GCMA: Guaranteed Contiguous Memory Allocator

SeongJae Park
Seoul National University
sjpark@dcslab.snu.ac.kr

Minchan Kim
LG Electronics
minchan.kim@lge.com

Heon Y. Yeom
Seoul National University
yeom@snu.ac.kr

ABSTRACT

While demand for physically contiguous memory allocation is still alive, especially in embedded system, existing solutions are insufficient. The most adapted solution is reservation technique. Though it serves allocation well, it could severely degrade memory utilization. There are hardware solutions like Scatter/Gather DMA and IOMMU. However, cost of these additional hardware is too excessive for low-end devices. CMA is a software solution of Linux that aims to solve not only allocation but also memory utilization problem. However, in real environment, CMA could incur unpredictably slow latency and could often fail to allocate contiguous memory due to its complex design.

We introduce a new solution for the above problem, GCMA (Guaranteed Contiguous Memory Allocator). It guarantees not only memory space efficiency but also fast latency and success by using reservation technique and letting only immediately discardable to use the area efficiently. Our evaluation on Raspberry Pi 2 shows 15 to 130 times faster and more predictable allocation latency without system performance degradation compared to CMA.

1. INTRODUCTION

Because resource requirement of processes is not predictable, keeping high resource availability with high utilization has always been one of the hardest challenges. Memory management has not been exception either. Especially, large memory allocation was pretty much impossible because of the fragmentation problem [7]. The problem seemed to be completely solved with the introduction of virtual memory [3]. In real world, however, the demand for physically contiguous memory still exists [5, 6, 13]. Typical examples are various embedded devices such as video codec or camera which requires large buffers.

A traditional solution for the problem was memory reservation technique. The technique reserves sufficient amount of contiguous memory during boot time and use the area only for contiguous memory allocation. This technique guarantees fast and successful allocation but could degrade memory space efficiency if contiguous memory allocation does not use whole reserved area efficiently. There are alternative solutions using additional hardware like Scatter/Gather DMA or IOMMU. They solve the problem by helping devices to access discontiguous memory as if it was contiguous, like virtual memory system does to processes. However, additional hardware can be too expensive to low-end devices.

EWiLi'15, October 8th, 2015, Amsterdam, The Netherlands.

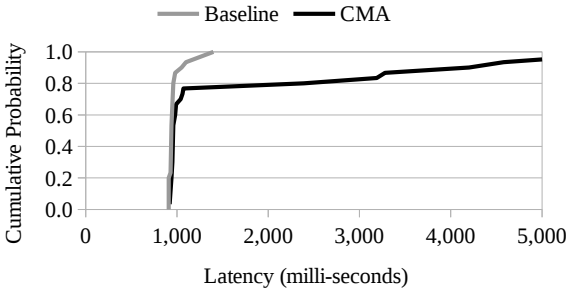


Figure 1: CDF of a photo shot on Raspberry pi 2 under background task.

Linux kernel has a subsystem for the problem, namely, CMA (Contiguous Memory Allocator) [13]. It basically follows the reservation technique and boosts memory efficiency by allowing the reserved area to be used by other *movable pages*. At the same time, it keeps contiguous memory allocation be available by migrating movable pages out of the reserved area if those movable pages are required for contiguous memory allocation. However, CMA is not very popular because memory migration usually takes a long time and could even fail from time to time. Figure 1 shows CDF of latency for a photo shot on Raspberry Pi 2 [19] under memory intensive background task, which is a realistic workload. Baseline means Raspberry Pi 2 using the reservation technique and CMA means modified to use CMA as alternative. Maximum latency for a photo shot using CMA is about 9.8 seconds while it is 1.6 seconds with reservation technique. Normally, 9.8 seconds for a photo shot is not acceptable even in the worst case. As a result, most system uses the reservation technique or additional hardware as a last resort despite expensive cost.

To this end, we propose a new contiguous memory allocator subsystem, GCMA (Guaranteed Contiguous Memory Allocator) [14], that guarantees not only memory space efficiency but also fast response time and successful allocation by carefully selecting additional clients for the reserved area as appropriate ones only. In this paper, we introduce the idea, design and evaluation results of GCMA implementation. GCMA on Raspberry Pi 2 shows 15 to 130 times faster and much predictable response time of contiguous pages allocation without system performance degradation compared to CMA, and zero overhead on latency of a photo shot under background task compared to the reservation technique.

2. BACKGROUND

2.1 Virtual Memory Management

Virtual memory [3] system gives a process an illusion that it owns entire contiguous memory space by letting the process use logical address which can be later mapped into discontiguous physical memory space. Because processes access memory using only logical address, system can easily allocate large contiguous memory to processes without fragmentation problem. However, giving every process the illusion cannot be done easily in reality because physical memory in system is usually smaller than the total sum of each process's illusion memory space. To deal with this problem, kernel conceptually expands memory using another large storage like hard disk drive or SSD. However, because large storages are usually much slower than memory, only pages that will not be accessed soon should be in the storage. When memory is full and a process requires one more page, kernel moves content of a page that is not expected to be accessed soon into the storage and gives the page to the process.

In detail, pages of a process can be classified into 2 types, file-backed page and anonymous page. File-backed page have content of a file that are cached in memory via *page cache* [15] for fast I/O. The page can be freed after synchronizing their content with the backing file. If the page is already synchronized with, it can be freed immediately. Anonymous page is a page that has no backing file. A page allocated from heap using `malloc()` can be an example. Because anonymous page has no backing file, it cannot be freed before its content is stored safely elsewhere. For this reason, system provides backing storage for anonymous pages, namely, `swap device`.

2.2 Contiguous Memory Allocation

Because `MMU`, a hardware unit that translates logical memory address into physical location of the memory, is sitting behind CPU, devices that communicate with system memory directly without CPU have to deal with physical memory address rather than taking advantage of the virtual memory. If the device, for example, requires large memory for an image processing buffer, the buffer should be contiguous in physical address space. However, allocating physically contiguous memory is hard and cannot always be guaranteed due to fragmentation problem.

Some devices support hardware facilities that are helpful for the problem, such as Scatter/Gather DMA or IOMMU [13]. Scatter/Gather DMA can gather buffer from scattered memory regions. IOMMU provides contiguous memory address space illusion to devices similar as MMU. However, neither Scatter/Gather CMA nor IOMMU is free. Supporting them requires additional price and energy, which can be critical in low-end device market unlike high-end market. Moreover, it seems that the trend would not disappear in the near future due to advance of IoT paradigm and emerging low-end smartphone markets. Low-end devices market is not the only field that could utilize efficient contiguous memory allocation. One good example is huge page [1] management. Because huge page could severely reduce TLB overflow, efficient huge page management is important for system with memory intensive workloads like high performance computing area. In this case, neither Scatter/Gather DMA nor IOMMU can be a solution because they do not guarantee real physically contiguous memory. This paper focuses on the low-end device problem, though.

Linux kernel already provides a software solution called CMA [13]. CMA, which is based on reservation technique, applies the basic concept of virtual memory that pages in virtual memory can move to any other place in physical memory. It reserves memory for contiguous memory allocation during boot time as reservation technique does. In the meantime, CMA lets movable pages to reside in the reserved area to keep memory utilization. If contiguous memory allocation is issued, appropriate movable pages are migrated out from the reserved area and the allocation is done using the freed area. However, unlike our hope, page migration is a pretty complex process. First of all, it should do content copying and re-arranging of mapping between logical address and physical address. Second, there could be a thread holding the movable page in kernel context. Because the thread is already holding the page, it cannot be migrated until the holding thread releases it. In consequence, CMA guarantees neither fast latency nor success. Another solution for fast allocation latency based on CMA idea was proposed by Jeong *et al.*[5, 6]. It achieves the goal by restricting use of reserved area as evicted clean file cache rather than movable pages. Because anonymous pages cannot be in the reserved area, the memory utilization could suffer under non file intensive workload despite its sufficiently fast latency.

3. GCMA: GUARANTEED CMA

3.1 Mistake on Design of CMA

In conceptual level, basic design of CMA is as below:

1. Reserve contiguous memory space and let contiguous memory allocation to be primary client of the area.
2. Share the reserved area with secondary clients;
3. Reclaim memory used by secondary clients whenever a primary client requests.

In detail, CMA chooses movable pages as its secondary client while using page migration as a reclamation method. Actually, basic design of CMA has no problem; the problem of CMA is that movable pages are actually not an appropriate candidate for secondary client. Because of many limitations of page migration described in Section 2.2, movable pages cannot easily give up the reserved area. That's why CMA suffers from slow latency and sometimes even fails. This problem can be avoided by selecting appropriate candidates instead of movable pages and managing them effectively.

3.2 Appropriate Secondary Clients

We have seen a bad candidate for secondary clients, movable pages. The problem is that movable pages cannot be freed for the primary client (contiguous memory allocation) immediately because of high cost and possible failures of page migration. Therefore, appropriate secondary clients should satisfy the following three requirements. First of all, it should be freed with an affordable cost. Second, it should not be accessed frequently because the area could be suddenly discarded for primary client. Finally, it should be out of kernel scope to avoid pinning of the page to other threads.

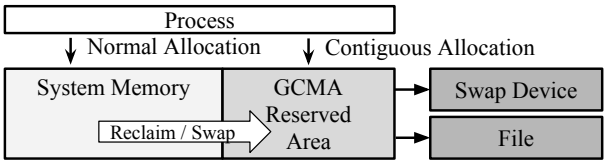


Figure 2: GCMA workflow.

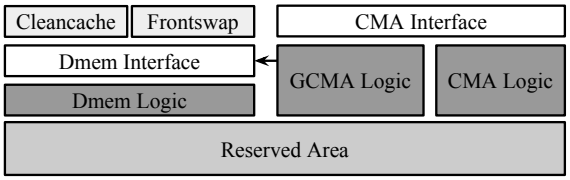


Figure 3: GCMA overall architecture.

For these purposes, final chance cache for clean pages that evicting from page cache and swapped out pages from system memory (Section 2.1) are good candidates. Evicting clean pages can be freed immediately without any cost. Swapping pages can be freed just after write-back. Those pages would not be accessed soon because they are chosen as reclaim target which kernel expected so. They are out of kernel scope because they are already evicted or swapped out. Additionally, system can keep high memory utilization because the final chance cache can utilize not only file-backed pages, but also anonymous pages. As consequence, GCMA can be designed as a contiguous memory allocator using final chance cache for evicting clean pages and swapping pages as its secondary clients. In other words, GCMA carries out two missions. A contiguous memory allocator and a temporal shelter for evicting clean pages and swapping pages is that.

3.3 Limitation and Optimization

Although secondary clients of GCMA are effective enough for contiguous memory allocation, it could have adverse effect on system performance compared to CMA. While movable pages, the secondary client of CMA, would be located in reserved area with only slight overhead, secondary clients of GCMA requires reclaim overhead before they are located in reserved area and it would consume processing power. Moreover, swapping pages would require write-back to swap device. It could consume I/O resource as well.

To avoid the limitation, GCMA utilizes the secondary client as write-through mode cache [8]. With write-through mode, content of pages being swapped out will be written to GCMA reserved area and backing swap device simultaneously. After that, GCMA can discard the page when necessary without incurring additional overhead because the content is already in swap device. Though using write-through mode could enhance GCMA latency, system performance could be degraded because it would consume much I/O resource. To minimize the effect, we suggest constructing the swap device with a compressed memory block device[4, 10], which could enhance write-through speed and swapping performance. We recommend Zram[10] as a swap device, which is an official recommendation from Google [17] for Android devices with low memory. After those optimization applied, GCMA proved it has no performance degradation at all from our evaluation owing to simple design and efficient implementation. Detailed evaluation setup and results are described in Section 5.

4. IMPLEMENTATION

To minimize migration effort of CMA users, GCMA shares CMA interface. Therefore, CMA client code can use GCMA alternatively by changing only one function call code from its initialization or turning on a kernel configuration option without any code change.

GCMA is implemented as a small independent Linux kernel module to keep code simple rather than increasing comprehensive hooks in kernel. Only a few lines of CMA code have been modified for interface integration and experimental features for evaluation. As evidence, GCMA implementation has been easily ported to various Linux versions, which have always been done in few minutes. In total, our implementation uses only about 1,300 lines of code for main implementation and only about 200 lines of code for changes to CMA. GCMA is published as a free / open source software under GPL license at `https://github.com/sjp38/linux.gcma/releases/tag/gcma/rfc/v2`.

4.1 Secondary Client Implementation

Final chance cache for evicting clean pages and swapping pages can make system performance improvements. The fact has been well known in the Linux kernel community and facilities for the chance, namely, Cleancache and Frontswap [2] have been proposed by the community. As those names imply, they are final chance cache for evicting clean pages and swapping pages. To encourage more flexible system configuration, community implemented only front-end and interface of them in Linux kernel and left back-end implementation to other modules. Once the back-end implementation has been registered, page cache and swap layer tries to put evicting clean pages and swapping pages inside that back-end implementation as soon as the victim is selected. If putting those pages succeeds, future reference of those pages will be read back from that back-end implementation which would be much faster than file or swap device. Otherwise, the victim would be pushed back to the file or swap device. Because the design fits perfectly with the secondary clients of GCMA, GCMA uses it by implementing those back-ends rather than inventing the interface again.

Back-end for Cleancache and Frontswap has similar requirements because they are just software caches. To remove redundancy, we implemented another abstraction for them, namely, *dmem* (discardable memory). It is a key-value storage that any entry can be suddenly evicted. It uses a hash table to keep key-value entry and constructs buckets of the table with *red-black tree*. It also tracks LRU list of entries and evicts bunch of LRU entries if the storage becomes too narrow as normal caches do. GCMA implements Cleancache and Frontswap back-ends simply using the *dmem* by giving its reserved area as an area for values of key-value pairs and let Cleancache and Frontswap use it with information for evicting clean pages and swapping pages as a key and content of the page as a value. When GCMA needs a page used by Cleancache or Frontswap for contiguous memory allocation, it evicts the page associated entry with *dmem* interface. Figure 3 depicts overall architecture of GCMA.

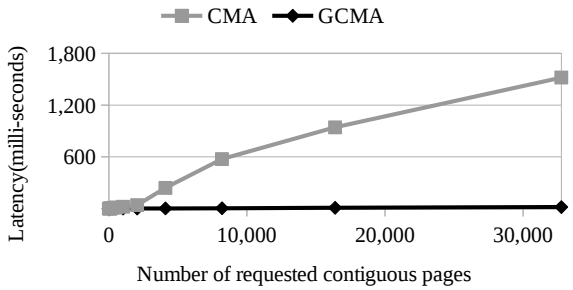


Figure 4: Averaged allocation latency.

5. EVALUATION

5.1 Evaluation Environments

Component	Specification
Processors	900 MHz quad-core ARM Cortex-A7
Memory	1 GiB LPDDR2 SDRAM
Storage	Class 10 SanDisk 16 GiB microSD card

Table 1: Raspberry Pi 2 Model B specifications.

We use `Raspberry Pi 2 Model B` single-board computer [19] as our evaluation environment. It is one of the most popular low-end mini computers in the world. Detailed specification is described in Table 1.

Name	Specification
Baseline	Linux rpi-v3.18.11 + 100 MiB swap + 256 MiB reserved area
CMA	Linux rpi-v3.18.11 + 100 MiB swap + 256 MiB CMA area
GCMA	Linux rpi-v3.18.11 + 100 MiB Zram swap + 256 MiB GCMA area

Table 2: Configurations for evaluations.

Detailed system configurations we use in evaluations is described in Table 2. Because Raspberry Pi development team provides forked Linux kernel optimized for Raspberry Pi via *Github*, we use the kernel rather than vanilla kernel. The kernel we selected is based on `Linux v3.18.11`. We represent the kernel as `Linux rpi-v3.18.11`. Raspberry Pi has used reservation technique for contiguous memory allocation from the beginning. After Linux implemented CMA, Raspberry Pi started to support CMA from November 2012 [16]. However, Raspberry Pi development team has found CMA problem and decided to support CMA in unofficial way only [12]. As a result, Raspberry Pi default configuration is reservation technique yet.

Our evaluation focus on the following three factors: First, latency of CMA and GCMA. Second, effect of CMA and GCMA on a real workload, Raspberry Pi Camera [18]. Third, effect of CMA and GCMA to system performance.

5.2 Latency of Contiguous Memory Allocation

To compare contiguous memory allocation latency of CMA and GCMA, we issue contiguous memory requests with varying allocation sizes. The first request is for 64 pages (256 KiB) and the size is doubled for subsequent requests until the last request is for 32,768 pages (128 MiB). We give 2 seconds interval between each allocation to minimize any effect to other processes and the system. We repeat this 30 times with both CMA and GCMA configuration.

The average latencies of CMA and GCMA are shown in Figure 4. GCMA shows `14.89x` to `129.41x` faster latency compared to CMA. Moreover, CMA failed once for 32,768 pages allocation while GCMA did not even though the workload has run with no background jobs. Even without any background job, CMA could meet secondary client page that need to be moved out because any movable page can be located inside CMA reserved area. In the case, moving the page out would consume lots of time and could fail in worst case. The worst case actually happened once during 32,768 pages allocation. On the other hand, because only evicting clean pages and swapping pages can be located inside reserved area of GCMA, GCMA wouldn't need to discard pages out unless memory intensive background workload exists. Moreover, CMA code is much bigger and slower than GCMA because CMA has to consider complicate situations of movable pages and migration while GCMA needs to consider only *dmem*. That's why GCMA shows much lower latency than CMA even without any background workload.

5.3 Distribution of Latencies

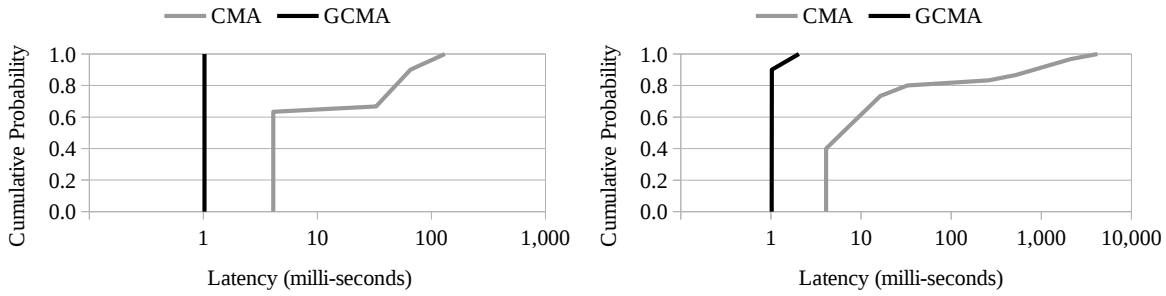
For predictability comparison between CMA and GCMA, we do the contiguous memory allocation workload again with only 1,024 contiguous pages requests and draw CDF of latencies. In this evaluation, we first do the workload without any background job to show latency of CMA and GCMA itself without any interference. After that, we do the workload with a background job to show latencies under realistic memory stress. For the background job, we use `Blogbench` benchmark. The benchmark is a portable file system benchmark that simulates a real-world busy blog server.

The result is shown in Figure 5. In ideal case without any other background jobs, GCMA latencies mostly lie under 1 millisecond while CMA latencies are anywhere from 4 milliseconds to even more than 100 milliseconds. Under Blogbench as a background job, GCMA latencies mostly lie under 2 milliseconds while CMA latencies lie from 4 milliseconds to even more than 4 seconds. The result says that GCMA latency is not only fast but also predictable and insensitive to background stress compared to CMA.

Even without a background workload, CMA shows much dispersed latency than GCMA because CMA is slower than GCMA basically due to complexity and has more chance of secondary client page clean-up. With a background workload, CMA could meet more secondary client pages to clean-up because any movable page of background job can be inside CMA reserved area. In contrast, because only evicting clean pages and swapping pages can be inserted to GCMA reserved area, GCMA would not meet secondary client pages to clean-up unless the background job has sufficiently large memory footprint.

5.4 Raspberry Pi Camera Latency

Though result from Section 5.3 is impressive, it shows only a potential of GCMA, not any performance effect on a real workload. That's why we evaluate latency of photo shots using Raspberry Pi Camera in this section.



(a) Without Blogbench

(b) With Blogbench

Figure 5: CDF of 1,024 contiguous pages allocation latency.

For a photo shot, Raspberry Pi 2 usually allocates 64 contiguous pages 25 times asynchronously using a kernel thread and keeps the allocated memory without release as long as possible [16]. The maintained memory is used for subsequent photo shots. Therefore, only the first photo shot is affected from contiguous memory allocation. For convenient evaluation, we configure the system to release every memory allocated for camera as soon as possible to make subsequent photo shots to be also affected.

We measure latency of each photo shot using Raspberry Pi Camera 30 times under every configuration with 10 seconds interval between each shot to eliminate any effect from previous shots. Each configuration uses Reservation technique, CMA, and GCMA for the photo shot. To explicitly show the effect of contiguous memory allocation, we measure contiguous memory allocation latencies caused for each photo shot under CMA and GCMA.

Without a background job, every configuration shows similar photo shot latency of about 0.9 seconds. There is no difference between CMA and GCMA though GCMA shows much faster latency than CMA from Section 5.3. There are two reasons behind this. First, though CMA is about 15 times faster than CMA for each allocation required by the camera, absolute time for the allocation is only 1 millisecond for CMA, 69 micro-seconds for GCMA. Because a photo shot issues the allocation only 25 times, the allocation latency comprises only a small portion of the entire photo shot latency. Second, CMA can do allocation almost without any page migration if there are no background jobs and the number of required pages is not many. In other words, the environment is too optimal.

To show real latency under a realistic environment, we do the camera workload with Blogbench in background as we did in Section 5.3 for a simulation of a realistic background workload. To minimize scheduling effect on latency, we set priority of photo shot process higher than that of background workload using `nice` [9] command.

The result is described in Figure 6. CMA shows much slower camera shot latency while GCMA shows similar latency with Baseline using Reservation technique. In the worst case, CMA even requires about 9.8 seconds to do a photo shot while GCMA requires 1.04 seconds in the worst case. Contiguous memory allocation latencies also show similar but more dramatic result. Latencies of CMA lie between 4 milliseconds and 10 seconds while GCMA's latencies lie between 750 micro-seconds and 3.32 milliseconds. This result means contiguous memory allocation using CMA produces extremely high and unpredictable latency under a realistic situation. With this evaluation result, it's not surprising that CMA on Raspberry Pi is not officially supported [12].

5.5 Effect on System Performance

	Baseline	CMA	GCMA
lat_ctx(usec)	147.36	143.525	142.93
bw_file_rd(MiB/s)	511.77	517.6	519.73
bw_mem_rd(MiB/s)	1426.33	1438.33	1434.5
bw_mem_wr(MiB/s)	696.5	701.25	699.966

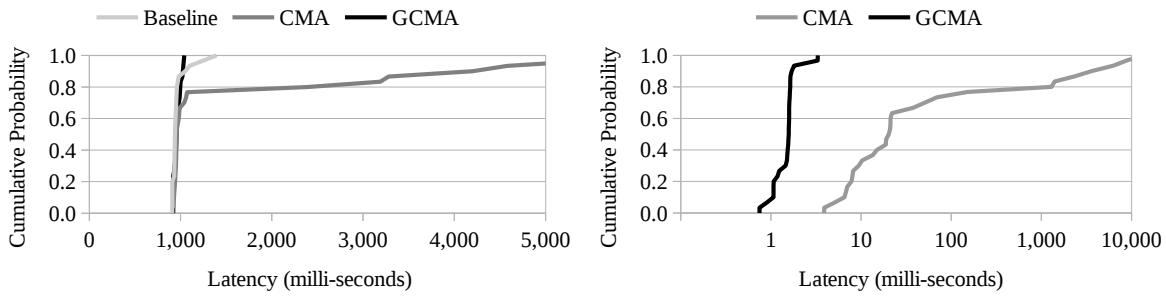
Table 3: LMbench measurement result.

Finally, to show performance effect of CMA and GCMA on system, we measure system performance under every configuration using two benchmarks. First, we run a micro-benchmark called lmbench [11], 3 times and get an average of results to show how performance of OS primitives is affected by CMA and GCMA.

The result is depicted in Table 3. Each line shows an averaged measurement of context switch latency, file read bandwidth, memory read bandwidth, and memory write bandwidth. CMA and GCMA tend to show improved performance compared to Baseline because of memory utilization though the difference is not so big. Differences between CMA and GCMA are just in margin of error.

To show performance under realistic workload, we run Blogbench job 3 times and measure average score normalized by Baseline configuration result. At the same time, we continuously issue photo shots on the background with 10 seconds interval as described in Section 5.4 to simulate realistic contiguous memory allocation stress on system.

The result is shown in Figure 7. Writes / reads performances with background job are represented as `Writes / cam` and `Reads / cam`. CMA and GCMA show enhanced performance for every case though the enhancements are not remarkably big. Those performance gains came from enlarged memory space that are rescued from reservation. GCMA shows even better enhancement than CMA owing to the light overhead of reserved area management that benefited from its simple design and characteristic of secondary clients. Moreover, GCMA get less performance degradation from background contiguous memory allocations than CMA owing to its fast latency. As a summary, GCMA is not only faster than CMA but also more helpful for system performance.



(a) Photo shot

(b) Contiguous memory allocation

Figure 6: CDF of photo shot / following memory allocation latencies under Blogbench.

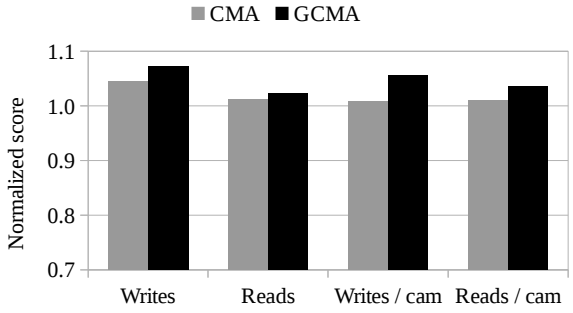


Figure 7: Blogbench performance with CMA and GCMA.

6. CONCLUSION

Physically contiguous memory allocation is still a big problem for low-end embedded devices. For example, Raspberry Pi, a popular credit card sized computer, still uses reservation technique despite of low memory utilization because hardware solutions such as Scatter/Gather DMA or IOMMU were too expensive and a software solution, CMA, was not effective.

We introduced GCMA, a contiguous memory allocator that guarantees fast latency, successful allocation, and reasonable memory space efficiency. It achieves those goals by using the reservation technique and effectively utilizing the reserved area. From our evaluation on Raspberry Pi 2, while CMA increases latency of a photo shot on Raspberry Pi from 1 second to 9 seconds in the worst case, GCMA shows no additional latency compared to the reservation technique. Our evaluation also shows that GCMA not only outperforms latency but also improves system performance as CMA does.

7. ACKNOWLEDGEMENTS

This work was supported by the National Research Foundation of Korea(NRF) grant funded by the Korea government(MSIP) (NRF-2015R1A2A2A01005995).

8. REFERENCES

[1] A. Arcangeli. Transparent hugepage support. *KVM Forum*, 2010.

[2] J. Corbet. Cleancache and Frontswap. *http://lwn.net/Articles/386090/*, 2010.

[3] P. Denning. Before memory was virtual. 1997.

[4] M. Freedman. The compression cache: virtual memory compression for handheld computers. 2000.

[5] J. Jeong, H. Kim, J. Hwang, J. Lee, and S. Maeng. DaaC: device-reserved memory as an eviction-based file cache. In *in Proc. 21th Int. Conf. Compilers, architectures and synthesis for embedded systems*, page 191. ACM Press, Oct. 2012.

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

[7] M. S. Johnstone and P. R. Wilson. The memory fragmentation problem. *ACM SIGPLAN Notices*, 34(3):26–36, Mar. 1999.

[8] N. P. Jouppi. Cache write policies and performance. In *Proceedings of the 20th annual international symposium on Computer architecture - ISCA '93*, volume 21, pages 191–201, New York, New York, USA, June 1993. ACM Press.

[9] D. MacKenzie. nice. *Linux man page*, 2010.

[10] D. Magenheimer. In kernel memory compression. *http://lwn.net/Articles/545244/*, 2013.

[11] L. McVoy and C. Staelin. lmbench: Portable Tools for Performance Analysis. *USENIX annual technical conference*, 1996.

[12] msperl. Rpiconfig. *http://elinux.org/RPiconfig*, 2014.

[13] M. Nazarewicz. Contiguous Memory Allocator. In *Proceedings of LinuxCon Europe 2012*. LinuxFoundation, 2012.

[14] S. Park. introduce gcma. *http://lwn.net/Articles/619865/*, 2014.

[15] A.-W. Robert Love. The Buffer Cache. In *Linux-Kernel Manual: Guidelines for the Design and Implementation of Kernel 2.6*, page 348. 2005.

[16] T. C. Ruth Suehle. Automatically Share Memory. In *Raspberry Pi Hacks: Tips & Tools for Making Things with the Inexpensive Linux Computer*, page 95. 2013.

[17] A. Team. Low RAM. *http://s.android.com/devices/tech/low-ram.html*, 2013.

[18] E. Upton. Camera board available for sale! *https://www.raspberrypi.org/camera-board-available-for-sale/*, 2013.

[19] E. Upton. Raspberry Pi 2 on sale now at $35. *https://www.raspberrypi.org/raspberry-pi-2-on-sale/*, 2015.