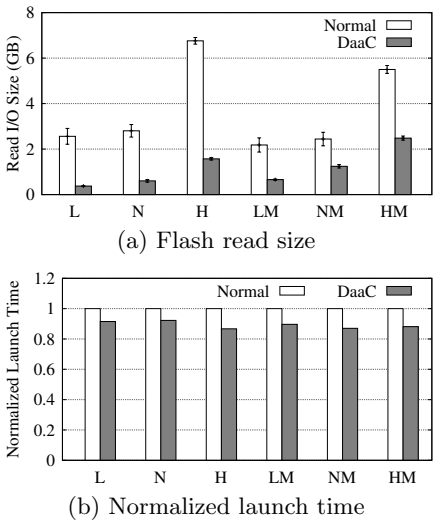


Figure 6: Flash read size and normalized application launch time in all workloads

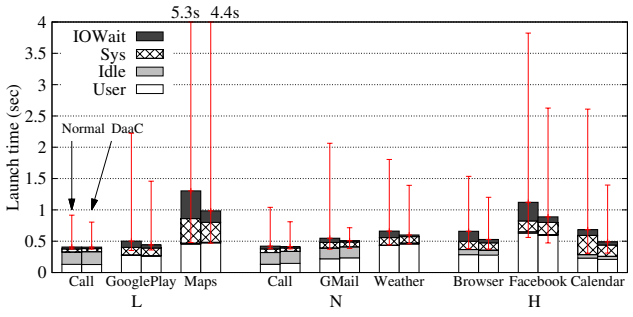


Figure 7: Breakdown of application launch time

5.3 Performance

Table 8 shows the basic performance of eCache and the Flash device in Nexus S. The performance of eCache is the average time of completing `get_page()` in the all workloads. Flash read performance is measured by reading 4KB of page directly. As eCache is an in-memory cache, eCache read performs 20 times faster than flash read.

Figure 6(a) shows the total size of read I/O requested 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 on-demand memory allocation using our scheme. As shown in the figure, our scheme significantly reduces read I/O by from 76% to 85% as compared to the normal case. Since the memory in eCache stores file caches, read I/O traffic is largely absorbed by 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 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.

Type	Latency (msec)
Flash Read	1.51
eCache Read	0.07

Table 8: Basic performance of eCache and flash device in Nexus S when reading one page

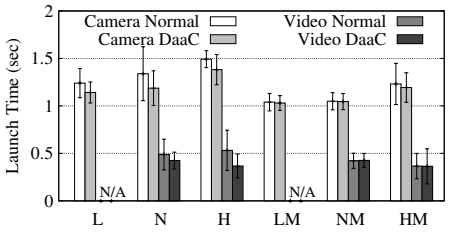
Figure 6(b) shows the geometric mean of the normalized launch times of all applications for each workload. As our scheme reduces many read I/O operations, the launch time is decreased by 8% to 16% as compared to the normal case. The main benefit of eCache during application launch comes from read I/O 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* times. The user and idle time denotes 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 eCache contributes to the reduction in read I/O and I/O-related kernel operations, the system and I/O-wait time are reduced from total launch time by 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 times of the applications. Except for the Call application, the applications showed reduced system and I/O-wait times by 22%-42% as compared to those of the normal case. As the portion of system and I/O wait times varies across the applications, the launch time of applications is reduced by 2% - 29%. The main reason for the negligible launch time reduction of the Call application is that the application launches most frequently during the workloads running. Accordingly, the file caches for the application likely resides in memory.

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 the other time than the application launch time. Accordingly, the other user interactive operations, such as menu changes, loading the other screen, will also benefit from the reduced I/O operations.

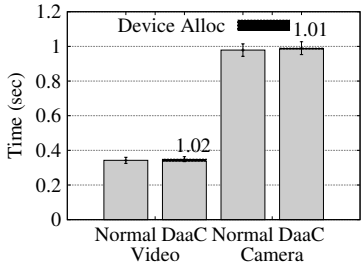
Meanwhile, the performance gain in application launch is not as much as those shown in Figure 3. We note that the figure shows the worst case performance of application launch when all page caches for an application is not in memory. Even in the normal case, frequently used files, such as Dalvik Cache files for Android framework and library files, can exist in memory. As some of those file data are in memory, the normal case cannot show the worst case launch performance.

Table 9 shows the types of files that are retrieved and hit in eCache during the normal workload running. Because other workloads have shown similar behaviors, we only show the result of the normal workload. Many commonly used files, such as framework-dependent files, system utilities and libraries, account for over a half of accessed and hit files in eCache. We believe that this is consistent with the results shown in the smartphone workload characterization study [16].

Since our scheme utilizes device-reserved memory for eCache,



(a) With the workloads



(b) Solo run

Figure 8: Launch time of device-dependent applications

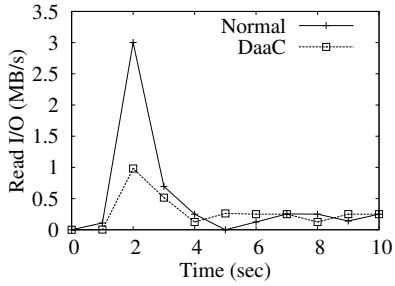


Figure 9: Read I/O occurred during Movies application

we have tradeoff between the gains from caching file and expenses of on-demand reservation time. In order to reveal this tradeoff, we first showed the activity time of Movies and Camera applications in each workload. The launch times of two applications include 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 static memory reservation. The reason is that the device memory allocation occurs in the middle of launching the application. Therefore, many file accesses can be hit in 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 triggered 2000 seconds before this Movies application launch; hence the files for the application are likely to be evicted from the page cache and even from 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 eCache. Table 10 shows the summary of actual read requests to 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 eCache.

Hit		Miss		Types
272	14%	37	12%	APK files (*.apk)
514	26%	88	28%	APK Dalvik caches (dalvik-cache/*@app@*)
37	2%	35	11%	Application-specific data (/data/*)
507	25%	19	6%	Framework Dalvik Caches (dalvik-cache/*@framework@*)
23	1%	1	0%	System utilities (/system/bin/*)
160	8%	21	7%	Frakework resources (/system/*)
455	23%	47	15%	Libraries (/system/lib/*)
12	1%	65	21%	Others

Table 9: The size of files hit and missed in eCache in the Normal workload (in MB)

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)

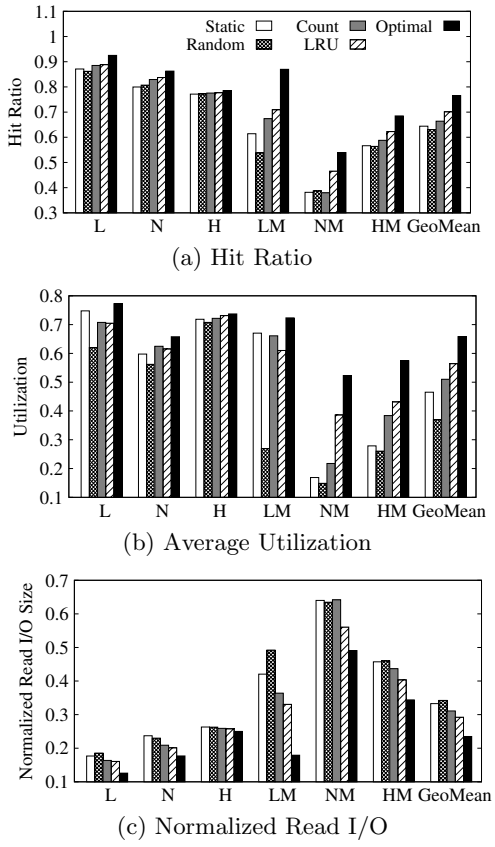
Table 10: The size of files hit in eCache when Movies application is launching (in MB)

In order to expose the cost of device memory allocation, 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 times of 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. The minimum cost to provide 70MB of memory using on-demand reservation is 0.3 seconds as shown in Figure 1(a). As compared to the cost, our scheme provides on-demand memory allocation 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 with other policies. Our policy is denoted as *LRU* that uses $C_i = l_i$ for page cost function. For comparison, we added four different policies. *Static* denotes static allocation policies. 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 possible regions for a device. *Count* assigns the same cost (i.e., 1) to pages which are in use in eCache for caching:

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



(a) Hit Ratio

(b) Average Utilization

(c) Normalized Read I/O

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

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

We evaluated each policy with trace-driven simulation in order to minimize runtime variation. A trace includes all accesses to 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 *hit ratio* of eCache, *average utilization* of eCache, and normalized *read I/O size*. The hit ratio represents how well the used policy provides device memory allocations without much losing the caching efficiency. The average utilization shows how many cached pages are in eCache. The average utilization is the average of the number of valid pages divided by the number of page frames in eCache at every second. The read I/O size for each workload is calculated based on the number of `get_page()` misses and represented in a normalized value to the normal case.

Across the workloads, multimedia intensive workloads (LM, NM, HM) have shown lower hit ratio because more frequent on-demand reservation requests for devices discard many cached files in 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) show higher hit ratio and perform less read I/O operations that static and random policies. Although the static and count policies show the higher eCache utilizations in workload L, N, and LM than cost-based policies in Figure 10(b), increasing the utilization of eCache is not related to the efficiency of eCache because the static and count policies show lower hit ratios 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 LRU-aware cost assignment is crucial for device memory allocation in eCache.

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 [31, 36] increases the reserved memory's utilization by allowing kernel to use the page frames in the reserved memory region similarly to Hotplug Memory [35, 17]. 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 while could sacrifice end user latency.

In contrast, Rental Memory Management [22], 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 the performance of our scheme will be improved against this approach for two reasons. First, the both approaches have the same size of file cache in system. In addition, based on the characteristics of applications depicted in Figure 5, 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 eCache rarely occurs. Second, our scheme always discards the least important pages in system while the REM approach does not. As a result, our scheme will show reduced read I/Os as compared to the Rental Memory Management approach.

Eviction-based Cache [6] is a storage cache management scheme. As data is evicted from upper-level cache (e.g., application server's page cache), the data is stored in the cache (e.g., storage server's cache). As the cache is managed exclusively [38] to upper-level cache, it increases the effective size of upper-level cache [6]. Chen et al. [5], also comprehensively studied storage cache management including the collaboration between the caches. We believe that many of the collaboration can improve the performance of eCache. The eviction-based cache is also used in virtual machine working tracing [25].

7. CONCLUSION AND FUTURE WORK

In resource-constrained embedded systems, 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 a 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 eCache, the system can greatly reduce 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 that with the static-reservation approach.

The limitation of our work is we assume that each device's reserved memory regions are fully used when on-demand reservation occurs. When the reserved region is partially used, we have another opportunity to increase memory efficiency by providing more sophisticated dynamic region selection during the on-demand reservation. We plan to extend our scheme to support those partially used reserved memory.

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

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

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

[7] K. Cho. iommu/exynos: Add IOMMU/system MMU driver for Samsung Exynos. `http://lwn.net/Articles/468015/`, November 2011.

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

[9] J. DeLuca. Compete ranking of top 50 web sites for february 2011 reveals familiar dip. `http://blog.compete.com/2011/03/29/compete-ranking-of-top-50-web-sites-for-february-2011-reveals-familiar-dip/`, March 2011.

[10] 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 *Proceedings of OSDI'10*, pages 1–6, 2010.

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

[12] 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.

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

[14] Google. monkeyrunner. `http://developer.android.com/guide/developing/tools/monkeyrunner_concepts.html`, March 2012.

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

[16] 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 *Proceedings of IISWC'11*, pages 81 –90, nov. 2011.

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

[18] Htcdev. Kernel source code. `http://htcdev.com/devcenter/downloads`, 2012.

[19] 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.

[20] Intel. Intel dynamic video memory technology (DVMT) 3.0. `http://download.intel.com/design/chipsets/applnots/30262305.pdf`, August 2005.

[21] J. Jeong, H. Kim, J. Hwang, J. Lee, and S. Maeng. Rigorous rental memory management for embedded systems. Technical Report CS-TR-2011-349, Korea Advanced Institute of Science and Technology, July 2011.

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

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

[24] R. Love. *Linux Kernel Development*. Addison Wesley, third edition, 2010.

[25] P. Lu and K. Shen. Virtual machine memory access tracing with hypervisor exclusive cache. In *Proceedings of USENIX ATC'07*, pages 3:1–3:15, 2007.

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

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

[28] 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.

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

[30] S. Moskovchenko. arm: msm: Add MSM IOMMU

support. http://lwn.net/Articles/399016/, August 2010.

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

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

[33] Z. Pfeffer. The virtual contiguous memory manager. In *Proceedings of OLS'10*, pages 225–230, 2010.

[34] Samsung. Open source release center. http://opensource.samsung.com, 2012.

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

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

[37] P. Willmann, S. Rixner, and A. L. Cox. Protection strategies for direct access to virtualized I/O devices. In *Proceedings of USENIX ATC '08*, pages 15–28, 2008.

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

[39] B.-A. Yassour, M. Ben-Yehuda, and O. Wasserman. On the DMA mapping problem in direct device assignment. In *Proceedings of SYSTOR '10*, pages 18:1–18:12, 2010.