Cache scheme of shared-buffer mappings for energy-efficiency of mobile devices

Jinkyu Jeong, Joonwon Lee and Euiseong Seo✉

The repetitive mapping and unmapping operations of shared buffers, which are used for data sharing among applications and devices, incur significant processing overhead. Thus, a significant amount of energy is consumed during video playback, especially when physical address space becomes heavily fragmented after a large number of application instances. A caching scheme that preserves mapping information of the shared buffer for later use of the same buffer mapping is proposed. The evaluation shows that the proposed scheme reduces processor utilisation by ~64% and energy consumption by ~17% during video playback compared with that of commercial smartphone systems.

Introduction: Video playback is an essential feature of mobile devices, such as smartphones and tablets. It is frequently regarded as a standard for measuring the energy efficiency and battery lifetime of a gadget because users typically play videos for extended periods of time.

Video play requires collaboration of the central processing unit (CPU), multimedia decoder and display device. The decoder is a hardware component that decodes compressed video data to original frames. Mobile devices are equipped with decoders to secure real-time decoding performance and energy efficiency. Video playback is a continual repetition of decoding of compressed video data and rendering of decoded frames. In each repetition, the application transfers the compressed video from the file system to the decoder; the decoder then sends the decoded frames to the display.

For data sharing among components, the buffer sharing scheme [1] has been widely used. In this scheme, when a buffer is allocated, a data-sharing entity, such as a process or device driver, can map the buffer to its virtual (or input/output (I/O)) address space to share the buffer. The actual data transfer between the buffer and device is accomplished through direct memory access (DMA). The IO-memory management unit (IO-MMU), which translates a virtual address of a peripheral device to the corresponding physical address, is pervasive in modern mobile systems. Therefore, the DMA commands are built not with physical but with virtual addresses of the buffer. As a result, a data transfer to or from the buffer requires a single DMA command even when the buffer is scattered over multiple discontiguous physical pages.

Conventional video players allocate a video buffer to deliver the compressed video data to the decoder, as shown in Fig. 1. This video buffer is first mapped to the address space of the video player. The video player unmaps the buffer after it fills the buffer with the compressed video data. In turn, the video buffer is mapped to the address space of the decoder. While the decoder reads the video buffer, the video player allocates and maps another video buffer for subsequent data. The second video buffer is read by the decoder after the decoder completes reading the first video buffer. This double buffering improves the processor and decoder utilisation by pipelining. Double buffering is also applied to the frame buffers, which are used to deliver the decoded frames to the display.

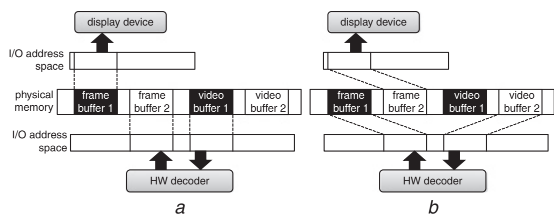

Fig. 1 *Conventional buffer management for video playback*
a Displaying 1st frame
b Displaying 2nd frame

When the device drivers receive a read/write request to a shared buffer, they map the buffer to the address spaces of the corresponding devices, process the request and finally unmap the buffer, although the buffer may need to be mapped again for forthcoming requests. This single-use mapping convention is based on the fact that the underlying device drivers, which actually map the shared buffer to the device address spaces, do not know whether they will use the buffer again.

Contiguous physical pages allocated for a buffer can be grouped into and managed as a super-page, whose size ranges from a few hundred kilobytes to a few megabytes. A super-page is the unit of mapping and unmapping operations. Therefore, to improve the performance of DMA operations and to reduce the overhead for mapping and unmapping operations [2], the kernel attempts to allocate as few pages as possible for the buffer by preferring super-pages. However, if the physical address space is highly fragmented, the kernel has no option but to allocate for the buffer a set of normal pages rather than super-pages.

Using a mobile system for a long period of time makes its physical memory highly fragmented due to excessive allocation and deallocation of 4 kB normal pages for repetitive application instantiations [3]. The experimental results illustrated in Fig. 2 show that the physical address space of the system described in Table 1 was heavily fragmented after 1000 times of application instantiations following the LiveLab usage model [4]. Therefore, a large portion of free memory, which is marked as the unusable index in Fig. 2, was unavailable to be allocated as 16 kB or 1 MB super-pages because of the fragmentation.

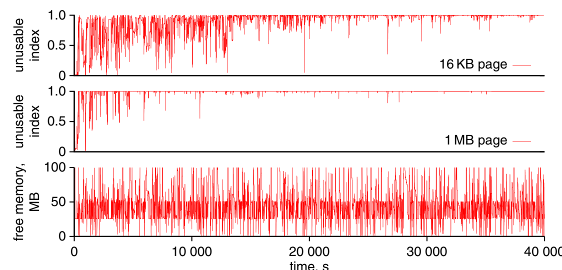

Fig. 2 *Ratio of unusable memory to free memory for super-page requests*

Table 1: System configuration for evaluation

Hardware	System on Chip	Samsung Exynos 5250
	CPU	1.7 GHz dual Cortex-A15
	random access memory	2 GB LPDDR3
Software	operating system	Android 4.2.2
	kernel	Linux 3.4.5
	application	Android media player
Video file	Codec	H.264
	res./frames per second	1920 × 1080/30 frames per second
	frame/video buffer	46 MB/17 MB

The physical memory fragmentation severely affected the performance of the video playback. The *iommu_map()* function, which handles mapping/unmapping of shared buffers, contributed only 0.18% of CPU utilisation during video playback of a 1080i MP4 file when the physical memory was not fragmented. However, the CPU utilisation of the function increased to 13.5%, which was approximately a half of the total CPU utilisation during the video playback. A similar mapping overhead increase occurred with a camera application, which also uses shared buffers to deliver frame data captured from the camera to the application and encoder. The CPU consumption of the mapping function increased from 0.17 to 8.61% under severe memory fragmentation. This high CPU utilisation not only affects the performance, but also significantly reduces the battery life.

In this Letter, we propose a buffer-mapping cache scheme that reduces the processing overhead incurred by mapping and unmapping of shared buffers under memory fragmentation. We evaluate the proposed scheme in terms of processor utilisation and power consumption.

Mapping cache: Fig. 3 depicts an overview of the proposed mapping cache scheme. The mapping cache, preserves the mapping table for the buffer. This occurs even after a sharing entity unmaps the buffer from its address space so that the mapping table will be used again when the entity requests the mapping of the same buffer. Since the reconstruction of the mapping table is not necessary when using the cached mapping data, the mapping overhead becomes marginal, even under a heavy memory fragmentation condition.

When the kernel generates a buffer, it constructs a gather/scatter table, which is the list of physical pages allocated for the buffer. The gather/scatter table is referred to when a sharing entity maps the buffer. The gather/scatter table for a buffer is destroyed when the corresponding buffer is removed. The mapping cache uses the status of the gather/

scatter table for a buffer to determine whether to keep or discard mapping data for the buffer.

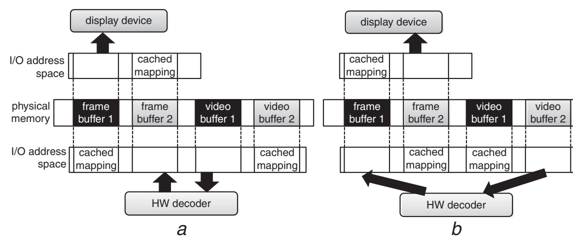

Fig. 3 *Use of buffer-mapping cache for video playback*

a Displaying 1st frame
b Displaying 2nd frame

When a sharing entity requests mapping of a buffer, the kernel constructs a mapping table from the virtual to physical addresses using the gather/scatter table. This table is searched to retrieve the physical addresses of the DMA targets by the IO-MMU. Each sharing entity, including devices and processes, has its own virtual address space. Therefore, the mapping cache contains a separate mapping table for each sharing entity that has requested mapping of the buffer at least once. The size of the mapping table is linear to the number of normal or super-pages; each entry is 4B.

As mentioned, the proposed cache scheme preserves the mapping table instead of removing it when the sharing entity unmaps the buffer and the gather/scatter table for the buffer remains. An entry in the mapping cache is actually a metadata structure for each preserved mapping table. The metadata structure contains information about the address of the corresponding gather/scatter table and the owner entity of the mapping table. When a gather/scatter table is released from physical memory, all entries corresponding to the gather/scatter table are together discarded from the mapping cache.

When the kernel receives a buffer-mapping request, it first searches for a matching entry in the cache. The kernel simply reuses the matching mapping table if it exists in the cache. When a change occurs in the gather/scatter table due to a resizing operation of the buffer, the mapping cache records the change in the relevant cache entries. In turn, the kernel adjusts the cached mapping tables accordingly when they are retrieved for reuse. This approach significantly reduces the overhead for mapping of a resized buffer because only the expanded or contracted mapping entries must be manipulated rather than a total reconstruction.

Yassour *et al.* [5] proposed an approach that prefetches or caches the mappings of the DMA target pages in the virtualised environment with IO-MMU to improve the I/O virtualisation performance. Unlike that approach, our scheme makes caching decisions at the granularity of shared buffers simply based on their status because the goal is to reduce the overhead for mapping of shared buffers.

Performance evaluation: We implemented the mapping cache scheme in the system described in Table 1 for evaluation. We measured the processor utilisation and power consumption during video playback with and without the mapping cache under both heavy fragmentation and no-fragmentation conditions. The fragmentation condition was simulated by prohibiting the use of super-pages for the buffer allocation. The processor utilisation was obtained from */proc/stat* every second.

Fig. 4 shows the time series of the processor utilisation under different setups. The utilisation remained calm most of the time; however, there were occasional fluctuations due to the event handling for system management tasks that were irrelevant to the video playback. The video playback began at the 20 second point. The average utilisation was 10.2% under no fragmentation and 26.5% under heavy fragmentation. This 16% difference was predominantly caused by the mapping overhead, as stated. However, the gap was reduced to only 1.2% with the proposed scheme, which was between 10 and 11.18%.

The time series of the power consumption during video playback, which was obtained from the battery sensor, is illustrated in Fig. 5. The device consumed 3254 μW on average during video playback without both memory fragmentation and the mapping cache. It consumed 3912 μW, which was a 20% greater value, under heavy memory fragmentation. The mapping cache suppressed the average power consumption to 3223 and 3297 μW under no-fragmentation and heavy fragmentation conditions, respectively. The difference between the cases without and with the mapping cache scheme was 16% under heavy fragmentation condition.

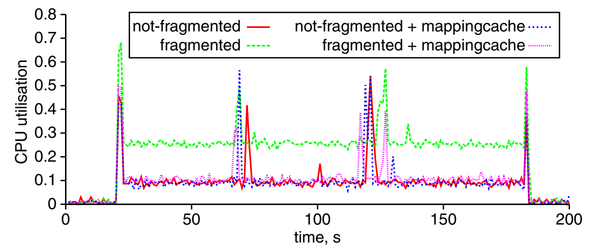

Fig. 4 *Processor utilisation during video playback under different conditions*

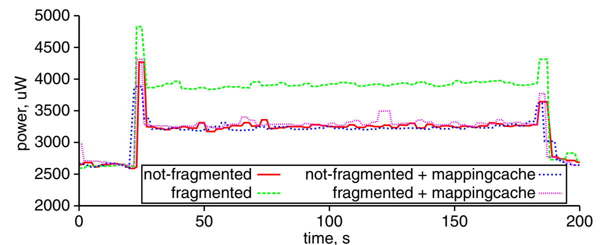

Fig. 5 *Power consumption during video playback under different conditions*

Conclusion: The physical memory of mobile consumer electronics, which are typically run for long periods of time, become heavily fragmented after numerous repetitions of application instantiations and executions. In this Letter, we have analysed the shared-buffer-mapping overhead under memory fragmentation, and proposed a buffer-mapping cache scheme to remedy the extremely high mapping overhead. Our evaluation shows that the proposed scheme saves up to ∼64% of processor utilisation and reduces up to ∼17% of energy consumption during video playback. Considering that video playback is an essential and typically long-running feature of mobile devices, we believe that the proposed scheme not only improves multitasking performance, but also significantly extends battery life for video playback.

Acknowledgment: This research was supported partly by the Industrial Convergence Source Technology Development Program through the Ministry of Science, ICT and Future Planning, Korea (10044313) and partly by the Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Education (NRF-2014R1A1A2054658).

© The Institution of Engineering and Technology 2015
21 January 2015
doi: 10.1049/el.2015.0244
One or more of the Figures in this Letter are available in colour online.

Jinkyu Jeong, Joonwon Lee and Euiseong Seo (*Sungkyunkwan University, Suwon, Republic of Korea*)

✉ E-mail: euiseong@skku.edu

References

1 Corbet, J.: 'Sharing buffers between devices', LWN.net, 2011, Article No. 454389. Available at http://www.lwn.net/Articles/454389
2 Amit, N., Ben-Yehuda, M., and Yassour, B.: 'IOMMU: strategies for mitigating the IOTLB bottleneck', *Lect. Notes Comput. Sci.*, 2012, **6161**, pp. 256–274
3 Gorman, M., and Whitcroft, A.: 'The what, the why and the where to of anti-fragmentation'. Ottawa Linux Symp., Ottawa, Canada, July 2006, pp. 369–384
4 Shepard, C., Rahmati, A., Tossell, C., Zhong, L., and Kortum, P.: 'LiveLab: measuring wireless networks and smartphone users in the field', *ACM SIGMETRICS Perform. Eval. Rev.*, 2010, **38**, (3), pp. 15–20
5 Yassour, B., Ben-Yehuda, M., and Wasserman, O.: 'On the DMA mapping problem in direct device assignment'. Proc. SYSTOR' 10, Haifa, Israel, May 2010 Article No. 18, 2010