Memory Contention Aware Swap Space Management

Su-Wei Yang, Ya-Shu Chen

***Abstract*—GPGPUs have been widely applied for high-performance applications, however, the performance bottleneck is the data movement from the limited physical memory. The cost of swap in/out pages significant impact the applications' performance under memory oversubscription. This study proposes a swap space management to reduce the performance degradation caused by memory thrashing. Evaluation results show that our proposed reduced lengthened latency up to 16% under the real-platform.**

***Index Terms*—memory contention, memory thrashing, memory bottleneck, memory oversubscription, swap space management**

I. Introduction

With the increased need of computing power and data parallelism accessing, the general-purpose graphics processing units (GPGPU) are widely applied to provide high performance, especially for huge data accessing applications, such as machine learning or image processing. Based on the report of NVIDIA GPU and Google TPU, with the increased computing units, the performance bottleneck of applications is main on the limited memory capacity and memory bandwidth [1]–[3]. To improve the performance, GPGPUs are now equipped with unified memory to share the memory with CPUs. However, when the GPGPU's task required data size exceeds the physical memory, i.e., when the memory is oversubscribed for multiple tasking, data must be swapped in memory from storage systems and some data must be swapped out from memory to storage systems. Real GPGPU applications suffer significant performance degradation from the memory oversubscription [4]. The performance degradation is main on the thrashing which occurs when GPGPU accessing the same page frequently and results in data are frequently swapped in/out from memory.

The thrashing problem caused by the memory oversubscription is difficult by the varied data accessing pattern among applications. The pessimistic control data accessing pattern might decrease the execution parallelism and then results in performance degradation. This research is motivated by the performance need of GPGPU applications and proposes the swap space management for such systems to minimize the lengthened response from thrashing.

Different from previous studies, we present to directly access the storage system [5] with our proposed swap space manager to resolve the thrashing problem. The idea is practicable because the storage bandwidth is significantly increased by the new bus protocol, such as the read speed and write speed of PCI-express based SSD are more than $2GB/s$ [6]. By directly accessing data in the storage system, the performance degradation from frequently swap in/out [7] can be avoided without decreasing the execution parallelism. Based on the idea, we then proposed the swap space management including how to choose the task to access data from storage system directly and how much data shall be accessed from storage directly. Our contributions are as follows:

- Directly access storage idea is proposed to avoid the thrashing problem.
- The priority assignment is proposed to minimize the performance impact from memory contention among multiple application execution.
- A swap space manager is presented to chose the victim task and determine the ratio of the swap size.

The remainder of this paper is organized as follows. Section II presents a survey of the relevant studies. Section III presents the system model and formulate the considered problem. In Section IV, our swap space management including priority assignment, the victim chosen, and the swap space ratio are proposed. Section V provides our evaluation of the proposed approach in various workload. Finally, Section VI offers a conclusion.

II. Related Work

To minimize the latency from the data sharing between heterogenous multiores [8], [9], the unified memory architecture (UMA) has been increasingly prevalent in recent years, such as Nvidia TX2 [10] and AMD Vega architecture [11]. Under UMA, the data sharing between heterogeneous multicores, such as CPU and GPU, can be simplified to access the same memory without extra translation and mapping time. The studies [11]–[15] have evaluated the performance of UMA and show that the performance can be significantly improved.

Although applied UMA architecture can reduce the data accessing time between heterogeneous multicores, the memory management under the heterogenous multicore system is still the performance bottleneck. The performance of machine learning-based applications shows the bottleneck in memory oversubscription [1]. The latency from the data accessing is getting large when multiple applications executing simultaneously by the report in [16], [17]. The latency of the data accessing mainly cause by memory contention between multiple application executing and the frequent swap in/out from memory oversubscription.

Researches [10], [15], [18]–[22] have explored the memory contention issue. Authors in [10], [15] regulated applications' memory requests by adjusting the CPU frequency and memory bandwidth for each application to reduce the memory contention. Authors in [18], [21] proposed to control memory

service interval for GPU kernel or to limit certain GPU kernels in certain interval for reducing the memory contention. Authors in [19], [20] dispatched certain memory channel to certain cores to reduce the memory interference. Authors in [22] proposed the application slowdown model (ASM) by periodically assigning the highest priority to the current executing application for a short length of time to minimizes the memory interference.

To resolve frequently swap in/out among applications with memory oversubscription which is called thrashing problem, studies [2], [8], [9], [23]–[26] have presented memory throttling mechanism. Authors in [2], [8] presented new evicted policies to reduce memory thrashing. Authors in [9], [24] presented a new way to handle Translation Lookaside Buffer (TLB) between the virtual memory and physical memory to reduce the thrashing. Authors in [25] designed a new architecture by virtually enlarging cache size through the under-utilized register file (RF) with lightweight ISA to reduce cache thrashing. redAuthors in [26] presented to suspend one streaming multiprocessor (SM) in GPU which is triggered by a request starting swapping data out, and to resume the corresponding GPU SM while there are sufficient capacity in system memory.

Different from the related studies, in this work, we explore both memory contention and thrashing issues. In order to reduce memory contention and avoid thrashing problem, a memory contention-aware priority assignment and swap space management have been proposed for the heterogenous multicore system with the limited memory and the high-speed storage.

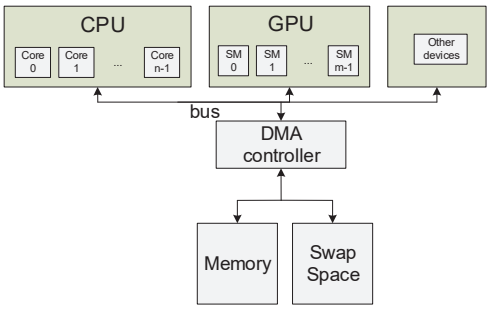


Fig. 1: Unified Memory Architecture

III. System Model

This paper explores a memory contention aware swap space management to minimize the response time of a set of tasks ($T = \{T_1, T_2, ...T_n\}$) under the system consists of CPU with multicores, GPU, memory controller, memory, and Solid State Disk (or SSD) as shown in Fig 1. To eliminate the data transmission between heterogeneous cores, in this paper, we consider the unified memory architecture (UMA) [8], [9], [11]–[15], [27] applied to communicate between CPU and GPU. On such a system, a task T_i is consist of n_i independent subtasks $T_{i,j}$. Each subtask first allocates data from memory (or SSD), then reads data to memory(SSD), then executes in GPU, and then writes data back to memory (SSD). When multiple tasks are executed in the system, the memory access becoming the performance bottleneck [15], [17]–[21]. The example is shown in Fig. 2, when multiple tasks are executed, tasks might be delayed by memory accessing queue. Assuming there are two tasks, T_1 and T_2, send requests to access memory. First, task T_1 is blocked by task T_2. Later, task T_2 is blocked by T_1 based on the first-in-first-out service fashion. Both tasks suffered delay because of memory contention.

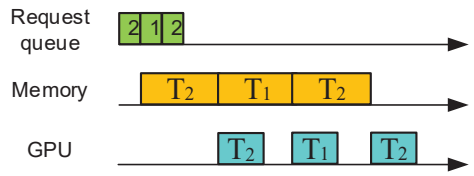


Fig. 2: Memory contention

Modern operating systems support swap space in the storage system to support multiple application execution. When the total required memory size is larger than the system memory, applications can be still executed. When the required data is in the storage system (e.g., SSD), the swapping has been triggered. The operating system chose a victim task (which is not a currently executing task) to swap data from memory to SSD, and then swap data of the currently executing task from SSD to memory. The swap time is related to larger compared to task's execution time. The example is shown in Fig. 3, the request from T_1 triggered the swapping because of insufficient memory space, then T_2 is swapped out from memory to SSD, and T_1 swapped from SSD to memory for executing. Later, the request from T_2 triggered the swapping because of insufficient memory space, then T_1 is swapped out from memory to SSD, and T_2 swapped from SSD to memory for executing. The similar procedure might be repeated and introduces a larger latency to tasks from data swapping cost. This is a so-called thrashing problem. Based on the report in NVIDIA [4]and [26], [28], the thrashing problem is getting serious with the increased number of cores or streaming multiprocessor in GPU.

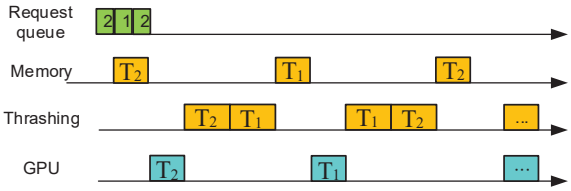


Fig. 3: Thrashing

In this paper, we propose memory contention-aware priority assignment to reduce memory contention, and propose a swap space management avoid thrashing problem to improve the system performance with the given system memory size.

Algorithm 1 Memory Contention-aware Priority Assignment

Input: The number of all tasks N, the response time for each application i with multiple applications interference $C_i(N)$, the response time for each application without other applications interference $C_i(1)$, and the number of sub task for each application n_i.

1: Calculate the application contention degree $W_i \leftarrow \frac{C_i(N)-C_i(1)}{C_i(1)} \times \frac{1}{n_i}$
2: Sort all tasks based on the non-increased value of W_i
3: Assign the higher priority to task with the larger W_i
4: Scheduling all tasks on the memory commend queue based on the assigned priority

Algorithm 2 Swap Space Management

Input: Given the memory allocation of each application $MEM(i)$, the system memory MEM_{sys}, and the number of applications n

1: Calculate the required swap space $SWAP_{req} \leftarrow \sum_{i=1}^{N} MEM(i) - MEM_{sys}$
2: Sort all applications with non-decreased priority
3: **for** i is 1 to n **do**
4: **if** $SWAP_{req} \geq MEM(i)$ **then**
5: $\alpha_i \leftarrow 1$
6: **else**
7: $\alpha_i \leftarrow \lceil \frac{SWAP_{req}}{MEM(i)} \rceil$
8: **end if**
9: $SWAP_{req} \leftarrow SWAP_{req} - \alpha_i MEM(i)$
10: **if** $SWAP_{req} = 0$ **then**
11: $break$
12: **end if**
13: $i \leftarrow i + 1$
14: **end for**
15: Execute tasks based on the assigned priority and swap ratio in the system

IV. Algorithms

To resolve the above technique issues, we then proposed two algorithms (1) memory contention-aware priority assignment in Section IV-A to reduce latency of a task from memory contention, and (2) swap space management to eliminate latency caused by the thrashing in Section IV-B.

A. Memory Contention-aware Priority Assignment

The response time of a task is lengthened by memory contention. To minimize the latency caused by memory contention, we propose to schedule tasks based on the memory contention degree. In this section, we present how to determine the memory contention degree of the given tasks. First, we execute each task T_i alone and then execute all tasks at the same time to respectively get the response time of $C_i(1)$ and $C_i(N)$ [16]. The difference between $C_i(1)$ and $C_i(N)$ is the interference between tasks. The larger interference (i.e., $\frac{C_i(N)-C_i(1)}{C_i(1)}$) implies larger overhead suffered by task T_i, and the task shall be scheduled earlier to reduce the memory contention. Moreover, the considered system has multiple computation units which can execute multiple subtasks at the same time when the required data are ready. As a result, when the number of subtasks (n_i) of a task T_i is larger, the interference can be reduced by parallel execution. Base on the previous observation, we estimate the memory contention degree by $W_i = \frac{C_i(N)-C_i(1)}{C_i(1)} \times \frac{1}{n_i}$. The detail is shown in Algorithm 1. After we estimate the memory contention degree of tasks, the larger value of W_i of T_i has a higher priority. During run-time, we schedule tasks based on the assigned priority.

B. Swap Space Management

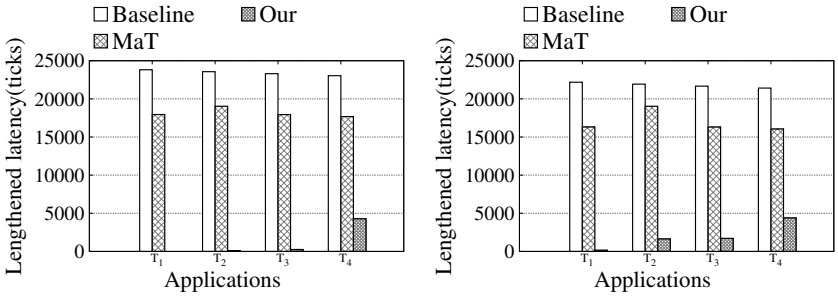
As discussed in the previous section, the thrashing problem is getting serious for data-oriented applications. Based on the UMA, CPU and GPU can directly access memory without buffering. With increased bus bandwidth between memory and storage system, and with increased read/write throughput of the storage system, we propose to execute lower priority tasks in the storage system directly to avoid thrashing overhead. The next issue is how to assign the swap space ratio of tasks to minimize the lengthened time (from thrashing) of all tasks. The idea of our proposed swap management is to chose the lower priority tasks as victim to be accessing data from SSD. After acquiring the priority obtained from memory contention degree in Algorithm 1, we first evaluate how much memory pages we should assign to swap space, and then limited the memory accessing of the application with lower priority, by accessing data from swap space directly, until the system memory is sufficient.

The detail is shown in Algorithm 2. Given the information including the required memory size of each application and the total memory size in the system, this algorithm assigns how much swap space of a task T_i (α_i) to applications. First, estimate the memory $MEM(i)$ requirement of each application T_i. When the required total memory size is larger than the system memory (i.e., $\sum_{i=1}^{N} MEM(i) > MEM_{sys}$), based on the priority assignment in the previous section, we chose the lowest priority as swap candidate first. If the remaining required memory is still larger than the system memory, we then chose the next swap candidate which is the lowest priority have not chosen. The swap ratio is determined by the $\lceil \frac{SWAP_{req}}{MEM(i)} \rceil$ for each task T_i to prevent thrashing and page allocation fragmentation. The procedure is stopped only when the required memory of non-swapping applications is no larger than the system memory. Note that our swap space assignment can also be applied to the current system which can not execute tasks in storage system directly. Under the system without supporting directly accessing storage, the thrashing overhead can still be reduced by our approach.

V. Performance Evaluation

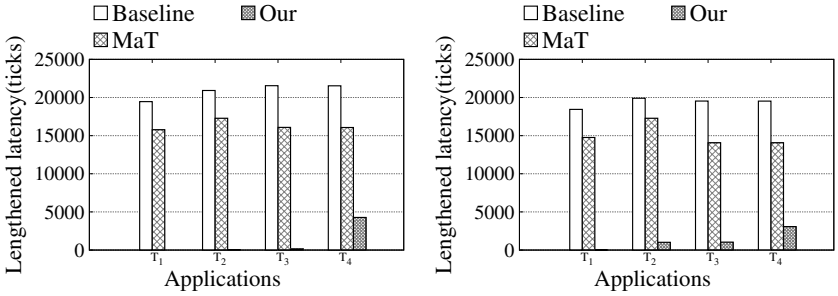
A. Experimental Setup

This section presents the performance evaluation to show the capability of our proposed methods. We compare our



(a) low computing (b) high computing

Fig. 4: Applications with the same number of subtasks



(a) low computing (b) high computing

Fig. 5: Applications with varied number of subtasks

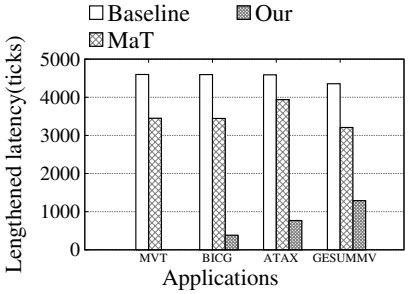


Fig. 6: Polybench

proposed policy with *Baseline* which swap out the least used task when the memory is not enough, and Memory-aware Throttling *MaT* [26] which suspends Stream Processor (SM) on GPU when the memory is not enough. Both *Baseline* and *MaT* service memory requests in first-in first-out (FIFO) fashion.

In the following experiments, the physical memory sizes are set as $18\ GB$ and $240\ MB$ to show the performance results under varied memory size, and the required memory sizes of synthesis workload and benchmark [29] are respectively $24\ GB$ and the $320\ MB$. The bandwidth of read/write data from memory is set as $20000\ MB/s$ which is measured from Nvidia jetson TX2, and the bandwidth from SSD are set as $492\ MB/s$ and $445\ MB/s$ [6], respectively. The synthesis workload consists of 4 tasks, and each tasks consists of $5-11$ subtasks. The computing execution times of subtasks are ranged from 1 tick to 6 times of thrashing time (which can be estimated by the ratio of the memory size to SSD bandwidth). The performance metric is *lengthened latency* which is the difference between the response time to execute the corresponding task alone and the response time to execute the corresponding task among multiple tasks under the swap manager.

B. Experimental Results

This section evaluates the response time of the proposed swap space management. The horizontal axis in Fig. 4, Fig. 5, and Fig. 6 indicates the identification of each task, whereas the vertical axis in these figures is *lengthened latency*. The

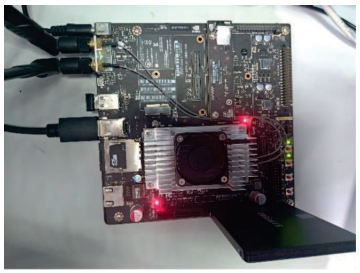


Fig. 7: Real Platform

smaller identification of the task represents the task with the higher priority which is assigned by our proposed Algorithm.

In Fig. 4, we first evaluate the workload in which all tasks have the same number of tasks (i.e., 5 subtasks). Fig. 4(a) and Fig. 4(b) respectively show the workloads with low computation demand and high computation demand. As shown in Fig. 4, the *lengthened latency* of each task is significantly reduced by our proposed because we avoid thrashing. The response time of T_4 is larger because it is the lowest priority tasks and some requests of T_4 is executed in SSD directly. By doing so, there is no lengthened latency suffered by the highest priority application, even the lengthened latency suffered by the lowest priority application is reduced about 76 % and 81 %, respectively, compared to *MaT* and *Baseline*. *MaT* outperformed than *Baseline*, because *MaT* suspend one streaming multiprocessor (SM) in GPU to reduce thrashing cost. However, the execution parallelism of *MaT* is decreased by the decreased number of kernels. As a result, our proposed outperformed than *MaT*. With increased computation demand of applications, as shown in Fig. 4(b), the lengthened latency suffered by the lowest priority application is reduced about 73 % and 79 %, respectively, compared to *MaT* and *Baseline*. Compared the results in Fig. 4(a) and Fig. 4(b), the performance gap between *MaT* and *Baseline* is decreased because the former reduces the execution parallelism by suspending SM to avoid thrashing. The lengthened latency of Fig. 4(b) is shorter than that of Fig. 4(a) because the thrashing cost might be hidden by the executing data transfer and computation simultaneously under the high computing applications. Based on the similar reason, compared Fig. 4(b) and Fig. 4(a), our lengthened latency is slightly increased on Fig. 4(b) from the effect of computing demands instead of thrashing cost.

In Fig. 5, we then evaluate the workload in which all tasks have the various number of tasks (i.e., T_1, T_2, T_3, and T_4 have respectively 5, 7, 9, and 11 subtasks). As shown in Fig. 5(a), the latency is various among each application because of the different number of subtasks and memory requests. Compared with *Baseline* and *MaT*, our proposed method can still have the shortest lengthened latency of all tasks. There is no lengthened latency suffered by the highest priority application, even the lengthened latency suffered by the lowest priority application is reduced about 73 % and 80 %, respectively, compared to *MaT* and *Baseline*. With increased computation demand of applications, as shown in Fig. 5(b), the lengthened latency suffered by the lowest priority application is reduced about 78 % and 84 %, respectively, compared to *MaT* and *Baseline*.

In Fig 6, the irregular applications, e.g., ATAX, BICG, GESUMMV, and MVT, which are collected from PolyBench/GPU [29] benchmarks are emulated as synthesis workload by only considering the memory request size and computation demand to be evaluated. Based on our proposed priority assignment, MVT and GESUMMV are respectively set as the highest and the lowest priority tasks in the system. As shown in Fig 6, our proposed can still outperform than other methods based on effectively swap space management. There is no lengthened latency suffered by the highest priority application (MVT), even the lengthened latency suffered by the lowest priority application (GESUMMV) is reduced about 60 % and 70 %, respectively, compared to *MaT* and *Baseline*. By eliminating the thrashing cost, our proposed outperforms than other approaches. Note that the response time of *MaT* is irregular as the priorities of tasks because it schedule tasks in first-in-first-out instead of priority-driven.

C. Case Study

Our proposed swap space management is also implemented by CGroup [30] in Ubuntu 16.0.4, kernel 4.4.38, and evaluated in real hardware, Nvidia TX2 with Samsung SSD 860 EVO as the swap space storage, as shown in Fig. 7. On current Linux, there is no way to directly access the swap space, to minimize the thrashing cost, we manage the memory usage of each task by using the Linux container CGroup memory set. For example, when the memory setting of a task which requests $3\ GB$ memory is set as $1\ GB$, and then the memory usage of the corresponding task is set as 1 GB by the CGroup container. Three applications to perform $3GB$ read/write operations in files are evaluated, and the system memory of TX2 only is $8\ GB$. Based on our proposed, we then only assign 1 GB memory [1] to the lowest priority task to avoid swap cost the other two high priority tasks. As shown in Fig. 8, compared to default Linux manager in Nvidia TX2, although the response time of the lowest priority task T_3 is increased $7.25s$, the total latency of all three tasks is reduced about 16%. The experiment results show that our proposed can still maximize performance without directly swap space accessing.

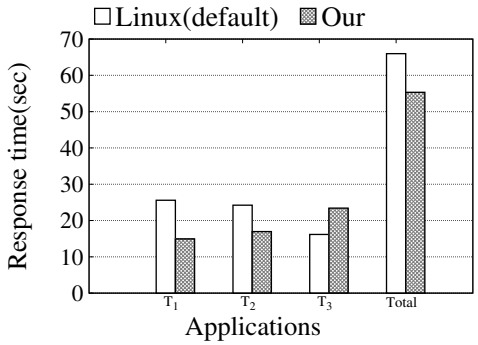


Fig. 8: Performance of Case Study

[1]Linux starting swapping when the system memory is used over than 90%. When the system memory is $8\ GB$, the swapping is triggered when the used memory is larger than 7 GB.

VI. Conclusion

The performance impact from data movement is getting serious under GPGPUs systems due to the huge data requirements of applications and the limited physical memory of systems. This paper first proposes to access the data from the storage system directly to avoid frequently swap in/out. We then propose the swap space management including priority assignment to minimize memory contention, and present how chose the victim task to directly access data on storage systems and determine swap space ratios of the victim tasks. The performance of the proposed algorithms was experimentally assessed using simulations considering directly access storage and using real-platform evaluations to account for the thrashing cost from non-directly storage data accessing. Results show that our proposed reduced lengthened latency up to 84% and 16%, respectively, under simulations and the real-platform, compared with a default setting. Future work will focus on storage and memory mapping, and page management.

References

[1] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers *et al.*, "In-datacenter performance analysis of a tensor processing unit," in *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2017, pp. 1–12.

[2] V. Seshadri, O. Mutlu, M. A. Kozuch, and T. C. Mowry, "The evicted-address filter: A unified mechanism to address both cache pollution and thrashing," in *Proceedings of the 21st international conference on Parallel architectures and compilation techniques*. ACM, 2012, pp. 355–366.

[3] J. Lee, M. Samadi, and S. Mahlke, "Vast: The illusion of a large memory space for gpus," in *2014 23rd International Conference on Parallel Architecture and Compilation Techniques (PACT)*. IEEE, 2014, pp. 443–454.

[4] "Gpu thrashing," http://on-demand.gputechconf.com/gtc/2018/presentation/s8430-everything-you-need-to-know-about-unified-memory.pdf.

[5] "Amd iommu," https://www.amd.com/system/files/TechDocs/48882_IOMMU.pdf.

[6] "Sata bandwidth," https://www.anandtech.com/show/13761/the-samsung-970-evo-plus-ssd-review/7.

[7] G. Lim, C. Min, and Y. I. Eom, "Virtual memory partitioning for enhancing application performance in mobile platforms," *IEEE Transactions on Consumer Electronics*, vol. 59, no. 4, pp. 786–794, 2013.

[8] D. Ganguly, Z. Zhang, J. Yang, and R. Melhem, "Interplay between hardware prefetcher and page eviction policy in cpu-gpu unified virtual memory," in *Proceedings of the 46th International Symposium on Computer Architecture*, ser. ISCA '19. New York, NY, USA: ACM, 2019, pp. 224–235. [Online]. Available: http://doi.acm.org/10.1145/3307650.3322224

[9] R. Ausavarungnirun, V. Miller, J. Landgraf, S. Ghose, J. Gandhi, A. Jog, C. J. Rossbach, and O. Mutlu, "Mask: Redesigning the gpu memory hierarchy to support multi-application concurrency," in *ACM SIGPLAN Notices*, vol. 53, no. 2. ACM, 2018, pp. 503–518.

[10] J. Kim, P. Shin, S. Noh, D. Ham, and S. Hong, "Reducing memory interference latency of safety-critical applications via memory request throttling and linux cgroup," in *2018 31st IEEE International System-on-Chip Conference (SOCC)*. IEEE, 2018, pp. 215–220.

[11] "Amd architecture," https://www.techpowerup.com/gpu-specs/docs/amd-vega-architecture.pdf.

[12] J. Zhang, D. Donofrio, J. Shalf, M. T. Kandemir, and M. Jung, "Nvmmu: A non-volatile memory management unit for heterogeneous gpu-ssd architectures," in *2015 International Conference on Parallel Architecture and Compilation (PACT)*. IEEE, 2015, pp. 13–24.

[13] B. Pichai, L. Hsu, and A. Bhattacharjee, "Architectural support for address translation on gpus: designing memory management units for cpu/gpus with unified address spaces," in *ACM SIGPLAN Notices*, vol. 49, no. 4. ACM, 2014, pp. 743–758.

[14] J. Power, M. D. Hill, and D. A. Wood, "Supporting x86-64 address translation for 100s of gpu lanes," in *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2014, pp. 568–578.

[15] W. Ali and H. Yun, "Work-in-progress: Protecting real-time gpu applications on integrated cpu-gpu soc platforms," in *2017 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 2017, pp. 141–144.

[16] B. M. Tudor, Y. M. Teo, and S. See, "Understanding off-chip memory contention of parallel programs in multicore systems," in *2011 International Conference on Parallel Processing*. IEEE, 2011, pp. 602–611.

[17] S. Bardhan and D. A. Menascé, "Predicting the effect of memory contention in multi-core computers using analytic performance models," *IEEE Transactions on Computers*, vol. 64, no. 8, pp. 2279–2292, 2014.

[18] S. H. Kim, D. Choi, W. W. Ro, and J.-L. Gaudiot, "Complexity-effective contention management with dynamic backoff for transactional memory systems," *IEEE Transactions on Computers*, vol. 63, no. 7, pp. 1696–1708, 2013.

[19] G. Jia, G. Han, A. Li, and J. Lloret, "Coordinate channel-aware page mapping policy and memory scheduling for reducing memory interference among multimedia applications," *IEEE Systems Journal*, vol. 11, no. 4, pp. 2839–2851, 2015.

[20] D. Lee, L. Subramanian, R. Ausavarungnirun, J. Choi, and O. Mutlu, "Decoupled direct memory access: Isolating cpu and io traffic by leveraging a dual-data-port dram," in *2015 International Conference on Parallel Architecture and Compilation (PACT)*. IEEE, 2015, pp. 174–187.

[21] S.-H. Lo, C.-R. Lee, Q.-L. Kao, I.-H. Chung, and Y.-C. Chung, "Improving gpu memory performancewith artificial barrier synchronization," *IEEE transactions on parallel and distributed systems*, vol. 25, no. 9, pp. 2342–2352, 2013.

[22] L. Subramanian, V. Seshadri, A. Ghosh, S. Khan, and O. Mutlu, "The application slowdown model: Quantifying and controlling the impact of inter-application interference at shared caches and main memory," in *Proceedings of the 48th International Symposium on Microarchitecture*. ACM, 2015, pp. 62–75.

[23] Z. Zhuang, C. Tran, J. Weng, H. Ramachandra, and B. Sridharan, "Taming memory related performance pitfalls in linux cgroups," in *2017 International Conference on Computing, Networking and Communications (ICNC)*. IEEE, 2017, pp. 531–535.

[24] A. Basu, J. Gandhi, J. Chang, M. D. Hill, and M. M. Swift, "Efficient virtual memory for big memory servers," in *ACM SIGARCH Computer Architecture News*, vol. 41, no. 3. ACM, 2013, pp. 237–248.

[25] J. Wang, Q. Wang, L. Jiang, C. Li, X. Liang, and N. Jing, "Ibom: An integrated and balanced on-chip memory for high performance gpgpus," *IEEE Transactions on Parallel and Distributed Systems*, vol. 29, no. 3, pp. 586–599, 2017.

[26] C. Li, R. Ausavarungnirun, C. J. Rossbach, Y. Zhang, O. Mutlu, Y. Guo, and J. Yang, "A framework for memory oversubscription management in graphics processing units," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 2019, pp. 49–63.

[27] Y. Suzuki, S. Kato, H. Yamada, and K. Kono, "Gpuvm: Gpu virtualization at the hypervisor," *IEEE Transactions on Computers*, vol. 65, no. 9, pp. 2752–2766, 2015.

[28] J. Kehne, J. Metter, and F. Bellosa, "Gpuswap: Enabling oversubscription of gpu memory through transparent swapping," in *ACM SIGPLAN Notices*, vol. 50, no. 7. ACM, 2015, pp. 65–77.

[29] S. Grauer-Gray, L. Xu, R. Searles, S. Ayalasomayajula, and J. Cavazos, "Auto-tuning a high-level language targeted to gpu codes," in *2012 Innovative Parallel Computing (InPar)*. Ieee, 2012, pp. 1–10.

[30] "Cgroup," https://www.kernel.org/doc/Documentation/cgroup-v1/cgroups.txt.