Managing GPU Buffers for Caching More Apps in Mobile Systems

Sejun Kwon
Sungkyunkwan University
sejun000@csl.skku.edu

Sang-Hoon Kim
KAIST
sanghoon@calab.kaist.ac.kr

Jin-Soo Kim
Sungkyunkwan University
jinsookim@skku.edu

Jinkyu Jeong
Sungkyunkwan University
jinkyu@skku.edu

ABSTRACT

Modern mobile systems cache apps actively to quickly respond to a user's call to launch apps. Since the amount of usable memory is critical to the number of cacheable apps, it is important to maximize memory utilization. Meanwhile, modern mobile apps make use of graphics processing units (GPUs) to accelerate their graphic operations and to provide better user experience. In resource-constrained mobile systems, GPU cannot afford its private memory but shares the main memory with CPU. It leads to a considerable amount of main memory to be allocated for GPU buffers which are used for processing GPU operations. These GPU buffers are, however, not managed effectively so that inactive GPU buffers occupy a large fraction of the memory and decrease memory utilization.

This paper proposes a scheme to manage GPU buffers to increase the memory utilization in mobile systems. Our scheme identifies inactive GPU buffers by exploiting the state of an app from a user's perspective, and reduces their memory footprint by compressing them. Our sophisticated design approach prevents GPU-specific issues from causing an unpleasant overhead. Our evaluation on a running prototype with realistic workloads shows that the proposed scheme can secure up to 215.9 MB of extra memory from 1.5 GB of main memory and increase the average number of cached apps by up to 31.3%.

1. INTRODUCTION

Recently, mobile systems, such as smartphones and tablets, have been overtaking PCs in personal computing environments. Due to the adoption of open platforms, users can easily download and run applications (or apps for short) whichever they want to use from app stores [24]. In this regard, it is important in mobile systems to carefully manage underlying resources so that the demands for running many and various apps are satisfied on resource-constrained hardware.

Memory is one of the most important resources to be managed carefully in mobile systems to satisfy the demands within its limited capacity. Many mobile platforms (e.g., Android) provide app caching as a means to provide high responsiveness of app launching. Caching an app means to keep the app in memory in a paused state instead of killing it when a user leaves the app, since launching the app from spawning a new process takes a long time due to the initialization overheads in the app and the system. Accordingly, when a user wants to use the cached app again, the app can quickly respond to the user's call by resuming execution rather than being restarted from scratch. Therefore, it is crucial to cache as many apps as possible with a given limited amount of memory. Since caching an app costs the memory allocated to the app, many studies have been proposed to improve the memory utilization of the limited memory space thereby increasing the number of cached apps [3, 10, 14, 16].

Among these previous studies, compressing in-memory data is a compelling solution for increasing the memory utilization. Due to the app caching, a large portion of memory is actually inactive but allocated for keeping the state of cached apps. By compressing such inactive memory, the system can secure more free memory and utilize it for caching more apps, caching more file data, and so forth. In practice, many smartphone manufacturers have been employing similar approaches in their state-of-the-art smartphones [21, 22]. Android, one of the most popular mobile system platforms, also encourages adopting the memory compression in low memory conditions [3].

However, the approaches to compressing inactive memory data, which are usually referred to as a compression cache, miss an opportunity for securing more memory. The compression cache has been targeting virtual memory (VM) pages which contain stacks and heaps of apps. In modern mobile systems, many apps leverage a graphics processing unit (GPU) to accelerate graphic operations for a high-quality user interface and a realistic 3-D graphic scenery, which requires a large amount of GPU buffers along with the VM pages. These GPU buffers, however, are managed in a different way from the VM pages so that the conventional compression cache cannot track nor compress unused GPU buffers.

978-1-4673-8079-9/15/$31.00 ©2015 IEEE

This paper proposes a scheme to compress the unused GPU buffers to secure more free memory. Unlike VM pages, it is difficult to select inactive GPU buffers since the input/output memory management unit (IOMMU) page table with which GPU accesses a GPU buffer does not provide the functionality to track page access. Moreover, the traditional demand paging approach cannot be applied to GPU buffers since GPUs are usually incapable of recovering from a page fault [19]. For these reasons, our scheme exploits the state of an app from a user's perspective. If an app goes to the background, it implies that the GPU buffers belonging to the app become inactive since the background app is invisible to a user and does not incur graphic operations usually. Thus, the GPU buffer of the app can be compressed to reduce its memory footprint. If the app is turning into the foreground, it indicates that any compressed GPU buffers should be decompressed since the buffers are about to be active. In addition, we carefully manage the amount of compressed GPU buffer to control the unfairness caused by the significant variance in the GPU buffer size among apps.

The proposed scheme is implemented on a Google Nexus 5 smartphone running the Android platform, and is evaluated with various popular applications. Evaluation results show that the proposed scheme can provide up to 215.9 MB of extra memory, which increases the average number of cached apps by up to 31.3 % without imposing significant overheads associated with compression and decompression.

The rest of this paper is organized as follows. The following section describes the background and related work. Section 3 presents the motivation of this work. Section 4 describes the proposed scheme. Section 5 shows the experimental results of the proposed scheme using a running prototype. Finally, this paper concludes in Section 6 with providing the future work.

2. BACKGROUND AND MOTIVATION

2.1 Background

2.1.1 Process Management in Android

Android [2] is one of the most popular platforms for mobile devices. It is comprised of the Android framework and the customized Linux kernel. The Android framework provides apps with a runtime environment on top of the kernel and manages the apps and the environment according to user interactions. The kernel controls the access to hardware devices and manages system-wide resources such as CPU and memory.

Android manages apps according to the Android app life cycle [4] which is tailored to the user behavior on mobile devices. It is known that users interact with many apps momentarily [9, 11, 23], which makes it burdensome for an end-user to manage the life of an app explicitly. Instead, Android caches apps as long as the system has enough free memory, and terminates one or more cached apps automatically when the free memory is low. When a user leaves from an app, the app is paused and sent to the background. If a user launches the app again, the app is brought back to the foreground immediately by resuming the app, thereby providing a fast response time of an app launch. Meanwhile, the framework estimates the importance of each app. Many reports have revealed that app usage has temporal locality [9, 20, 23, 26]. In this sense, the framework maintains an LRU list of cached apps, named *a cached app list*, and estimates the importance of an app based on the LRU distance of the app in the list. The estimated importance is provided to an in-kernel memory reclamation module called *Low Memory Killer (LMK)*. When the free memory in a system is low, LMK terminates the least important app and reclaims the memory associated with the app.

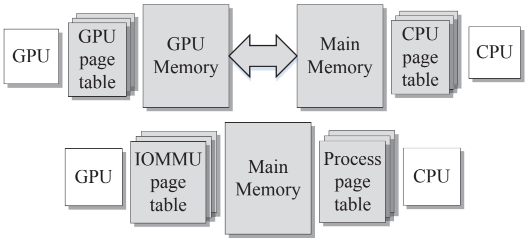


Figure 1: Shared memory architecture common in mobile systems

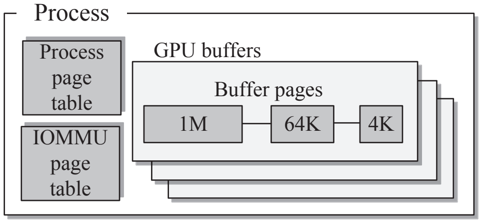


Figure 2: Components related to GPU buffers

An app might be cached or not according to the memory status and the recency of app use. An app is launched via *app start* if the app is not cached, or via *app resume* if the app is cached. The app resume is considered more efficient than the app start in terms of user experience and energy since the app start involves many software activities including spawning a process to host the app, loading data from a storage device, and initializing the app.

2.1.2 GPU Buffer

A GPU buffer is a memory object that is used for the input and output of GPU operations. GPU reads data from a GPU buffer, processes it, and outputs a result to a GPU buffer. Thus, GPU buffers usually contain GPU commands, textures for surfaces, and frame buffers.

There are two types of GPU memory architectures according to the location where the GPU buffers are allocated. One of the types is the private GPU memory architecture in which GPU buffers are allocated from private GPU memory, which is very fast and optimized for concurrent access from many GPU cores. This type of architecture is prevalent in traditional desktops and servers which demand very high GPU performance, and most of the GPU memory management studies have conducted on this architecture [15, 17, 18, 28].

In contrast, mobile systems usually employ a shared memory architecture in which GPU buffers are dynamically allocated from the main memory. Figure 1 depicts the shared memory architecture. GPU cannot afford its private memory in mobile devices since the devices are usually built from a system-on-a-chip (SoC) architecture in which resources are highly limited. GPU references a GPU buffer via an IOMMU page table, which is private to a process. Likewise the process page table, the IOMMU page table is set according to the current on-going GPU operation. Throughout the paper, we assume the shared GPU memory architecture.

A GPU buffer is managed by a graphic device driver. To create a GPU buffer, the device driver allocates *buffer pages* and maps the buffer pages to a GPU address space via the IOMMU page table. The mapping in the MMU page table is updated by the page fault handler on a per-page basis to amortize mapping overheads. Contrary to the MMU page table, however, once a GPU buffer is allocated, the mappings for the pages in the buffer are established together since GPU is not capable of providing demand paging [19]. Also, the buffer page is usually a superpage comprised of a number of physically contiguous 4 KB pages to reduce the IOMMU mapping overhead and to improve input/output translation lookaside buffer (IOTLB) efficiency [5, 32]. An allocated GPU buffer is indexed by its GPU-side address in its owner process.

2.2 Related Work

Due to the advantages of caching apps, many researchers have proposed various schemes for increasing memory utilization and caching more apps in mobile systems. Data compression is a compelling technique to increase the effective amount of memory by reducing the memory footprint of data. For decades, great effort has been devoted to the data compression technique from both the hardware and software points of views. We will focus on a software approach.

Wilson's early work [29] suggested to insert an additional level into the memory hierarchy that keeps data in a compressed form. This idea has been developed further to the *compression cache*, in which uncompressed memory acts like a cache of compressed data. Rarely referenced data is compressed to reduce its memory footprint whereas frequently accessed data is kept uncompressed so that the overheads incurred by compression and decompression is minimized [8, 27]. The work also attempted to adaptively balance the amount of memory between the compressed ones and the uncompressed ones. These approaches are, however, outdated for the modern memory management system that usually manages orders of magnitude larger memory than theirs.

Recently, ZRAM (previously known as compcache) [6, 10] has exploited the key idea of the compression cache. Instead of inserting a layer in the memory hierarchy, ZRAM offloads data compression and decompression to the block device layer for the sake of the modularity and simplicity. ZRAM introduces a general-purpose virtual block device that stores data into RAM in a compressed form, and utilizes the block device as a swap device. This approach is accepted to the Linux kernel mainline and is used for dealing with high memory demands in recent mobile systems [3, 21, 22]. Zswap [13] employs the similar approach to ZRAM. Unlike ZRAM, however, Zswap can utilize a physical block device as well as the main memory to store compressed in-memory data. Yang et al. [31] employed the similar idea for embedded devices. Kim et al. [16] applied memory deduplication to mobile systems and proposed several techniques to reduce CPU costs associated with finding identical pages. However, these studies only focus on the VM pages containing stacks, heaps, memory mapped files, and so forth, and do not take into account GPU buffers, which occupy a considerable amount of memory in mobile systems.

3. MOTIVATION

The trend in mobile devices that adopt a larger and higher-resolution display also drives the size of the GPU buffer to be larger. Since the GPU buffer is allocated from the main memory in the shared GPU memory architecture, the GPU buffer occupies a considerable fraction of the main memory. However, traditional memory management schemes have disregarded GPU buffers and do not reclaim GPU buffers even if they are inactive.

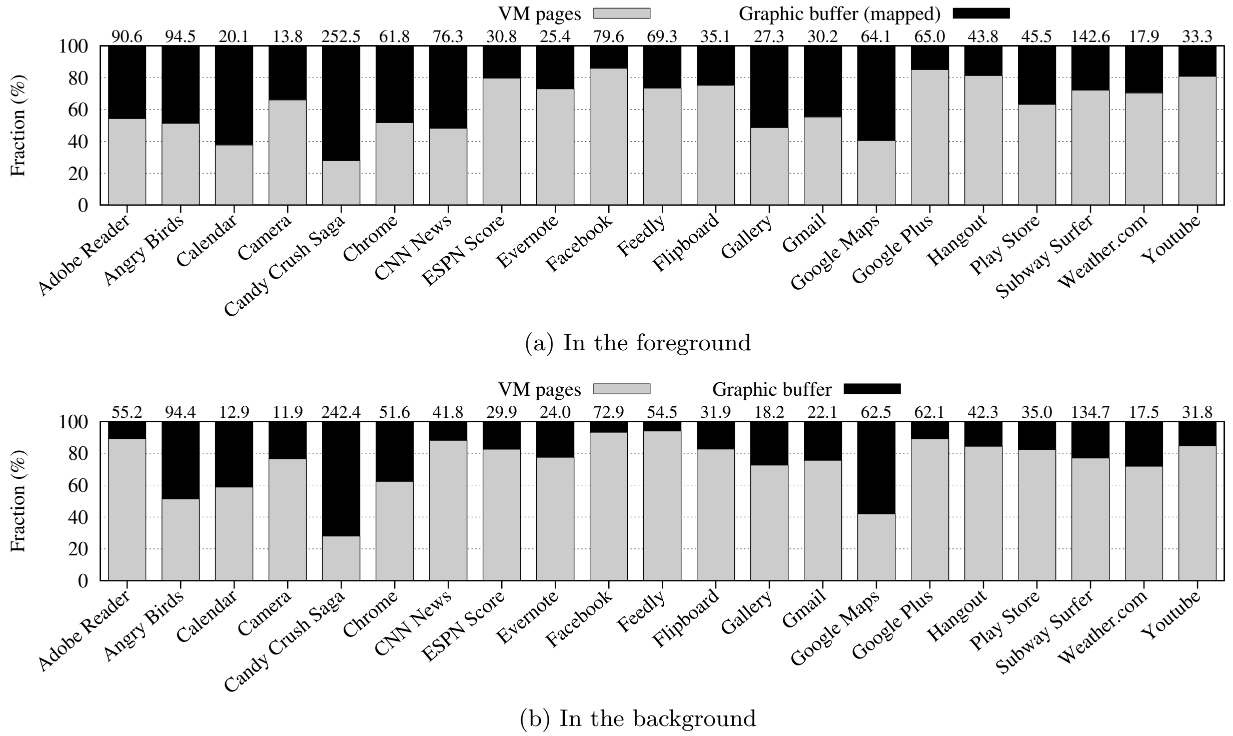
Figure 3 presents the fraction of GPU buffers over the memory footprint measured by the unique set size (USS) in popular Android apps when the apps are in the foreground and background. We modified the stock kernel to track the size of GPU buffers of each app. We presented USS instead of the resident set size (RSS) since USS of an app represents the caching cost of the corresponding app more accurately. The number on top of each bar indicates the memory footprint of the app in MBs.

When the apps are in the foreground, they use a large fraction of memory as their GPU buffers whose size ranges from 14.1% to 72.2% (36.4% on average) of the USS. This is due to that a foreground app actively utilizes GPU to accelerate graphic operations for drawing elements on a screen. In particular, game apps (Candy Crush Saga, Subway Surfer, and Angry Birds in this example) allocate GPU buffers extensively. When the apps go to the background, many apps voluntarily deallocate most of their GPU buffers. The remaining fraction of the GPU buffers is, however, not negligible for some apps in spite of the deallocation. Meanwhile, some apps do not release but keep most of their GPU buffers. This behavior makes sense in that the app might have to create the GPU buffers again upon next resume if the buffers are released thereby delaying the resume time of the app. For these reasons, 26.2% of the memory footprint of a background app is occupied by GPU buffers on average.

The GPU buffer belonging to a background app is not likely to be used in the near future since the background app is usually paused and does not request for a GPU operation. For this reason, the GPU buffers belonging to background (or cached) apps are one of the major culprits of decreasing the memory utilization. If the GPU buffers are reclaimed and utilized for caching more apps, we could greatly improve the memory utilization. However, traditional memory management schemes, such as paging and the compression cache [10, 13], do not consider nor manage the memory allocated for GPU buffers.

4. APPROACH

The observations described in the previous section motivated us to actively manage the GPU buffer of background apps at the system level. This section describes in detail how we manage GPU buffers and what we have considered.



(a) In the foreground

(b) In the background

Figure 3: A breakdown of memory footprint in popular Android apps

4.1 Design Considerations

Our key idea is to reclaim GPU buffers associated with the apps dormant in the background. To realize the idea, we first have considered swapping out GPU buffers to external storage. However, the amount of the GPU buffers of an app ranges up to hundreds of MBs, which implies significant I/O traffic to the storage device. This is not desirable in mobile systems as they are usually equipped with a flash memory-based storage device that wears out as data is written. Also, carving out a part of the storage device and dedicating it as a swap space is impractical for the resource-constrained mobile systems.

Instead, our approach is to compress GPU buffers rather than to swap them out. We choose this approach because we expect that GPU buffers will be highly compressible since the data in the buffers usually represents textures or GPU commands in which the regularity of data is inherent [30]. Our preliminary evaluation shows that the compression ratio (i.e., the ratio of the compressed size to the original size) of GPU buffers is 25.4% on average. This result indicates that our expectation is true in practice (see Section 5 for details).

There are three challenges regarding compressing a GPU buffer. First, the proposed scheme should be transparent to apps as well as the framework so that the scheme can be easily employed without modifying them. This is critical as it is impractical to control or enforce the large number of published apps. Second, we have to pick the proper GPU buffers to compress. Otherwise, any gains obtained from the compression can be offset by the overhead incurred by performing the compression and decompression. It goes without saying that the overhead should be minimized as well.

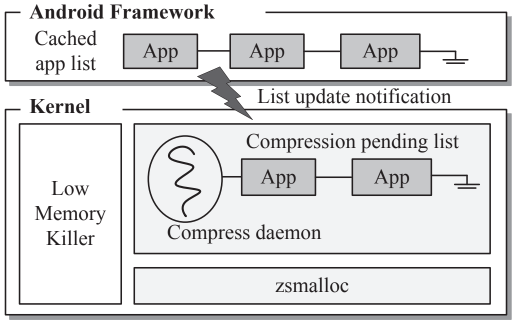


Figure 4: Overall system architecture

Finally, we must consider any influence to user experience since the end-users of mobile systems are very sensitive to the latency [25].

4.2 System Architecture Overview

Figure 4 presents the overall architecture of the proposed scheme. Recall that the Android framework notifies to the kernel whenever an update to the cached app list is occurred. Our scheme identifies an app to compress (i.e., dormant in the background) by hooking the notification. If such an app is found, its GPU buffers are compressed so that their memory footprint is reduced. Subsequent GPU operations from the cached app are postponed until the app is brought back to the foreground to prevent unnecessary decompression.

The compressed buffers are decompressed in the reverse order of the compression. When an app is brought back to the

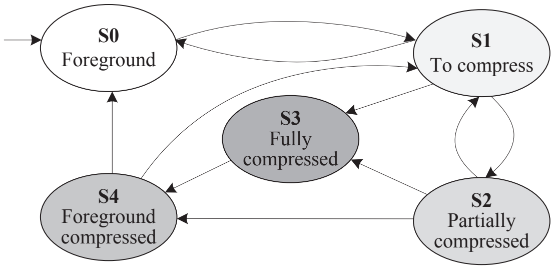


Figure 5: State transitions of an app

foreground, the system identifies the app by hooking the notification from the framework, and grants GPU operations to the app. If there is a pending operation or when the app requests for a GPU operation, the kernel restores the compressed GPU buffers of the app, and then dispatches the GPU operation to GPU so that GPU processes the operation.

An app in the system can be assigned to a state according to the status of its GPU buffer. Figure 5 outlines the states and transitions among them. When an app is launched and is in the foreground, the app is in the S0 state. When the app is sent back to the background, its state is changed to S1 and the app becomes a candidate for the GPU buffer compression. As GPU buffers of the app are compressed, it transitions to S2 or S3. In S1 to S3 states, GPU operations from the app is not scheduled but suspended. When the app is brought to the foreground, its state is changed to S4 in which an issued GPU operation is processed. If the app has any pending GPU operation or requests for a new GPU operation, its state is changed to S0 after decompressing all compressed GPU buffers.

4.3 GPU Buffer Compression

Our scheme relies on the notification from the framework to identify a candidate app for the compression, and the notification is invoked by the system server context which maintains the framework. Hence, performing the time-consuming compression in this context can impair the system integrity as well as the system performance by delaying other time-critical management tasks. Thus, we separate the buffer compression from the app notification. When a notification indicates that an app needs to be compressed, the notification returns immediately after inserting the app in the *compression pending list* shown in Figure 4. The list is sorted by the distance from the tail of the cached app list so that its head points to the app at the LRU position of the cached app list. When the system becomes idle and the compression pending list is not empty, the kernel wakes up a kernel thread called *compress daemon* that performs the GPU buffer compression. The compress daemon removes an app from the head of the compression pending list, and compresses GPU buffers associated with the app. The compress daemon iterates these routines until the pending list becomes empty. The compress daemon is set to run with the lowest priority (i.e., the nice value of 19) so that the CPU-intensive data compression does not affect user experience by competing for CPU with a foreground activity visible to a user.

To compress a GPU buffer, the daemon scans the buffer page list in the GPU buffer and compresses each buffer page in the list. Due to the IOMMU overhead and IOTLB efficiency, the buffer page is usually a superpage comprised of a number of physically contiguous 4 KB pages. The daemon compresses the buffer page in the 4 KB granularity and produces compressed data chunks. These chunks having various sub-page sizes are stored with the zsmalloc memory allocator [12], which is optimized for the data like the compressed chunks. When the daemon requests for memory space for a chunk, the zsmalloc allocator returns a handle for the specified size. The daemon copies the compressed chunk to the space pointed to by the handle and keeps the handle in the handle array in the GPU buffer descriptor. The buffer page is returned to the kernel when the entire buffer page is compressed and stored.

Before freeing the buffer page, any mapping to the buffer page should be updated accordingly. The buffer pages are mapped in the process page table at the CPU-side and the IOMMU page table at the GPU-side. The process page table entries corresponding to the buffer page are updated to be invalid so that a later reference to the page triggers the page fault handler that deals with the compressed page. For the IOMMU page table entry, we can skip the unmapping since a further access to the page will not occur because no GPU operation from the app is scheduled until the subsequent buffer decompression updates the mapping properly.

4.4 GPU Buffer Decompression

It is impossible to decompress compressed pages on-demand because GPUs for mobile systems are incapable of handling a memory fault in general [19]. When a page fault occurs, CPU can resume its execution after recovering from the fault via a page fault handler. In contrast, many GPUs in mobile devices consider the page fault unrecoverable and reset GPU cores upon the fault. This incapability leads to a design that all GPU buffers should be ready prior to issuing any GPU operation to GPU. In other words, the compressed buffers of an app should be completely decompressed when the app comes back to the foreground.

The restoration of compressed GPU buffers is done in the reverse order of the buffer compression. For each compressed buffer page in the GPU buffers of an app, the corresponding buffer page is allocated from the system memory allocator and is written by decompressing the compressed chunks pointed to by zsmalloc handles. After recovering the buffer page, the chunks pointed to by the handles are freed. The restoration is iterated on every compressed GPU buffers. We deliberately employ the per buffer page restoration policy to reduce the memory overhead while performing the decompression. A compressed chunk cannot be deallocated until it is restored to a buffer page. If the entire GPU buffer is allocated at once, the system has to hold the GPU buffer and its compressed chunks at the same time, which incurs high memory overhead. In our policy, in contrast, the allocation of the buffer page and the deallocation of the compressed chunk overlap so that the required maximum memory footprint is lowered substantially.

As a buffer page is restored in an app, the corresponding entry in the IOMMU page table is updated accordingly. Since

GPU always accesses GPU buffers via virtual addresses instead of physical addresses, relocating buffer pages does not cause any problems to GPU-side access. Contrarily to the IOMMU page table, the process page table is updated by the page fault handler later on a per-page basis to amortize the mapping overhead. When a page is referenced, a page fault occurs and the page fault handler is invoked. The page fault handler finds out the corresponding GPU buffer descriptor by looking up an index that maps the page address to the corresponding GPU buffer. If the corresponding GPU buffer is compressed, the handler restores the GPU buffer. Then, the corresponding page table entry is updated to the buffer page in the GPU buffer.

Decompressing GPU buffers upon a request for a GPU operation makes sense in that the GPU operation requires every GPU buffers to be ready. This policy, however, cannot filter out unnecessary buffer decompression initiated by an app in the background. Usually, an app in the background does not request for GPU operations since any graphical update is not visible to a user. However, we found that some apps generate GPU operations even though they are in the background and trigger the buffer decompression. However, the GPU operations can be deferred in most cases because the outcome of the GPU operations is less likely to be user-perceivable until the app comes to the foreground. For this reason, we can eliminate unnecessary buffer decompression by considering the status of apps (i.e., in the foreground or background). Thanks to the framework, the kernel is aware of the status of each app. We modified the original decompression policy to consider the status of an app as follows; if an app in the background requests for a GPU operation, the operation is not scheduled to GPU but postponed until the app comes to the foreground. When the app comes to the foreground, any pending operations are dispatched to the GPU operation queue, which will trigger the buffer decompression accordingly. In this way, we can prevent unnecessary buffer decompression. Actually, we verified that this scheme does not impair the correctness of apps nor incur any noticeable delay on an app's resume throughout the evaluation. However, there might be concerns about the apps that generate a number of GPU operations in the background. For example, an app can utilize GPU for general-purpose computations with OpenCL [1]. We believe that the GPU device driver and the framework can identify this kind of GPU access and let the GPU operations be scheduled to GPU after decompressing the GPU buffers of the app.

4.5 Taming Overheads

Unlike the decompression, we do not have to compress all GPU buffers at once nor to insist on a one-size-fits-all policy. In fact, compressing all GPU buffers yields the most extra memory, but it may result in an inefficiency and unfairness among apps. Assume a trending game app that is frequently used by a user. If its GPU buffers are compressed whenever the app goes to the background, the GPU buffers are frequently compressed and decompressed back and forth as the app comes and goes to the foreground and background. This is undesirable since the compression and decompression expend precious system resources. Also, if the app utilizes a large amount of GPU buffers, it will always experience a long delay to decompress all GPU buffers whenever it is resumed; it is unfair for the app. In this sense, we make each cached app have a cap and adjust the cap dynamically with respect to the likelihood of the app's reuse. Less amount of GPU buffers are compressed for the app that is more likely to be used soon. Many researchers on human behaviors on mobile devices indicate the recency is one of the simple yet effective metrics to predict the likelihood of an app's reuse [9, 20, 23, 26]. Thus, we employ a linear capping policy that sets the cap for the foreground app to 0 MB and increases the cap linearly as the app shifts to the LRU position of the cached app list. We choose the linear policy for the sake of simplicity. Evaluation with more sophisticated policies is left as the future work.

We limit the maximum value of the cap so that the delay incurred while decompressing buffers does not interfere with user experience much. According to the research on the human-computer interaction [25], a user will notice a delay when it exceeds 150 ms and begin to be annoyed when it exceeds one second. For this reason, 575 ms, the median of the thresholds, is a reasonable choice as the maximum allowance of the delay incurred by the decompression. This 575 ms is equivalent to around 100 MB of the uncompressed data in our evaluation system with the LZ4 compression algorithm. Thus, we pick 100 MB as a hard limit of the cap in our system. The proper value for the limit could be different on other systems, but the value can be obtained in the same approach.

To incorporate the policy, we modified the compress daemon to incrementally compress GPU buffers. As Android caches up to eight apps, the cap is increased by 12.5 MB for each app from the MRU position of the cached app list. When the compress daemon picks up an app from the compression pending list, it compares the current cap with the currently compressed amount. If the compressed amount is lower than the cap, the compress daemon compresses more GPU buffers until the compressed amount exceeds the cap or no uncompressed buffer remains.

5. EVALUATION

5.1 Methodology

We implemented a proof-of-concept prototype of the proposed scheme on a Google Nexus 5 smartphone running Android 4.4.3 (KitKat) and the Linux kernel 3.4.0. The smartphone is built with a Qualcomm MSM8974 Snapdragon 800 system-on-a-chip (SoC), which is equipped with a quad-core 2.26 GHz CPU, Adreno 330 GPU, and 2 GB of RAM.

We evaluated the scheme with micro-benchmarks and an in-house benchmark that launches apps in a given order. We ran the benchmark with a workload comprised of 256 app launches of 22 popular apps belonging to various categories according to the smartphone usage reported by the LiveLab study [23]. Table 1 lists the apps we have used and their composition in the workload.

As a baseline, we used the stock vanilla kernel which is referred to as Van. We evaluated two compression configurations—Full and Fair—both of which use the LZ4 compression algorithm [7]. Full refers to the compression policy that compresses all GPU buffers at once whereas Fair refers to the one that incrementally compresses with the cap described in

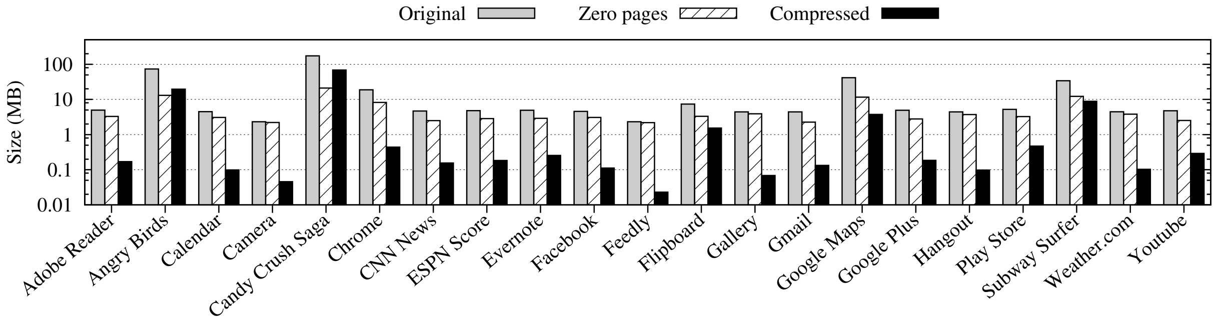


Figure 6: The amount of GPU buffers and their compressibility

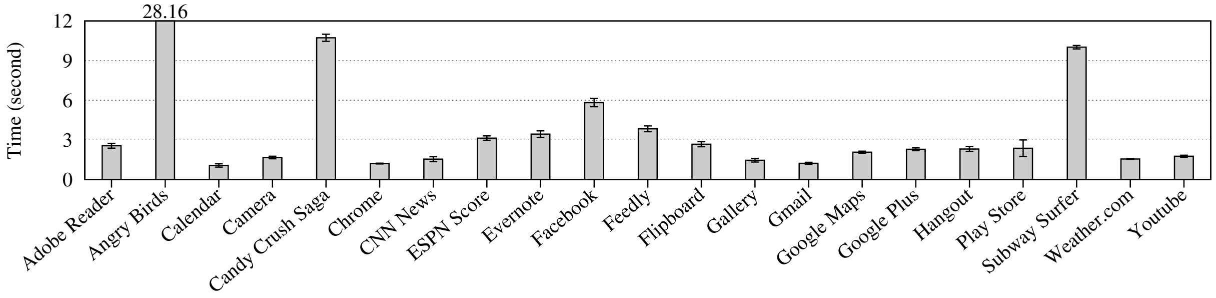


Figure 7: Average app start time of apps

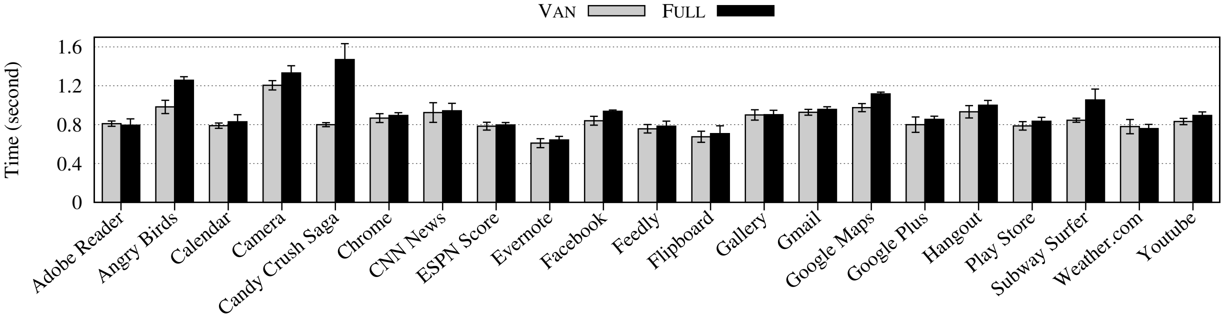


Figure 8: Comparison of the average app resume time

Section 4.5. Recall that FAIR compresses 12.5 MB at a time and up to 100 MB for an app.

Throughout the evaluation, we used the *performance* CPU frequency governor to eliminate an unmanageable performance variance. Also, the device is connected to the network over a 5 GHz WiFi channel that is not interfered with other access points nor devices nearby. Other parameters and configurations are set to default unless otherwise specified.

5.2 Micro-benchmark Analysis

First, we estimate the feasibility of the GPU buffer compression. We measured the size of the original GPU buffers and their compressed ones when an app is in the background. Figure 6 compares the size of the original ones and the compressed ones. Note that the y-axis is in log-scale. The size varies greatly across apps, ranging from 2.3 MB (Camera) to 173.5 MB (Candy Crush Saga). These GPU buffers are compressed to 0.02 MB to 68.5 MB, which occupies only 25.4% of the original size on average. A further analysis reveals that the high compressibility comes from a high fraction of zero pages in the buffers which are compressed very well. In fact, we tried to exploit these zero pages by handling them in a separate way. However, it turned out that LZ4 already deals with zero pages very efficiently in terms of time and space, returning scarce benefit in spite of the engineering effort. Thus, we decided not to handle zero pages separately.

The next evaluation is about the time to launch an app. To measure the time, we recorded the screen of the device on 30 frames-per-second (FPS) video while we launch an app via app start and resume. Then we analyzed the video frame-by-frame offline and obtained the time to launch the app. The given FPS of the video provides 33.3 ms of resolution in time. Figure 7 and 8 summarize the average and standard deviation of four launch times of each app. The results from FULL and FAIR are omitted in the launch time analysis (in Figure 7) as the app start is irrelevant to the buffer compression. Also, Figure 7 has a different scale on the y-axis compared to Figure 8 because app start times and resume times are quite different in their ranges.

In every app, the app start takes longer by 1.3× to 28.7×

Table 1: The list of apps in the evaluation workload and their fractions

Category	Apps	%
Productivity	Adobe Reader, Evernote, Calendar	6.6
News	ESPN Sports Center, CNN, The Weather Channel	4.3
Games	Angry Birds, Subway Surfer, Candy Crush Saga	18.4
Communication	Gmail, Hangout, Facebook, Google+, Flipboard, Feedly	38.3
References	Google Maps	6.6
Multimedia	Movie, Camera, Gallery, Youtube	11.7
Shopping	Play Store	4.7
Browser	Browser	9.4

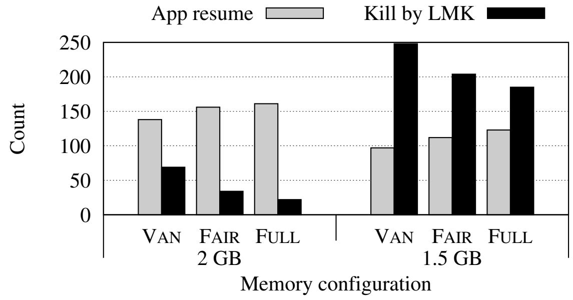


Figure 9: Count of app resumes among the entire workload

than its app resume. This suggests that caching an app can significantly improve user experience and an optimization for mobile systems has to consider this in mind. In most cases, the resume time of Full is increased by less than 100 ms compared to Van. However, the apps with large GPU buffers experience a noticeable time increase in their resume, which takes up to 0.67 second. This implies that the GPU buffer compression does not impair user experience in most cases, but can cause an undesirable latency increase for an app using a large amount of GPU buffers. We believe that this result justifies the necessity of the Fair policy.

5.3 Analysis with Realistic Workload

First of all, we report the number of app resumes and kills caused by the LMK. These are popular and practical metrics to measure the overall system performance of mobile systems. We conducted the evaluation on two memory size configurations to further understand the sensitivity of the scheme to the system memory size. We compared the original memory configuration (2 GB) with a reduced memory size configuration (1.5 GB) obtained by limiting the system memory with the kernel boot parameter. Figure 9 summarizes the results.

On both memory configurations, Full outperforms Van and Fair. Specifically, Full increases the app resumes by up to 26.8% while decreasing the number of app kills by the LMK by 68.1% compared to Van. It is due to that Full provides larger effective memory to the system by reducing the memory footprint of GPU buffers, thereby allowing the system to cache more apps. Fair positions between Van and Full as it also provides extra memory to the system but not as much as Full does. From these results, we can confirm that the proposed buffer compression scheme is practical and effective so that it can greatly improve overall user experience as well as system performance. To the rest of the paper, we will present the results obtained from the 1.5 GB memory configuration since they show the gains of our scheme more clearly yet the trends are same to the ones from the 2.0 GB configuration.

To understand the implication of the GPU buffer compression on user experience, we analyze the app resume time while running the workload. Unfortunately, we cannot perform the video analysis described in the micro-benchmark analysis since it is infeasible on the workload; a single workload run takes around 100 minutes, which implies we have to skip over the 100 minute-long video and analyze 256 app launches frame-by-frame. Instead, we use the launch time provided by the Android framework. Although it is not as accurate as the video analysis, we believe it can help to illustrate the implication of our scheme. Figure 10 summarizes the results. A box represents the average resume time, and the bar at the top of the box indicates its standard deviation. Note that the app without the box means the app was not resumed during the benchmark, and the one without the bar means the app was resumed once.

On most apps, Van and Full take the minimum and the maximum of the resume time, respectively, and Fair falls between them. In specific, the apps experience a longer resume time in Full than Van, which is increased by 122.8 ms on average and 1039.6 ms at the worst case. On the other hand, with the Fair policy, the resume time is only increased by 50.2 ms on average and 352.4 ms at the worst case. This result shows that the Fair policy effectively tames the latency caused by the decompression, thereby providing fairness among apps in terms of user experience. The apps that do not belong to this case (such as Camera, ESPN Score, Feedly, Flipboard and Play Store) can be explained with their confidences of samples; they have only one app resume or overlapped confidence ranges that prevents us from drawing a conclusion.

To quantify the memory saving obtained by the GPU buffer compression, we sampled the original buffer size and the current buffer size of apps at every second while running the workload. The current buffer size is obtained by aggregating the size of the uncompressed GPU buffers and the space allocated by the zsmalloc allocator. We obtained the memory saving by subtracting the current buffer size from the original buffer size. Figure 11 shows the cumulative ratio of the samples over the amount of memory saving. We can observe a substantial memory saving of up to 215.9MB can be obtained using Full. Fair saves less amount of memory than Full, up to 130.8 MB, but the saved amount is still considerable. The median of the memory saving is 63.7 MB and 112.2 MB for Fair and Full, respectively.

The framework can utilize the saved memory for caching more apps. To further understand the influences on the app caching, we collected the number of cached apps at every

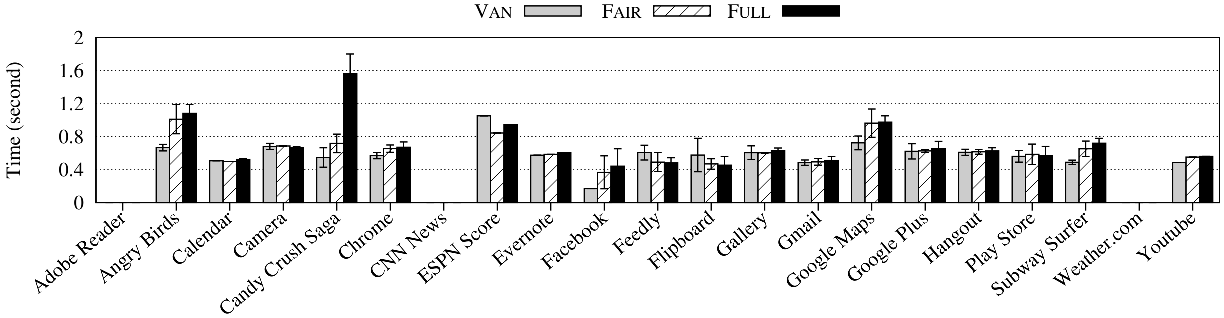


Figure 10: Average app resume time during the workload

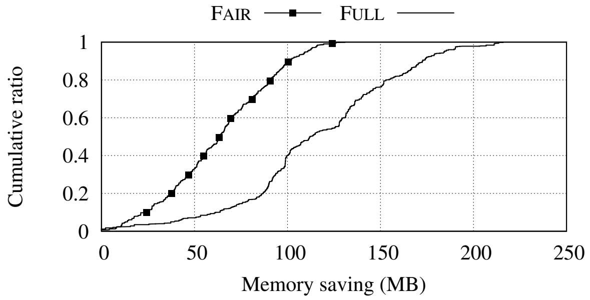


Figure 11: Memory saving obtained by the buffer compression

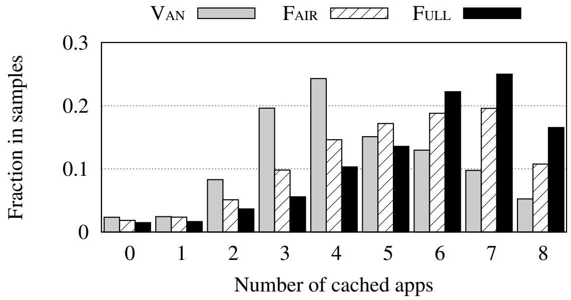


Figure 12: Average app resume time of apps

second while running the workload, and summarized the results in Figure 12. Recall that Android caches up to eight apps. Every configuration maintains a decent number of cached apps. However, VAN caches fewer apps than FAIR and FULL as it cannot utilize the extra memory saved by the buffer compression. In specific, the average number of cached apps is 4.38, 5.22, and 5.75 for VAN, FAIR and FULL, respectively. From the result, we can confirm that the GPU buffer compression allows the system to cache more apps.

Lastly, we analyze the CPU time of the configurations to provide the energy efficiency of the proposed scheme. Figure 13 shows a breakdown of CPU time measured while running the workload. Surprisingly, we found that the CPU utilization of FAIR and FULL is comparable to or 1.2% lower than that of VAN, which is counter-intuitive since data compression and decompression consume CPU cycles. Specifically, FULL spends less CPU time in user than VAN. We attribute the reduction of CPU time to the increased number of app resume. The app start spends more CPU time than the app resume. Even if the buffer compression might contribute some time to the system mode, the fewer app start can improve the overall CPU time. As a result, FAIR ends up using more time in both user and system modes than FULL even though it compresses less.

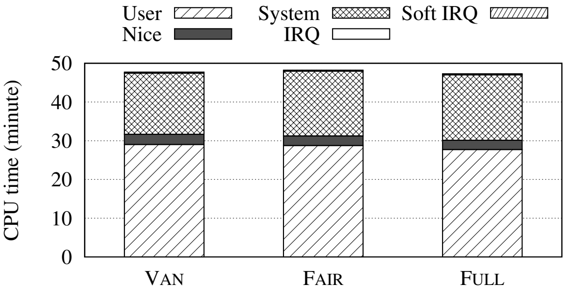


Figure 13: Breakdown of CPU time per modes

6. CONCLUSION AND FUTURE WORK

We found that GPU buffers are not managed well even though they occupy a considerable amount of memory in mobile systems. We suggested a novel scheme to reduce the memory footprint of the GPU buffers by compressing them. Also, we provided a technique to tame the overhead incurred by the decompression. The evaluation using a realistic workload confirms that the proposed scheme can provide substantial extra memory so that the system can cache more apps, thereby improving user experience and even energy consumption.

The GPU in mobile systems lacks of the capability to handle the page fault, making an on-demand decompression approach impossible. We hope that the insight provided by our work motivates GPU manufacturers to consider the on-demand memory fault handling capability in designing the GPU for mobile systems. We are working on to differentiate the compression policy according to characteristics of the types of GPU buffers (e.g., texture, command, and so forth). We leave customizing the memory allocator to store the compressed buffers more efficiently as the future work.

7. ACKNOWLEDGEMENTS

This work was supported partly by the National Research Foundation of Korea (NRF) grant funded by the Korea Government (MSIP) (No. 2013R1A2A1A01016441) and by the Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Education (NRF-2014R1A1A2054658).

References

[1] OpenCL. https://www.khronos.org/opencl.

[2] Android. http://www.android.com.

[3] Android. Low RAM. https://source.android.com/devices/tech/low-ram.html.

[4] Android. Managing the activity lifecycle. http://developer.android.com/training/basics/activity-lifecycle/index.html, October 2013.

[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. OLS*, pages 9–20, July 2007.

[6] A. F. Briglia, A. Bezerra, L. Moiseichuk, and N. Gupta. Evaluating effects of cache memory compression on embedded systems. In *Proc. OLS*, pages 53–64, June 2007.

[7] Y. Collect. LZ4: extremely fast compression algorithm. http://code.google.com/p/lz4, 2013.

[8] F. Douglis. The compression cache: Using on-line compression to extend physical memory. In *Proc. Winter USENIX Conference*, pages 519–529, 1993.

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

[10] N. Gupta. compcache: Compressed caching for Linux. http://code.google.com/p/compcache.

[11] D. Hackborn. Multitasking the Android way. http://android-developers.blogspot.kr/2010/04/multitasking-android-way.html, 2010.

[12] S. Jennings. Zsmalloc: memory allocator for compressed pages. http://lwn.net/Articles/474880/, Jan. 2012.

[13] S. Jennings. Zswap: Compressed swap caching. http://lwn.net/Articles/528817, 2013.

[14] J. Jeong, H. Kim, J. Hwang, J. Lee, and S. Maeng. Rigorous rental memory management for embedded systems. *ACM Trans. Embedded Computing Systems*, 12(1s), 2013.

[15] F. Ji, H. Lin, and X. Ma. RSVM: A region-based software virtual memory for GPU. In *Proc. PACT*, pages 269–278, 2013.

[16] S. Kim, J. Jeong, and J. Lee. Selective memory deduplication for cost efficiency in mobile smart devices. *IEEE Trans. Consumer Electronics*, 60(2):276–284, May 2014.

[17] Y. Kim, J. Lee, J.-E. Jo, and J. Kim. GPUdmm: A high-performance and memory-oblivious GPU architecture using dynamic memory management. In *Proc. HPCA*, pages 546–557, Feb. 2014.

[18] J. Lee, M. Samadi, and S. Mahlke. VAST: The illusion of a large memory space for GPUs. In *Proc. PACT'14*, pages 443–454, 2014.

[19] J. Menon, M. De Kruijf, and K. Sankaralingam. iGPU: exception support and speculative execution on GPUs. In *Proc. ISCA*, pages 72–83, 2012.

[20] A. Rahmati, C. Shepard, C. Tossell, M. Dong, Z. Wang, L. Zhong, and P. Kortum. Tales of 34 iPhone users: How they change and why they are different. *Technical Report TR-2011-0624*, 2011.

[21] Samsung Electronics, Co. Galaxy S3. http://www.samsung.com/global/galaxys3/.

[22] Samsung Electronics, Co. Galaxy S4 - life companion. http://www.samsung.com/global/microsite/galaxys4/.

[23] C. Shepard, A. Rahmati, C. Tossell, L. Zhong, and P. Kortum. LiveLab: measuring wireless networks and smartphone users in the field. *ACM SIGMETRICS Performance Evaluation Review*, 38(3):15–20, January 2011.

[24] The Nielsen Company. Smartphones: So many apps, so much time. http://www.nielsen.com/us/en/insights/news/2014/smartphones-so-many-apps--so-much-time.html, July 2014.

[25] N. Tolia, D. G. Andersen, and M. Satyanarayanan. Quantifying interactive user experience on thin clients. *Computer*, 39(3):46 – 52, Mar. 2006.

[26] I. Trestian, S. Ranjan, A. Kuzmanovic, and A. Nucci. Measuring serendipity: Connecting people, locations, and interests in a mobile 3G network. In *Proc. IMC*, pages 267–279, 2009.

[27] I. C. Tuduce and T. Gross. Adaptive main memory compression. In *Proc. USENIX ATC*, pages 237–250, 2005.

[28] K. Wang, X. Ding, R. Lee, S. Kato, and X. Zhang. GDM: Device memory management for GPGPU computing. In *Proc. SIGMETRICS*, pages 533–545, 2014.

[29] P. R. Wilson. Operating system support for small objects. In *Proc. ECOOP-OOOSWS*, pages 80–86, Oct. 1991.

[30] P. R. Wilson, S. F. Kaplan, and Y. Smaragdakis. The case for compressed caching in virtual memory systems. In *Proc. USENIX ATC*, June 1999.

[31] L. Yang, R. P. Dick, H. Lekatsas, and S. Chakradhar. Online memory compression for embedded systems. *ACM Trans. Embedded Computing Systems*, 9(3):27:1–27:30, Mar. 2010.

[32] B.-A. Yassour, M. Ben-Yehuda, and O. Wasserman. On the DMA mapping problem in direct device assignment. In *Proc. SYSTOR*, 2010.