CLMalloc: Contiguous memory management mechanism for large-scale CPU-accelerator hybrid architectures

Yushuqing Zhang[a], Kai Lu*[a], Wenzhe Zhang[a]
[a]College of Computer, National University of Defense Technology, China 410073
* kailu@nudt.edu.cn

ABSTRACT

Heterogeneous accelerators play a crucial role in improving computer performance. General-purpose computers reduce the frequent communication between traditional accelerators with separate memory and the host computer through fast communication links. Some high-speed devices such as supercomputers integrate the accelerator and CPU on one chip, and the shared memory is managed by the operating system, which shifts the performance bottleneck from data acquisition to accelerator addressing. Existing memory management mechanisms typically reserve contiguous physical memory locally for peripherals for efficient direct memory access. However, in large computer systems with multiple memory nodes, the accelerator's memory access behavior is limited by the local memory capacity. The difficulty of addressing accelerators across nodes prevents computers from maximizing the benefits of massive memory. This paper proposes a contiguous memory management mechanism for a large-scale CPU-accelerator hybrid architecture (CLMalloc) to simultaneously support the different types of memory requirements of CPU and accelerator programs. In simulation experiments, CLMalloc achieves similar (or even better) performance to the system functions malloc/free. Compared with the DMA-based baseline, the space utilization of CLMalloc is increased by 2×, and the latency is reduced by 80% to 90%.

Keywords: accelerator, computer system, memory management, dynamic allocator

1. INTRODUCTION

Heterogeneous accelerators are an important means to improve computer performance, especially in supercomputers and other devices that require high performance. Traditional heterogeneous accelerators have independent memory[1,2], and ordinary computers usually use products such as NVIDIA[3] to achieve high-speed interconnection between CPU and GPU, so as to improve data replication speed. Another solution is to build a hybrid hardware architecture that allows the CPU and accelerator to share memory, mostly used in large computer systems, as shown in Figure 1.

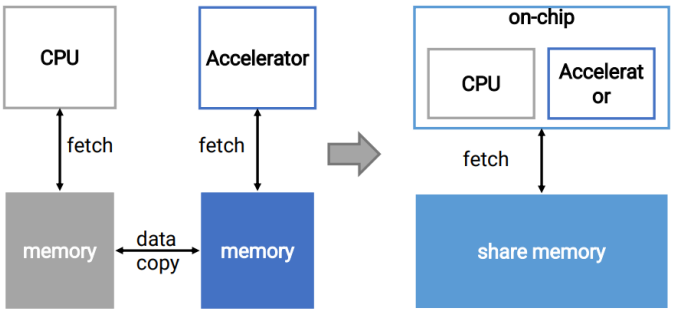

Figure 1. New accelerator architecture.

IOMMU[4-6] uses a separate MMU to map peripheral-accessible physical addresses to host physical addresses, allowing accelerators to directly access memory. But compared with directly sharing physical memory, independent MMU is not conducive to the mixed programming of CPU-accelerator programs. For large-scale computer systems, the data volume often reaches dozens of GB, which may occupy multiple memory hardware. The operation of IOMMU to replace the page table may become very complicated, leading to the difficulty of accelerator addressing. Therefore, we consider the

Third International Symposium on Computer Engineering and Intelligent Communications (ISCEIC 2022), edited by Xianye Ben, Proc. of SPIE Vol. 12462, 1246203 · © The Authors. Published under a Creative Commons Attribution CC-BY 3.0 License · doi: 10.1117/12.2660807

case where CPU and the accelerator use the same MMU and manage the memory uniformly through the operating system.

This hybrid architecture presents new challenges:

Firstly, the design philosophies of CPUs and accelerators are based on different goals. General-purpose operating systems typically use a virtual memory mechanism to provide various programs with contiguous memory addresses, while limiting physical memory fragmentation to the page level. In contrast, a single-purpose external device often can only rely on the CPU to reserve a continuous physical memory for it, and realize direct access through simple segment mapping. DMA is a continuous memory management mechanism designed by Linux for external devices to directly access memory.

Secondly, some peripherals use 32-bit addressing and need to occupy some lower address space, so the actual memory address of the CPU may be in the state of "available memory-peripherals-available memory-peripherals", resulting in a vacuum between available memory addresses.

In addition, due to the powerful virtual memory mechanism, the priority of allocating contiguous physical is too low to support data calculation in large-scale computer systems. As a local memory management mechanism, DMA has a limited allocation range. When Linux tries to expand the memory range reserved for DMA, both the address vacuum caused by peripherals and the boundaries of memory hardware may cause the accelerator to fail to allocate contiguous memory.

This paper proposes CLMalloc, a new memory allocation mechanism for large-scale CPU-accelerator hybrid architectures, allowing CPU to intelligently skip the address vacuum and provide continuous memory regardless of memory boundaries with the similar performance to system functions.

In order to facilitate the mixed programming of CPU and accelerator, we provide a pair of interfaces: cmt_malloc and cmt_free. Users can perform different types of memory allocation by switching interface functions, such as calling cmt_malloc and cmt_free to process a large amount of calculation data, and calling malloc/free in other cases. We conducted various stress tests in simulated scenarios, compared with the existing method, the space utilization rate of CLMalloc is increased by nearly 2×, and the time cost is reduced by about 80% to 90%.

The innovations of this paper are as follows:

- We propose a memory management mechanism for the new CPU-accelerator hybrid architecture.
- We provide a unified memory interface for user programs to support CPU-accelerator program.
- CLMalloc has been applied to our prototype system.

2. RELATED WORKS

We have not found other memory management studies for hybrid CPU-Accelerator architectures. Most of the current work is based on IOMMU. Linux also provides contiguous physical memory allocation methods for peripheral memory management, such as Huge Page[7], DMA[8,9], etc. This section explains why these approaches fail to meet the new architectural requirements.

2.1 IOMMU

IOMMU reserves a fixed piece of physical memory for peripherals mapped to host physical memory addresses through independent translation tables. In this way, a peripheral can have the same virtual memory view as CPU, but it does not know the physical address and mapping of the actual access. As mentioned earlier, some DMA operations may cause memory segmentation faults, and IOMMU needs to avoid this problem by remapping the translation table.

IOMMU effectively reduces the latency of frequent communication between the accelerator and CPU while limiting its writable range to ensure safety. However, different MMUs do not support mixed programming of CPU applications and accelerator applications simultaneously. For example, in a hybrid program, the ordinary assignment store and the large-scale computational data store are obviously accessed by different hardware. Different translation tables will cause addressing and caching difficulties.

2.2 Huge Page

Huge Page refers to a page that is different from normal pages and can reach a size of 2MB to 1GB. Huge page sizes can ensure memory contiguousness over a wider range and cause less TLB overhead. However, excessively large pages are likely to cause the same problems as the segment allocation mechanism, resulting in inefficient memory management. We focus on the operating system level and mostly talk about making allocated pages contiguous.

2.3 DMA

DMA directly allocating continuous physical memory for peripherals s through the buddy system. Linux reserves a fixed range of contiguous memory when it initializes DMA, divided into several cma_area managed by an overall control structure *node*. In each cma_area, DMA can allocate any size of memory space, which consists of multiple continuous pages.

Different memory requests can share a cma_area. However, if each request is relatively large and must exclusive a cma_area, it will also cause serious waste. Memory migration by DMA can coalesce fragments of free memory, but it is too expensive and runs the risk of inconsistencies caused by remapping page tables.

3. OVERVIEW

3.1 Architecture

We implemented CLMalloc in the Linux kernel, and the architecture is shown in Figure 2. CLMalloc consists of two modules: 1) In the operating system, the segment allocator uses DMA to reorganize the intermittent memory space and allocate it efficiently; 2) At the runtime layer, we provide a dynamic memory allocator. The dynamic memory allocator uses the memory buffer pool to reduce overhead in high-frequency operations and multi-threading scenarios, and provides interface functions cmt_malloc and cmt_free to the user.

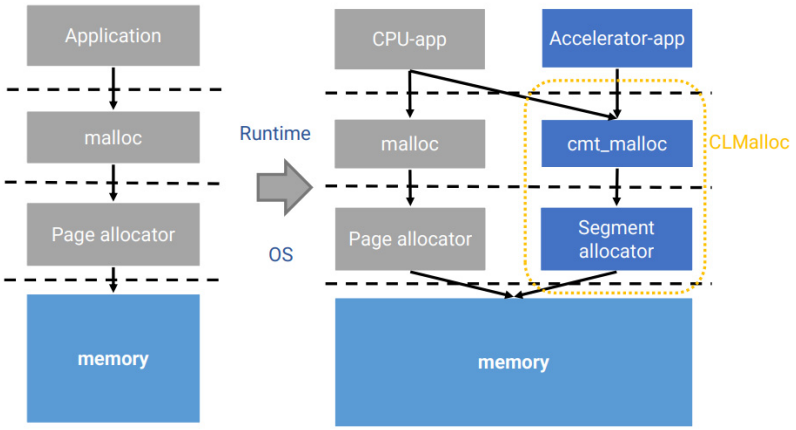

Figure 2. Memory support for accelerators.

3.2 Baseline

The design of DMA demonstrates the effectiveness of using cma_area to allocate contiguous pages, and provides an idea for managing multiple noncontiguous memory addresses. Since the size and number of cma_area can be specified manually, we treat each address range of available memory as one cma_area. As shown in Figure 3, each cma_area has a private control structure cma to flexibly control its internal memory allocation. Multiple memory regions can be accessed sequentially via *node*, from which we implement a baseline in a prototype system with multiple memory hardware and achieve contiguous memory allocation between adjacent memory hardware.

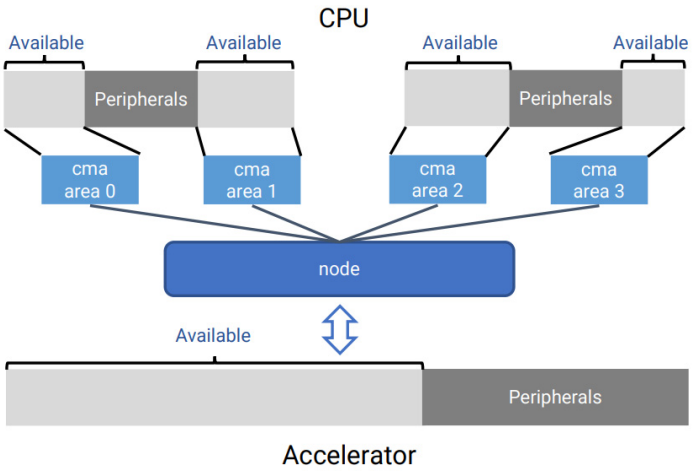

Figure 3. Memory organization based on cma_area.

Each cma control structure contains a bitmap to manage the allocation of the cma_area. DMA traverses the array of cma_area to find the first cma_area with enough space when receiving a memory requirement. Then DMA finds the range of available pages in its bitmap, calculates the corresponding page frame number *pfn* according to the start index of this range, and passes it to the buddy system to allocate physical memory.

We assume that the size of a cma_area is 4GB, and the baseline manages memory requirements less than 4GB by DMA. When receiving a memory requirement of more than 4GB, the baseline first calculates the total number of required cma_area *k*, and traverses the cma_area array in reverse order. The baseline will start from the first completely free cma_area it finds, and will allocate the memory of the adjacent cma_area in turn, until the allocated cma_area reaches *k* or finds a non-free cma_area. If the total number of allocated pages when the traversal is stopped is greater than the number of pages requested by the memory count, then the excess pages in the header of the last allocated cma_area will be released, otherwise all allocated pages will be released and continue traverse.

The baseline does not require the addition of additional data structures and is easy to understand and implement, but performance depends heavily on the order of queries. As shown in Figure 4, faced with 4 memory requirements of the same size, all allocations in 4(a) are successful, while 4(b) will generate memory migration. Also, each cma_area is invisible to others. Only when a cma_area is accessed, can it be determined whether it is free or non-free. Even if there are less than *k* adjacent free cma_area in the reverse order allocation process, the baseline still needs to do a lot of repetitive work before reporting the error, which greatly affects the response time of the request.

Figure 4. Performance differences causing by different request sequences.

4. CLMALLOC DESIGN

4.1 Global Bitmap

The memory organization of the baseline offers the possibility to allocate contiguous memory between adjacent cma_area. We propose a global bitmap mapping algorithm to further reduce memory waste and time delay. Global bitmap consists of a private bitmap for each cma_area. When a memory request is received, the algorithm first calculates the available range in the global bitmap, and passes the corresponding page frame base address *pfn* and the number of cma_area start_zone to *node*. The method for computing start_zone and *pfn* is shown in Table 1.

Table 1. Get start address.

Input: bitmap, count
Output: pfn, start_zone

```
1   bitmap_maxno ← size of global bitmap
2   if count is larger than bitmap_maxno then
3      return NULL
4   end if
5   lock cma_mutex
6   bitmap_no ← index of the next zero in bitmap
7   if bitmap_no is larger than bitmap_maxno then
8      unlock cma_mutex
9      return NULL
10  end if
11  unlock cma_mutex
12  j ← 0
13  for each cma do
14     j ← j + cma_bitmap_maxno
15     if j is larger than bitmap no then
16        j ← j - cma_bitmap_maxno
17        start_zone ← this cma
18        pfn ← base_pfn + bitmap_no - j
19        break
20     end if
21  end for
22  return pfn
```

Node allocates virtual memory starting from start_zone. Starting from pfn the buddy system allocates all the remaining local physical pages of the cma_area with the index start_zone. When the number of pages allocated by the buddy system exceeds *count*, it frees the excess pages at the end of the last cma_area, which can form a larger free space with the next cma_area.

Through the global bitmap, the size and location of the remaining space of cma_area are transparent to each other. When receiving a large memory request, *node* does not need to repeatedly try allocation to find the available memory range, while start_zone and *pfn* limited the memory fragmentation in one page. Compared with baseline, the global bitmap algorithm can respond to more memory requests and cause the most negligible memory waste. If the granularity of the bitmap is suitable, the only possible cause of fragmentation before memory overload is alignment operations, which is inevitable for both distribution algorithms.

This method will also bring some problems, such as:

1) Will the additional bitmap make the available space cramped?

2) Will the addresses across memory banks slow down their actual memory-access speed for smaller memory requests?

3) After a program releases its memory, will subsequent allocations to this range still cause fragmentation?

We analyzed these issues as follows.

First of all, the global bitmap can artificially select the number of pages (power of 2) represented by 1 bit to set an appropriate size of the bitmap. Compared with general memory requests, this overhead is negligible. Second, if the access delay exists, when a memory request(less than 4GB) cannot be allocated on a separate cma_area, it will only cause once. Actually, different memory bars are still continuous from the perspective of accelerators, and accelerators can access them through the segment map.

Finally, when some programs release their memory, all the following requests smaller than this newly available area will continue to use it economically. The remaining available area can always form a larger continuous space with the next section of released memory. Although we cannot predict when and where a piece of memory will be released, this method has still significantly reduced the possibility of memory migration.

4.2 Pre-Allocation

DMA uses buddy system to allocate physical pages, but buddy system requires that the number of pages allocated each time be a power of 2. When the requested number is between 2^i and 2^{i+1}, 2^{i+1} pages will be allocated, but in the global bitmap, only the bits corresponding to (2^i, count] will be set to 1. The bits in (count, 2^{i+1}] are still zero, and for the operating system, these pages are still free, but they cannot be allocated again, in fact.

In order to maximize the advantages of the global bitmap, we propose pre-allocation. When the accelerator is initialized, all physical memory will be allocated but not marked in the global bitmap. For a program, CLMalloc still calculates the starting *pfn* through the global bitmap as it needs but only allocates the virtual memory and maps it to a piece of physical memory that is the same size. Also, this mapping will be cancelled when the program releases memory, while the physical memory is not really released.

This optimization allows more fine-grained memory use and reduces multiple operations on physical memory, providing good time performance. We will give specific data in Section 5.

4.3 Dynamic Memory Allocator

In order to make CLMalloc performance comparable to system functions, we add a dynamic memory allocator in its runtime layer. Referring to existing memory allocators[10-19], we set a buffer pool maintaining a global heap and P processor heaps, as shown in Figure 5.

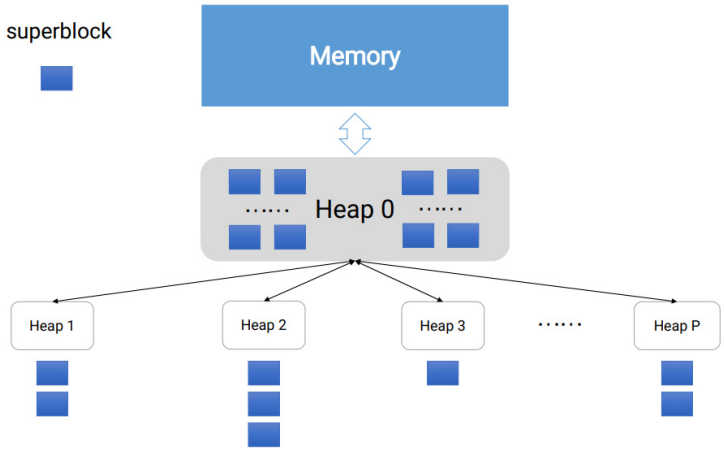

Figure 5. Buffer pool.

The global heap is responsible for taking a large amount of memory from the operating system and organizing it as an array of superblocks, then distributing these superblocks to processor heaps with corresponding memory requests. If all the superblocks are not enough, the global heap will take more memory from the operating system and create new superblocks. There is no interaction between the processor stacks. When the usage rate is lower than a specific threshold *f*, data on the superblock will be migrated to other superblocks (held by the same heap). The global heap will recycle free superblocks for reuse.

The buffer pool dramatically reduces the actual number of accesses to virtual memory and avoids much possible thread competition relying on this heap-superblock model. In order to distinguish it from the original system function malloc/free, we provide a simple interface: cmt_malloc/cmt_free. Both CPU and accelerator programs can freely select the memory allocation method through two pairs of interface functions.

5. EVALUATION

5.1 Environment

All experiments are based on Linux 4.19.46 kernel, Ubuntu 18.0.4. 256MB of memory is reserved when DMA is initialized, divided into 16 cma_area on average. The default page size is 4KB, and the Huge-Page size is 2MB.

5.2 Availability

The availability of CLMalloc, Baseline, and Huge-Page evaluates different types of memory requirements. To facilitate evaluating the usability of these methods for multiple consecutive large memory allocations, we conduct multiple memory allocation tests in units of 20MB (1*cma_area < size < 2*cma_area). There are five types of memory requests:

- 0 < size < Huge-page 1)
- Huge-Page < size < 1*cma_area 2)
- 1*cma_area < size < 16*cma_area 3)
- 20MB * T

  0 < T ≤ 8 (the number of cma_areas divided by 2) 4)

  8 < T ≤ 12 (the maximum number of times that DMA can theoretically allocate) 5)

Table 2 shows that Huge-Page cannot allocate more contiguous memory than its page size at the physical level. A 20MB memory requirement requires two full cma_areas, and Baseline can support up to 8 such allocations before DMA performs memory migration to merge memory fragments. CLMalloc can support 12 allocations, reaching the theoretical maximum allocation range, which can effectively reduce the frequency of memory migration.

Table 2. Availability of Huge-Page, Baseline and CLMalloc.

	Huge page (2M)	Baseline	CLMalloc
1MB	√	√	√
8MB	×	√	√
100MB	×	√	√
20MB × 8	×	√	√
20MB × 9	×	×	√
20MB × 12	×	×	√

5.3 Performance

We use the baseline as a benchmark to test the performance of Global-bitmap, Global-bitmap + pre-allocation and Global-bitmap + pre-allocation + buffer pool (CLMalloc) under 8 to 12 times of 20MB memory requirements.

Figure 6 presents the latencies for the four test subjects. Global-bitmap has a certain improvement in performance compared to baseline, but due to the inherent waste of buddy system, it can only support one more allocation. After the pre-allocation of physical memory, the memory fragments are fully utilized, and the delay caused by multiple memory accesses is also significantly reduced. On this basis, CLMalloc uses the buffer pool to reduce the time cost by about 80% to 90%.

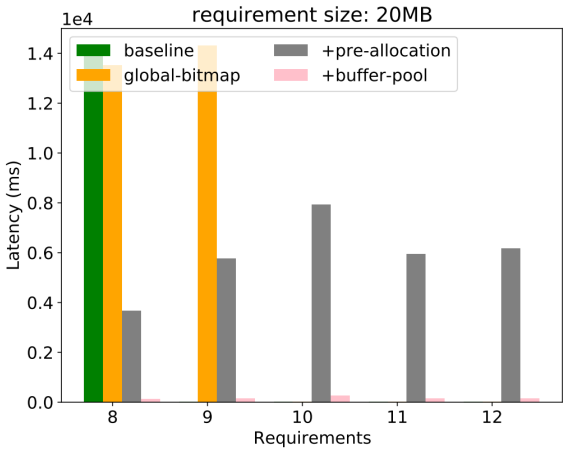

Figure 6. Buffer pool.

Our goal is to provide the accelerator with contiguous physical pages without compromising CPU performance. Therefore, we conducted a simulated stress test on the system functions malloc/free and CLMalloc, and collected their latency in two scenarios of receiving continuous large memory requests and frequent memory access requests.

We collected the latency of system functions malloc/free and CLMalloc when executing 100, 1000, 10000, 100000 random consecutive large memory requests by 1, 2, 4 and 8 threads. Figure 7 shows that CLMalloc achieves similar performance to local allocation in simulated tests.

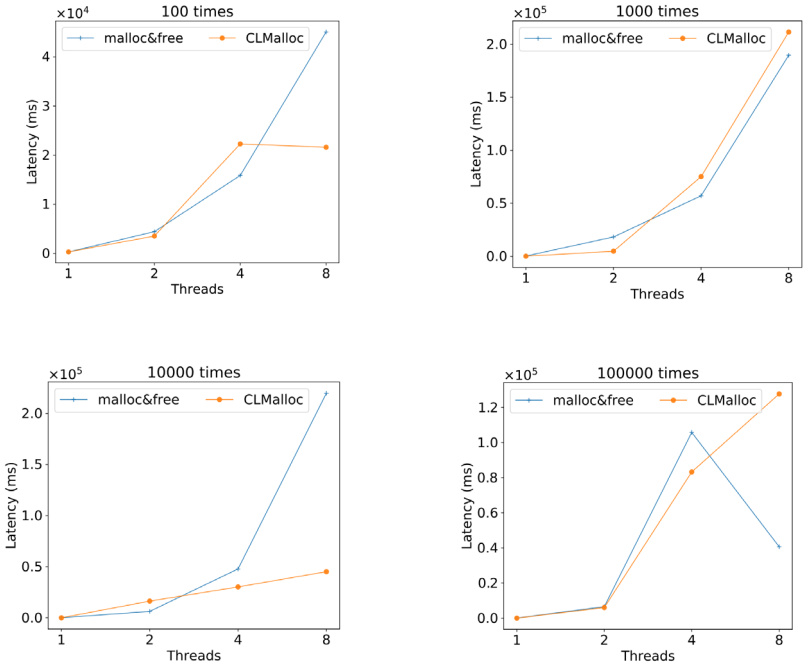

Figure 7. Multiple allocation of random large memory.

In the frequent memory request section, we selected four data structures commonly used in programming: hashmap, vector, priority queue and unordered map for evaluation. Figure 8 shows the latency of malloc/free and CLMalloc executing 100,000 scale inputs in the ycsb dataset by 1, 2, 4 and 8 threads. The multiple heaps provided by the buffer pool effectively reduce the probability of multi-thread contention. Therefore, the latency of malloc/free increases linearly as the degree of parallelism increases, while CLMalloc significantly outperforms system functions.

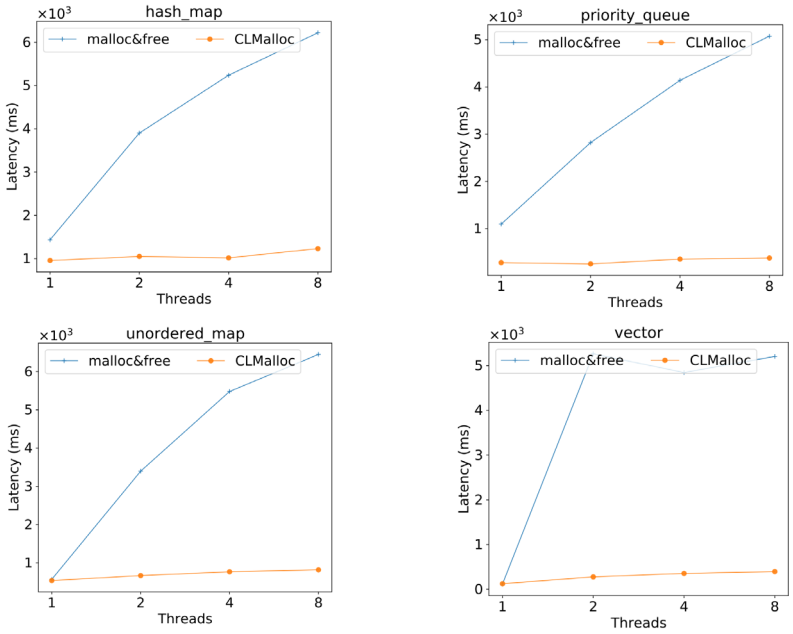

Figure 8. Frequent memory requests.

6. CONCLUSION

We designed CLMalloc, a memory management mechanism for the CPU-accelerator hybrid architecture, which can simplify the addressing and execution of accelerator programs without affecting the CPU program and improve the efficiency of its programming and caching.

Through the reorganization of memory, CLMalloc provides a structure that can continuously access the available memory for use by intermittently addressed CPU and continuously addressed accelerators. The construction of a global bitmap enables contiguous memory allocation of arbitrary size, at a specified location. Double-layer precise positioning is performed by start_zone and *pfn*, and memory fragmentation is limited to one page. Pre-allocation avoids memory waste and operating system allocation errors caused by the partner system. The memory buffer pool further reduces the number of memory-accesses and speeds up the system.

CLMalloc has now been applied to the prototype system. In future research work, we will make detailed adjustments to the behavioral logic of the memory allocator according to project requirements and actual application scenarios of the operating system. Compared with the completeness of the storage performance experiment, the amount of testing the algorithm accepts during the experiment is still relatively small. In the subsequent experimental links, we will increase the practical application of the test, which will help discover more hidden problems.

ACKNOWLEDGEMENT

This paper is supported by NSF 61902405, National High-level Personnel for Defense Technology Program (2017-JCJQ-ZQ-013).

REFERENCES

[1] Kurth A , Vogel P , Capotondi A , et al. HERO: Heterogeneous Embedded Research Platform for Exploring RISC-V Manycore Accelerators on FPGA[J]. 2017.

[2] B. D. de Dinechin et al., ”A clustered manycore processor architecture for embedded and accelerated applications,” 2013 IEEE High Performance Extreme Computing Conference (HPEC), 2013, pp. 1-6, doi: 10.1109/HPEC.2013.6670342.
[3] https://www.nvidia.cn/design-visualization/nvlink-bridges/
[4] Malka M , Amit N , Ben-Yehuda M , et al. rIOMMU: Efficient IOMMU for I/O Devices that Employ Ring Buffers[J]. ACM SIGPLAN Notices, 2015, 50(4):355-368.
[5] Vogel P , Marongiu A , Benini L . Exploring Shared Virtual Memory for FPGA Accelerators with a Configurable IOMMU[J]. IEEE Transactions on Computers, 2019:1-1.
[6] Markuze, Alex, Tsafrir, et al. True IOMMU Protection from DMA Attacks: When Copy is Faster than Zero Copy[J]. ACM SIGPLAN Notices: A Monthly Publication of the Special Interest Group on Programming Languages, 2016, 51(4):249-262.
[7] William Stallings, translated by Chen Yu, ”Operating Systems Internals and Design Principles,” Beijing: Publishing House of Electronics Industry. 2007.
[8] Mao Decao, Hu Ximing. ”Scenario Analysis of Linux Kernel Source Code,” Zhejiang University Press.2001
[9] Yu Xiangzhan, Yin Lihua. ”Dynamic shared memory buffer pool technology” [J]. Journal of Harbin Institute of Technology. 2004
[10] Abraham Baer Galvin et al. ”Operating System Concepts,” Beijing: Higher Education Press. 2005.
[11] Berger E D, Zorn B G. McKinley K S. ”OOPSLA 2002: Reconsidering custom memory allocation,” ACM SIGPLAN Notices.2013. 48(4S): 46-57
[12] Michael M M. ”Scalable lock-free dynamic memory allocation,” ACM SIGPLAN Notices. 2004. 3906): 35-46
[13] Hudson R L. Saha B. AdI-Tabatabai A R, et al. ”McRT-Malloc: A scalable transactional memory allocator,” Proceedings of the 5th International Symposium on Memory Management. Ottawa. Canada, 2006: 74-83
[14] Evans J. ”A scalable concurrent malloc (3) implementation for FreeBSD,” Proceedings of the BSDCan Conference. Ottawa. Canada.2006: 1-14
[15] Seos, KimJ, Lee J. ”Sfmalloc: A lock-free and mostly synchronization-free dynamic memory allocator for manycores,” Proceedings of the 2011 International Conference on Parallel Architectures and Compilation Techniques (PACT). Galveston Island. USA, 2011: 253-263
[16] Herter J, Backes P, Haupenthal F.et val. ”CAMA: A Predictable Cache-Aware Memory Allocator,” Proceedings of the 2011 23rd Euromicro Conference on RealTime Systems (ECRTS). Porto, Portugal, 2011: 23-32
[17] Liu R, Chen H. ”SSMalloc: A low-latency. Locality-conscious memory allocator with stable performance scalability,” Proceedings of the Asia-Pacific Workshop on Systems. Seoul, Korea, 2012: 15
[18] Jang, Byunghyun, Schaa, et al. ”Exploiting Memory Access Patterns to Improve Memory Performance in Data-Parallel Architectures.”[J]. IEEE Transactions on Parallel & Distributed Systems, 2011, 22(1):105-118.14 Y. Zhang et al.
[19] R. Ron, E. Weissman, K.K. Vansenathan, et al. ”Context switching mechanism for processing cores with general-purpose CPU cores and tightly coupled accelerators:,” CN104205042B[P]. 2019.