Poster: Designing a Memory Disaggregation System for Cloud

Wonsup Yoon
KAIST

Jisu Ok
KAIST

Sue Moon
KAIST

Youngjin Kwon
KAIST

ABSTRACT

Memory disaggregation is a new datacenter paradigm separating compute and memory nodes. While memory disaggregation improves memory utilization and scalability, it poses challenges for cloud applications, particularly in terms of high tail latency. Existing memory disaggregation systems focus on optimizing the disaggregation stack, but it does not always guarantee excellent application performance. We review existing memory disaggregation and tail-optimized systems and explain their limitations in this context. We also propose two preliminary solutions: asynchronous page fault handling and a faulty request classifier. The emulation result shows that asynchronous page fault handling reduces tail latencies by 50% compared to synchronous handling.

CCS CONCEPTS

• **Computer systems organization → Cloud computing**.

KEYWORDS

memory disaggregation,cloud computing,tail latency

ACM Reference Format:
Wonsup Yoon, Jisu Ok, Sue Moon, and Youngjin Kwon. 2023. Poster: Designing a Memory Disaggregation System for Cloud. In *ACM SIGCOMM 2023 Conference (ACM SIGCOMM '23), September 10, 2023, New York, NY, USA.* ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/3603269.3610854

1 INTRODUCTION

Memory disaggregation is a datacenter design paradigm that decomposes a computer system into separately managed compute and memory nodes connected by a fast interconnect such as RDMA. By dynamically allocating memory resources on demand, it improves memory utilization, memory scaling across hardware boundaries, and high availability due to separate hardware fault domains [3].

Yet, for all the benefits, memory disaggregation introduces new challenges to cloud applications. Cloud applications are vulnerable to high tail latency. The inherent design of cloud applications, characterized by a fan-out architecture, renders a mere 1% of latency spikes capable of causing substantial deterioration in user experience [4]. Under the memory disaggregation setup, any latency increase from remote memory access calls for attention in application performance, particularly for those whose average response time should stay within a few microseconds.

The main design goal has been designing a high-performance memory disaggregation stack. Infiniswap first introduced a remote memory paging system [6]. Subsequent studies have lowered paging latency [1, 10, 14, 19] and avoided paging latency [15, 17, 18].

Nevertheless, we have found out that the existing techniques used for the high-performance disaggregation stack do not always lead to good application performance. Our prior work uses synchronous page fault handling to shorten handling latency as much as possible [19]. However, when it comes to cloud applications, synchronous handling introduces head-of-line blocking: a request inducing page faults blocks pending requests without page faults. The head-of-line blocking of subsequent requests results in higher latencies, especially at the tail.

In this paper, we seek a new memory disaggregation system design for tail-sensitive cloud applications. We review existing systems and their limitations in this scenario. Then, we introduce our preliminary solutions, possible outcome, and challenges.

2 BACKGROUND AND RELATED WORK

Memory disaggregation systems. Memory disaggregation is a system design that splits computing and memory into separate nodes. Memory disaggregation systems are implemented in many different ways: as a kernel feature [1, 6, 10, 14, 16, 19] or as a library feature [15, 17, 18, 20]. However, their objectives are the same: reducing or avoiding remote memory access latency. To achieve the objectives, for example, DiLOS [19] and Hermit [14] eliminate unnecessary tasks in the critical path by handling them asynchronously. As a result, the average page fault handling latency is reduced to 3.3 μs in the most recent study [19].

Systems for tail-sensitive cloud applications. General-purpose OS design is not well suited to tail-sensitive cloud applications. To achieve lower latencies, researchers have replaced Linux's stock networking stack and scheduler with tail-optimized ones. For network stack, many systems bypass the kernel to shorten the network data path and successfully provide microsecond-scale latencies [2, 8, 12]. Another approach is designing a tail-sensitive scheduler. As ZygOS has demonstrated, scheduling algorithms play an essential role in providing microsecond-scale tail latency with high dispersion [13]. It also has introduced scheduling algorithms other than the partitioned first-come-first-serve (FCFS) model for lowering tail latency. After ZygOS, Shinjuku has realized preemptive scheduling (processor sharing model) for microsecond-scale requests [7], Shenango relocates cores in a few microseconds [11], and Persephone has designed an application-ware scheduler using header information [5].

3 PROBLEM STATEMENT

In this section, we discuss how cloud applications perform on memory disaggregation setup. Throughout this paper, we focus on a paging-based memory disaggregation system.

Synchronous page fault handling. One of the techniques for high-performance memory disaggregation systems is *synchronous page fault handling*. If a page fault exception occurs, the exception

ACM SIGCOMM '23, September 10, 2023, New York, NY, USA

ACM ISBN 979-8-4007-0236-5/23/09…$15.00
https://doi.org/10.1145/3603269.3610854

handler issues an RDMA page fetch request and waits upon its completion by busy polling. This technique provides low page fault latency. However, we argue that the low page fault latency does not always result in low request handling latency, especially at the tail.

Head-of-line blocking. The main disadvantage of synchronous page fault handling is the head-of-line blocking (HoLB) problem. In the FCFS scheduler, widely used for cloud applications, requests with page faults would block subsequent requests in the same queue, many of which are requests without page faults. Moreover, during the blocking, the CPU employs busy polling to check completion. It wastes lots of CPU cycles, even though there are pending requests ready to be handled.

Existing solutions. HoLB is a well-known problem in the FCFS scheduler. To mitigate HoLB, cloud systems employ work-stealing queues or processor sharing. However, none of these systems fundamentally solves HoLB in synchronous page fault handling. Work-stealing queue designs approximate centralized FCFS [7]. Thereby if the number of ongoing page faults is identical to the number of worker threads, the HoLB persists. Shinjuku's preemptive scheduler-based processor sharing is another way to prevent HoLB, but it has a limitation: 5 μs preemption frequency. It is too large for scheduling remote page fault handling, whose latency is 3.3 μs. Increasing the preemption frequency is also impossible. Higher frequency would induce high overhead and crashes [5].

4 PRELIMINARY SOLUTIONS

In this section, we introduce our approaches, possible outcomes, and challenges.

Setup. To estimate possible outcomes, we conduct an experiment with emulations. We measure throughput and latencies (average, p50, p95, p99, maximum) of the Redis key-value store with page fault handling. We emulate the page fault handling by inserting delays in the Redis server. For values, we use the latency metrics of DiLOS [19]. We use two kinds of delays: sync delay for synchronous tasks and async delay for asynchronous tasks. Sync delay is implemented by busy waiting, and async delay employs Redis's timer events. For workload, we use GET requests (128 bytes payload and 1024 key space) generated by a `redis-benchmark` tool.

4.1 Asynchronous Page Fault Handling

To prevent HoLB, we suggest *asynchronous page fault handling*. When a page fault exception occurs on the request handler, the exception handler issues its page fetch request and yields its context to another request handler. Later, the page fetch completes, then the original request handler comes back and executes the remaining operations. Using this technique, fetching does not block requests in the pending queue.

Possible outcomes. We model synchronous page fault handling by inserting sync delay. For asynchronous page fault handling, we use both delays: sync delay for the synchronous parts (exception, page request issuing, etc.) and async delay for page fetching. We also include a version without page faults as an upper bound. Table 1 summarizes the evaluation result. Synchronous page fault handling introduces throughput degradation (15%) and about two times higher P99 latency. Asynchronous page fault handling, on the other hand, provides P99 latency close to the no-fault version.

	Tput	Avg	Min	P50	P95	P99	Max
no-fault	172K	0.14	0.04	0.14	0.17	0.23	2.27
sync	146K	0.29	0.08	0.27	0.39	0.55	2.48
async	175K	0.15	0.04	0.14	0.19	0.24	2.78

Table 1: Throughput (reqs/s) and latencies (ms) of key-value stores without page fault handling (no-fault), with synchronous page fault handling (sync), and with asynchronous page fault handling (async).

Challenges. To realize asynchronous page fault handling, we have to design a scheduler that can handle exceptions and context switching in a few microseconds. However, the page fault exceptions are kernel events, while fast context switching mechanisms (such as fiber and green thread) are implemented in userspace. Therefore, we are considering single mode and one address space unikernels [9] which relay kernel events to the application without huge costs.

4.2 Page Fault Predictor

Another way to reduce HoLB is to categorize requests by processing time and to put them in distinct queues [5]. It prevents blocking requests with short processing times by requests with long processing times. We are considering a similar approach in memory disaggregation by predicting page faults. The page fault predictor determines and separates fault-inducing requests into distinct request queues. Then, fault-inducing requests do not block requests without page faults, resulting in lower latencies.

Challenges. However, predicting page faults is a challenging task. How can we know whether a request causes page fault before processing? One of the possible solutions is history-based prediction. On the key-value store, the most recently accessed key-value pair is likely to be cached in the compute node and does not incur a page fault. We can use this characteristic to predict page faults within requests.

5 CONCLUSION

In this paper, we argue that the existing memory disaggregation systems are not suited to tail-sensitive cloud applications. To address this, we suggest two preliminary solutions: asynchronous page fault handling and page fault prediction. Our emulation study shows that our approach would enhance performance, namely throughput and tail latency.

ACKNOWLEDGMENTS

We thank our anonymous reviewers for their helpful comments. This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT). (No. 2022R1A2C2009062)

REFERENCES

[1] Emmanuel Amaro, Christopher Branner-Augmon, Zhihong Luo, Amy Ousterhout, Marcos K. Aguilera, Aurojit Panda, Sylvia Ratnasamy, and Scott Shenker. 2020. Can Far Memory Improve Job Throughput?. In *Proceedings of the Fifteenth European Conference on Computer Systems* (Heraklion, Greece) *(EuroSys '20)*. Association for Computing Machinery, New York, NY, USA, Article 14, 16 pages. https://doi.org/10.1145/3342195.3387522

[2] Adam Belay, George Prekas, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. 2014. IX: A Protected Dataplane Operating System for High Throughput and Low Latency. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. USENIX Association, Broomfield, CO, 49–65. https://www.usenix.org/conference/osdi14/technical-sessions/presentation/belay

[3] Sergey Blagodurov, Mike Ignatowski, and Valentina Salapura. 2021. The Time is Ripe for Disaggregated Systems. https://www.sigarch.org/the-time-is-ripe-for-disaggregated-systems/.

[4] Jeffrey Dean and Luiz André Barroso. 2013. The Tail at Scale. *Commun. ACM* 56, 2 (feb 2013), 74–80. https://doi.org/10.1145/2408776.2408794

[5] Henri Maxime Demoulin, Joshua Fried, Isaac Pedisich, Marios Kogias, Boon Thau Loo, Linh Thi Xuan Phan, and Irene Zhang. 2021. When Idling is Ideal: Optimizing Tail-Latency for Heavy-Tailed Datacenter Workloads with PerséPhone. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles* (Virtual Event, Germany) *(SOSP '21)*. Association for Computing Machinery, New York, NY, USA, 621–637. https://doi.org/10.1145/3477132.3483571

[6] Juncheng Gu, Youngmoon Lee, Yiwen Zhang, Mosharaf Chowdhury, and Kang G. Shin. 2017. Efficient Memory Disaggregation with Infiniswap. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. USENIX Association, Boston, MA, 649–667. https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/gu

[7] Kostis Kaffes, Timothy Chong, Jack Tigar Humphries, Adam Belay, David Mazières, and Christos Kozyrakis. 2019. Shinjuku: Preemptive Scheduling for μsecond-scale Tail Latency. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. USENIX Association, Boston, MA, 345–360. https://www.usenix.org/conference/nsdi19/presentation/kaffes

[8] Antoine Kaufmann, Tim Stamler, Simon Peter, Naveen Kr. Sharma, Arvind Krishnamurthy, and Thomas Anderson. 2019. TAS: TCP Acceleration as an OS Service. In *Proceedings of the Fourteenth EuroSys Conference 2019* (Dresden, Germany) *(EuroSys '19)*. Association for Computing Machinery, New York, NY, USA, Article 24, 16 pages. https://doi.org/10.1145/3302424.3303985

[9] Anil Madhavapeddy and David J. Scott. 2014. Unikernels: The Rise of the Virtual Library Operating System. *Commun. ACM* 57, 1 (jan 2014), 61–69. https://doi.org/10.1145/2541883.2541895

[10] Hasan Al Maruf and Mosharaf Chowdhury. 2020. Effectively Prefetching Remote Memory with Leap. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, 843–857. https://www.usenix.org/conference/atc20/presentation/al-maruf

[11] Amy Ousterhout, Joshua Fried, Jonathan Behrens, Adam Belay, and Hari Balakrishnan. 2019. Shenango: Achieving High CPU Efficiency for Latency-sensitive Datacenter Workloads. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. USENIX Association, Boston, MA, 361–378. https://www.usenix.org/conference/nsdi19/presentation/ousterhout

[12] Simon Peter, Jialin Li, Irene Zhang, Dan R. K. Ports, Doug Woos, Arvind Krishnamurthy, Thomas Anderson, and Timothy Roscoe. 2014. Arrakis: The Operating System is the Control Plane. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. USENIX Association, Broomfield, CO, 1–16. https://www.usenix.org/conference/osdi14/technical-sessions/presentation/peter

[13] George Prekas, Marios Kogias, and Edouard Bugnion. 2017. ZygOS: Achieving Low Tail Latency for Microsecond-Scale Networked Tasks. In *Proceedings of the 26th Symposium on Operating Systems Principles* (Shanghai, China) *(SOSP '17)*. Association for Computing Machinery, New York, NY, USA, 325–341. https://doi.org/10.1145/3132747.3132780

[14] Yifan Qiao, Chenxi Wang, Zhenyuan Ruan, Adam Belay, Qingda Lu, Yiying Zhang, Miryung Kim, and Guoqing Harry Xu. 2023. Hermit: Low-Latency, High-Throughput, and Transparent Remote Memory via Feedback-Directed Asynchrony. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. USENIX Association, Boston, MA, 181–198. https://www.usenix.org/conference/nsdi23/presentation/qiao

[15] Zhenyuan Ruan, Malte Schwarzkopf, Marcos K. Aguilera, and Adam Belay. 2020. AIFM: High-Performance, Application-Integrated Far Memory. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 315–332. https://www.usenix.org/conference/osdi20/presentation/ruan

[16] Yizhou Shan, Yutong Huang, Yilun Chen, and Yiying Zhang. 2018. LegoOS: A Disseminated, Distributed OS for Hardware Resource Disaggregation. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. USENIX Association, Carlsbad, CA, 69–87. https://www.usenix.org/conference/osdi18/presentation/shan

[17] Chenxi Wang, Haoran Ma, Shi Liu, Yuanqi Li, Zhenyuan Ruan, Khanh Nguyen, Michael D. Bond, Ravi Netravali, Miryung Kim, and Guoqing Harry Xu. 2020. Semeru: A Memory-Disaggregated Managed Runtime. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 261–280. https://www.usenix.org/conference/osdi20/presentation/wang

[18] Chenxi Wang, Haoran Ma, Shi Liu, Yifan Qiao, Jonathan Eyolfson, Christian Navasca, Shan Lu, and Guoqing Harry Xu. 2022. MemLiner: Lining up Tracing and Application for a Far-Memory-Friendly Runtime. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. USENIX Association, Carlsbad, CA, 35–53. https://www.usenix.org/conference/osdi22/presentation/wang

[19] Wonsup Yoon, Jisu Ok, Jinyoung Oh, Sue Moon, and Youngjin Kwon. 2023. DiLOS: Do Not Trade Compatibility for Performance in Memory Disaggregation. In *Proceedings of the Eighteenth European Conference on Computer Systems* (Rome, Italy) *(EuroSys '23)*. Association for Computing Machinery, New York, NY, USA, 266–282. https://doi.org/10.1145/3552326.3567488

[20] Yang Zhou, Hassan M. G. Wassel, Sihang Liu, Jiaqi Gao, James Mickens, Minlan Yu, Chris Kennelly, Paul Turner, David E. Culler, Henry M. Levy, and Amin Vahdat. 2022. Carbink: Fault-Tolerant Far Memory. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. USENIX Association, Carlsbad, CA, 55–71. https://www.usenix.org/conference/osdi22/presentation/zhou-yang