JOURNAL OF SEMICONDUCTOR TECHNOLOGY AND SCIENCE, VOL. 23, NO. 5, OCTOBER, 2023
https://doi.org/10.5573/JSTS.2023.23.5.273

ISSN(Print) 1598-1657
ISSN(Online) 2233-4866

Empirical Analysis of Disaggregated Cloud Memory on Memory Intensive Applications

Yeonwoo Jeong, Gyeonghwan Jung, Kyuli Park, Youngjae Kim, and Sungyong Park

Abstract—**Disaggregated Cloud Memory (DCM) is a hypervisor-based solution that allows client node to extend local memory by leveraging underutilized memory from remote node. These two nodes are generally connected through Remote Direct Memory Access (RDMA)-based high-bandwidth InfiniBand networks. DCM has been a viable alternative to mitigate the performance degradation of memory-intensive applications in memory-constrained environments. There has also been a growing interest in developing memory-intensive applications with managed languages (we call managed applications) such as Java and Python. These managed languages are easy to use but introduce unpredictability in memory usage at runtime. Despite the advantage of memory extension in DCM, the empirical studies that analyze the performance impact and overhead of running managed applications in DCM are lacking. This paper presents the results of a comprehensive study of DCM on both managed and unmanaged applications. The experimental results revealed that the performance degradation of unmanaged applications in DCM is only less than 6% due to fast remote paging and optimized page eviction policy. However, Garbage Collection (GC) severely degrades the performance of managed applications when page fault occurs, while DCM mitigates the performance degradation efficiently.**

Manuscript received Jun. 12, 2023; reviewed Aug. 10, 2023; accepted Oct. 7, 2023

Dept. of Computer Science and Engineering, Sogang University, Seoul, Korea
E-mail : akssus12@sogang.ac.kr, siblue202@sogang.ac.kr, kyuripark@sogang.ac.kr, youkim@sogang.ac.kr, parksy@sogang.ac.kr
Corresponding author: Sungyong Park

Index Terms—**Disaggregated cloud memory, memory capacity extension, memory, disaggregation**

I. INTRODUCTION

As demand surges for in-memory application processing including deep learning and graph processing, it becomes imperative to ensure adequate memory capacity for the seamless execution of applications [1-4]. However, due to the physical constraints of memory chipset on the mainboard, the enhancement of memory density within a single computational unit is inherently limited. With the ongoing rise in dataset size and application complexity, the working set size surpasses the physical memory capacity of an individual computational node. While disk swapping traditionally served to augment program availability, the slow disk I/O incurred during memory paging results in a significant performance degradation for applications.

To mitigate the performance degradation of in-memory applications, there are several researches to extend local memory capacity [2, 5, 6]. This is achieved by borrowing the underutilized memory from remote nodes, all linked by a high-speed network such as InfiniBand [7]. A prominent solution among these is Disaggregated Cloud Memory [8] (DCM), a hypervisor-based memory expansion technology. Within this system, the guest OS of a Virtual Machine (VM) perceives the memory of both the local and remote nodes as a singular, unified memory space, offering a promising alternative to accelerating memory- hungry applications. It employs a transparent remote paging mechanism and specialized memory management to locate highly localized memory pages within the local memory space.

Memory extension solutions that leverage remote memory come with several challenges.

Managed application in DCM : Recently, Java Virtual Machine (JVM)-based managed frameworks including Apache Spark [9], SciJava [10] and ImageJ [11] are emerging. These managed applications are easy for users who have little knowledge about programming because JVM takes care of complex memory management. Due to the advantages of managed applications, users often run these applications without much consideration of memory capacity in underlying compute nodes. Memory allocation patterns generated at runtime are unpredictable until the initial execution for most applications. Notably, when the working set of a managed application exceeds the memory capacity allocated to a VM, the resulting page faults can cause significant performance degradation. In such situations, a solution that seamlessly handles page faults without requiring additional resources becomes an attractive alternative.

Meanwhile, the performance impact and overhead of running managed applications in DCM have not been extensively studied. Most of existing works focus on evaluating the effectiveness of DCM and the workloads are limited to the unmanaged applications [12-14]. Indeed, conducting an in-depth analysis of both managed and unmanaged applications is crucial to determine the feasibility and effectiveness of DCM for running memory-intensive applications.

Factors affecting the performance in DCM : In order to answer the benefits and challenges of memory disaggregation, this paper conducts a thorough investigation, with an emphasis on three primary factors that highlight the advantages of DCM: (1) *page fault cost*, (2) *garbage collection*, (3) *page caching*. This paper conducts a comparative study of DCM and OS swap mechanism, focusing on their performance when running memory-intensive applications in a local memory-constrained environment.

We have conducted our experiments on a setup comprising two servers, each equipped with a 28-core Intel Xeon Gold 6330 CPU and 128GB memory, interconnected via a 100Gb/s InfiniBand network. Our research reveals that DCM significantly enhances application performance, achieving up to a 7.27 times improvement over OS swap mechanism. Remarkably,

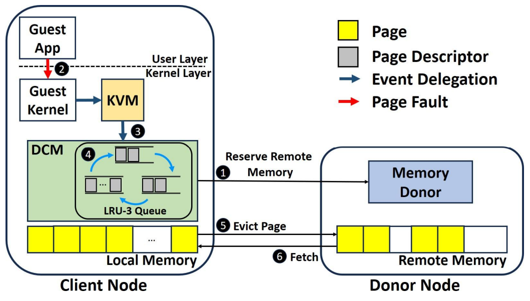


Fig. 1. An Overview of Disaggregated Cloud Memory (DCM).

even when the working set occupies only 60% of local memory space, the performance degradation is limited only to 6% compared to a native memory computing environment. The key observations found through an empirical study are summarized as follows.

1) Fast remote paging and optimized page eviction policy in DCM are particularly effective in unmanaged applications, where users control memory directly. This results in a minimal performance degradation of less than 6% compared to the ideal scenario, even in situations where only 60% of the working set resides in local memory.
2) While running managed applications where memory lifecycle is managed by JVM, the overlap between garbage collection and remote paging can counteract the efficiency gains achieved by fast remote paging and the optimized page eviction policy in DCM.
3) DCM demonstrates dramatic performance improvements over traditional computing environments while performing comparably to native memory computing, showing that it can be a viable alternative to addressing memory-scarce environments.

II. BACKGROUND

1. Disaggregated Cloud Memory

DCM [8] represents a novel VM-based remote memory expansion solution that effectively grants access to substantial memory resources from remote nodes and virtualizes unified memory with local memory. It comprises two interconnected components: a client node and a donor node, linked through a high bandwidth 100 Gb/s InfiniBand network as shown in Fig. 1.

The client node, functioning as a VM node, is further divided into distinct host and guest areas. The host area executes a KVM hypervisor that operates atop the host OS, facilitating the deployment of multiple VMs. These VMs, in turn, host various guest OSes and user applications. According to [15], the access latency of an application to fetch a 4KB memory page from local memory ranges from 10 ns to 256 ns. In contrast, when the application reads the same 4KB memory page from a remote server connected by InfiniBand with a speed of 100Gb/s, it takes approximately 2.8 μs. To mitigate the communication overhead, DCM employs an optimized memory paging. For example, in DCM, when a page is replaced, it is not immediately evicted from local memory. Instead, it is reserved by placing it in another queue that predicts its likelihood of being accessed in the near future. This approach reduces the frequency of remote paging during page faults, effectively implementing a caching scheme to minimize communication overhead.

Fig. 1 illustrates the sequence of steps involved in DCM process of fetching a memory page from a donor node during a page fault event. Before DCM communicates between client node and donor node, a preliminary operation is performed to reserve the memory of the donor node as the swap area. DCM module requests the donor node to reserve a swap region of a certain size (❶). Then, the donor node returns the RDMA buffer addresses of the area, and a channel is established, through which RDMA (read/write) operations can be executed.

When an application running on the guest VM accesses local memory, if the requested page does not exist in the local memory, the guest VM is exited with a page fault event (❷). At that time, the guest kernel delegates the page fault handling to the host kernel/KVM. Then, KVM delegates the responsibility of handling the page fault event to DCM (❸), which performs the necessary memory page replacement. DCM first checks whether there are free spaces in local memory. When the available memory spaces do not exist, it selects a victim page descriptor (❹) that has memory page information according to the LRU-3 policy [8] and evicts corresponding page to the remote node to create free space (❺). Finally, DCM terminates page fault handling by fetching the page (❻) located in the remote node to local memory through RDMA communication.

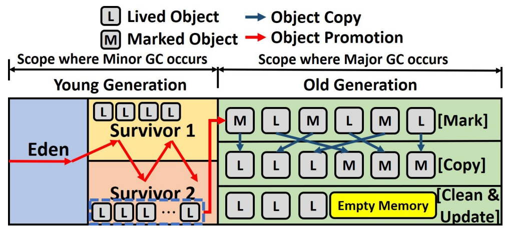


Fig. 2. Garbage Collection Process.

2. Garbage Collection (GC)

In recent years, a proliferation of highly portable and easy-to-program JVM-based frameworks have emerged for handling memory-intensive applications. One of the reasons why JVM applications are widely used in many industrial areas is that users do not need to manually manage the memory lifecycle of objects during runtime. When coding at the application level, garbage collection takes care of memory management required for executing the code. Garbage collection is an automatic process which identifies and frees objects that are no longer in use by the program, thereby freeing up memory resources.

Fig. 2 shows overall garbage collection process. At first, new objects are allocated in *eden* at *young generation*. After sometime, the objects are promoted to survivor area when *eden* is full. If the *survivor 1* is full, minor GC occurs and the live objects are copied to *survivor 2* and the dead objects are cleared. In fact, this process has no significant performance overhead.

Major GC accounts for most part of the performance degradation occurring due to garbage collection because all application threads are temporary paused until major GC is completed. The garbage collection discussed in this paper refers to major GC. Major GC consists of four steps including *mark, copy, cleanup, update reference*.

- **Mark** : In this step, the garbage collector identifies all objects directly accessible from the program and marks them as live objects to collect the garbage.
- **Copy**: All marked live objects are copied to a different part of the heap memory so that they are

Table 1. Specifications of server in DCM

CPU	Intel Xeon Gold 6330, 2.00 GHz 28 Core X 2
Memory	16 GB (DDR4, 3200 MHz) X 8
Network	Mellanox ConnectX-5 100 Gb/s EDR HCA
SSD	Intel NVMe SSD 750 [16] (R/W : 2.2 Gb/s, 0.9 Gb/s)
OS	Linux Kernel 4.18, Centos 8.4

located contiguously in the memory.

- **Cleanup**: After the *copy* step, the *old generation* of the heap memory is cleared out at once.
- **Update Reference**: The garbage collector then updates any internal references in the copied objects to point to the new locations of the referenced objects.

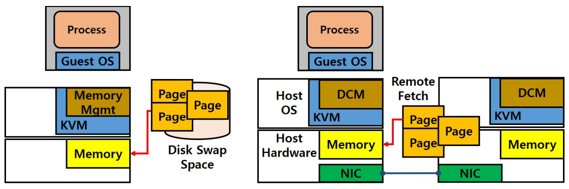
III. EVALUATION

1. Experimental Methodology

1) Configurations: To establish an experimental environment for DCM, we built a client node and donor node connected by 100 Gb/s InfiniBand. The details of experimental specification are shown in Table 1.

2) Workloads: We implemented widely used memory intensive applications, written either in C and Java. For workloads written in C, user manages the memory lifecycle by calling *malloc* and *free* API. Conversely, for workloads written in Java, JVM manages the memory lifecycle [18], thereby allowing application development without explicit concern for memory management. To investigate potential performance differences resulting from memory management by the user or the JVM, we conducted experiments using both unmanaged and managed workloads. The details of the workload can be found in Table 2.

For the managed workload, we utilized the application provided by Intel HiBench benchmark suite [17] that is executed on Apache Spark [9], a notable distributed processing system used for big data workloads. Garbage collection policy was set to use ParallelGC [19].



(a) Disk Swap (b) DCM

Fig. 3. Experimental Environment.

3) Experimental Environment: To ascertain the performance improvement of running applications in DCM in comparison to the virtualized computing environment, it is imperative to conduct a quantitative verification. For this purpose, we designated SSD disk shown in Table 1 as the swap space. The experimental environments for DCM and disk swap are illustrated in Fig. 3.

An application executing in a VM loads necessary working set into memory during runtime. If the requested working set exceeds the local memory, the operating system evicts pages mapped to the client node's memory area into swap space, thus creating free space. Then it subsequently retrieves the pages from the swap space, as depicted in Fig. 3(a). In contrast, DCM has the capability to procure memory pages from the donor node via remote fetching, effectively enlarging the available memory capacity (Fig. 3(b)).

4) Comparison Targets and Evaluation Criteria: To denote the proportion of local memory utilized by working set, we define L(N) as a working set occupying N% of local memory. For instance, in the case of L(50), half of the application's working set is loaded into local memory, while the remaining is mapping to either remote memory or swap space. For the performance metric, we denote makespan that refers to the total time taken to complete a set of tasks. We also categorized three evaluation criteria to represent the impact on DCM: *page fault cost*, *garbage collection*, and *page caching*. These

Table 2. Experimental Workloads

Application	Workload Type	Description
Grep [12]	Unmanaged	Entails searching through a large dataset to identify and extract specific words.
GroupByAggregation [12] (GAG)	Unmanaged	Determines the sum of values corresponding to identical keys within a file.
PageRank [17] (PR)	Managed	Calculates the importance of a webpage based on the number and incoming links.
Bayesian Classification [17] (BC)	Managed	Predicts class membership probabilities belongs to a particular group.

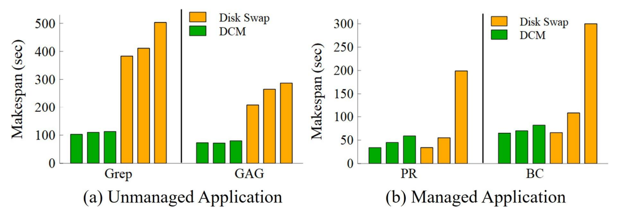


Fig. 4. Makespan of applications per the experimental environments. In the figure, X-axis represents the ratio of local memory to the working set size, listed in the order of L(100), L(80), and L(60).

criteria let us explore deeply the performance implications of each factor in the context of DCM.

2. Makespan

Fig. 4 shows the makespan for each workload against the proportion of working set residing in local memory. Across all workloads, applications running on DCM out-performs those in disk swap. The performance gap widens as the fraction of the working set located in local memory decreases, with DCM exhibiting up to 3.5 times speed-up in execution time. This substantial performance disparity stems from the contrasting I/O performances of DCM and disk swap during the handling of page faults.

Interestingly, the performance variation within DCM is contingent upon whether garbage collection is engaged during application execution. In Fig. 4(a), all workloads running on DCM do not exhibit significant performance variations as local memory ratio to working set is diminished. In Grep, the slowdown at L(60) is approximately 6% compared to L(100).

Contrasting the unmanaged application, the managed application shows divergent performance patterns. It shows a modest degradation in performance as the ratio of working set residing in local memory decreases as shown in Fig. 4(b). This is because the software overhead of marking and copying objects during garbage collection counteracts the benefits of fast remote paging. We have further described in the next section.

3. Influence Factors

In this section, we demonstrate how fast remote paging and its careful victim selection policy in DCM impact over- all performance. For this, we have selected an unmanaged application for our experimental workload.

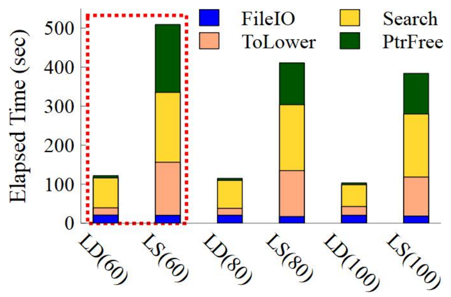


Fig. 5. Breakdown task time analysis in Grep workload. In the figure, LD/LS denotes as local memory ratio in DCM and disk swap, respectively.

The reason for choosing an unmanaged workload was to clearly isolate the overhead associated with page faults, thereby minimizing any potential interference from software overheads such as garbage collection. We selected Grep workload among the workloads because it shows the most distinct performance pattern. To quantitatively compare the page fault overhead between the two environments, we define page fault cost as shown in Eq. (1). In Eq. (1), we denote *PFC, PFL, PFF, N* as page fault cost, page fault latency, page fault frequency and the total number of page fault, respectively. In short, *PFC* is calculated by multiplying the page fault frequency by the total time to get faulted pages from remote memory or disk swap space.

$$PFC = \sum_{i=1}^{N} PFL_i \times PFF_i \qquad (1)$$

1) ***Page Fault Cost****:* Fig. 5 shows a time breakdown for each operation in Grep workload. In all DCM scenarios, *ToLower* and *PtrFree* operations account for less than 10% of the total execution time. Meanwhile, these two operations become significantly more time-consuming, taking up over 45% of the total execution time in disk swap.

This significant performance difference is due to the memory paging that occurs during the processing of these two operations, which quickly depletes local memory and causes frequent page faults. Firstly, *ToLower* operation involves converting uppercase sentences to lowercase, necessitating the loading of a string into memory. If sufficient memory is not available, a page fault occurs, causing the CPU to block while waiting for the faulted page into local memory. Besides, *PtrFree* operation performs memory deallocation, which triggers a page-out process. If the memory page to be freed resides in swap space, a page fault is triggered, thus halting the free operation until the page is relocated to local memory.

As illustrated by the red dotted line in Fig. 5, DCM exhibits negligible performance degradation over L(100) even at L(60). At L(60), the average page fault latency for DCM is 72 μs, while that of a page fault in the disk swap is 1 seconds. The number of page faults is also reduced by about 37% on DCM (20660) compared to disk swap (32659). Calculating this as the page fault cost, DCM took a total of 1.4 seconds, while disk swap took 33.3 seconds, which is about 23 times less overhead. This implies that remote paging and optimized page victim selection policy in DCM handles page fault events more rapidly and reduces the time during which the CPU is blocked, thereby minimizing the performance degradation of the application.

Summary 1

The combination of fast remote paging and selective page eviction policy in DCM leads to a dramatic performance enhancement compared to disk swap, minimizing both the page fault latency and page fault frequency.

***Garbage Collection**:* To investigate the relation between garbage collection and memory paging, we performed the following experiment. We configured the local memory, remote memory, and disk swap space to 16GB. As detailed in Table 2, our experimental workload consisted of java applications from Intel HiBench benchmark suite [17], built on Apache Spark. Apache Spark initiates a JVM process (i.e., executor) with heap memory to execute tasks. Based on this framework configuration, we allocated 32GB as the memory size for the executor, encompassing both local memory and disk swap space. This configuration facilitated the observation of performance patterns in DCM and disk swap, specifically when OS swapping coincides with garbage collection as the working set size surpasses the local memory (16GB).

In a managed application, objects are managed by JVM. Consequently, even if the input size is small, the

Table 3. Working set sizes generated at runtime by different PageRank input data sizes

Input/Working Set Size	Case
0.36 GB/7.97 GB	L(100)
0.72 GB/17.6 GB	L(90)
1.08 GB/19.3 GB	L(80)
1.44 GB/21.3 GB	L(70)

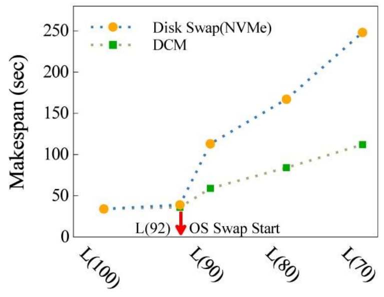


Fig. 6. Makespan of the managed application (PageRank). In the figure, X-axis represents the ratio of local memory to working set, with the ratio decreasing from left to right.

working set size can increase as the application processes due to object creation. In our study, we categorized the working sets into four types, as presented in Table 3. This classification is based on the observation that accessing swap space or remote memory occurs when the working set created at runtime exceeds the local memory capacity.

There is no significant performance difference between DCM and disk swap, where all working sets are loaded into local memory at L(100). We found that the working set at L(92) approximately aligns with the local memory size. Beyond L(92), OS swap was triggered, causing the makespan to increase as shown in Fig. 6. At L(90), where OS swapping and remote paging overlap in earnest, we observed the performance degradation under both DCM and disk swap.

This observation contradicts previous observation suggesting minimal performance degradation even when the working sets are evicted from the local memory in unmanaged applications. This contradiction emerges due to the garbage collection overhead, which dilutes the benefits of fast remote paging. Garbage collection entails marking objects for temporary copying to a different memory space before they are cleared. When the objects marked for cleanup are evicted from the local memory to the backing store, the cleanup operation will come to a halt. It can only resume once the cleanup process retrieves the object back into local memory. As depicted

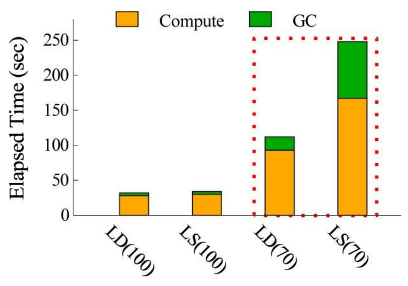


Fig. 7. Breakdown of computation time and GC time as a percentage of the total execution time.

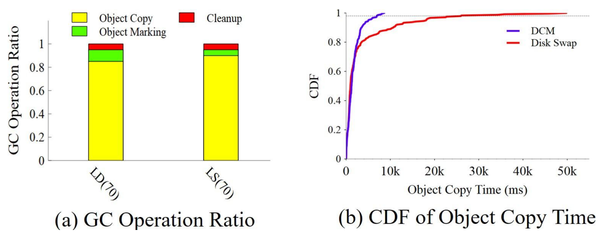


Fig. 8. (a) shows how much time each garbage collection task accounts for in the total garbage collection time; (b) shows a Cumulative Distribution Function (CDF) of object copy times in the L(70) for both cases.

in Fig. 2, all application threads are temporarily halted until the garbage collection process is completed. Consequently, the accumulation of slowdown caused by OS swapping during garbage collection can significantly degrade performance.

Of course, DCM still outperforms disk. As shown in Fig. 7, the overhead is not negligible, with garbage collection time making up 15% of total execution time at L(70). When we examine the garbage collection time in more detail, we observe that object copy time constitutes about 85% in both environments (Fig. 8(a)). It indicates that the eviction of copied marked objects to the backing store and their subsequent retrieval is an dominant operation when page fault occurs.

In order to investigate how both environments impact object copy time when page faults and garbage collection are overlapped, we also represented tail latency using a Cumulative Distribution Function (CDF). As a result, DCM exhibits a smoother tail latency, while the disk swap displays a more extended tail latency spectrum in Fig. 8(b). DCM shows less performance degradation due to its ability to execute page replacements quickly via fast remote paging, thus minimizing the amount of time application threads stop.

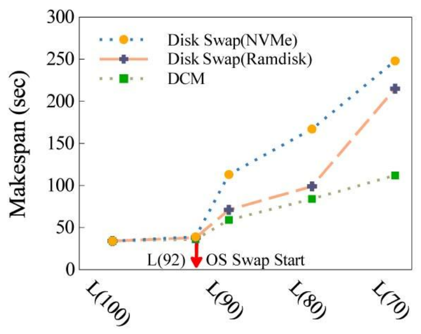


Fig. 9. Makespan of the managed application (PageRank) in DCM and disk swap (NVMe-SSD, Ramdisk).

Summary 2

While running JVM applications on DCM, garbage collection overhead counteracts the efficiency gains achieved by fast remote paging and the selective page eviction policy. However, DCM's fast remote paging substantially reduces the time application threads are halted when page faults and garbage collection are overlapped, thereby yielding superior performance over disk swap.

*3) **Paging Caching**:* Although DCM enables a fast remote paging, it inevitably incurs a not negligible commu- nication overhead with each fault event. DCM mitigates the communication overhead by recognizing memory access patterns and preventing eviction of pages that are likely to be accessed in near future.

In this section, we explore how the selective page eviction impacts performance in DCM. For this, we configured ramdisk as the swap space because it is much faster than SSD. In this context, both ramdisk setting and DCM have similar memory access latencies. The key difference lies in the handling of eviction, where DCM utilizes a policy that reserves eviction through multiple opportunities during page events. Through our experiments, we aim to validate the effectiveness of DCM's optimized page eviction policy.

Fig. 9 compares the makespan between DCM and an environment that sets different types of disks as swap space. It is observed that the performance of disk swap with ramdisk and DCM is similar in L(80). Intriguingly, starting at L(70), ramdisk performance deteriorates relative to DCM. Given that ramdisks generally perform I/O operations at memory speed, we might expect the makespan at DCM as well. However, the performance disparity stems from the fact that more careful page eviction selection in DCM, which provides a sort of caching effect, is more efficient than LRU page replacement approach. The impact of the careful page eviction policy becomes more significant as the working set size increases and more working sets are evicted from local memory.

Summary 3

Optimized page eviction policy in DCM reduces the number of page faults, thus decreasing communication overhead required for fetching pages. This indicates that this page replacement strategy proves to work better in memory-constrained environments than the LRU policy (page replacement policy in kernel).

IV. Conclusion

We conducted a comprehensive study of DCM on both managed/unmanaged applications. Through an empirical analysis, we summarized our observations. (1) DCM shows a minimal 6% performance penalty compared to native memory computing, even when the working set is partially loaded into local memory. (2) Garbage collection has a detrimental effect on the performance of managed applications during page fault occurrences. However, DCM effectively mitigates this performance degradation by employing the combination of fast remote paging and an optimized page eviction policy. (3) Optimized page eviction policy in DCM works better in memory-constrained environment than LRU page replacement algorithm in kernel. We strongly believe that DCM can be effectively utilized as a memory expansion solution to ensure optimal performance of applications in memory-constrained environments. These findings also provide valuable insights and guides for users considering DCM as a memory expansion solution.

Acknowledgment

We thank the anonymous reviewers for their kind comments on this work. This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) (No. NRF-2021R1A2C2014386).

References

[1] A. Eisenman, D. Gardner, I. AbdelRahman, J. Axboe, S. Dong, K. Hazelwood, C. Petersen, A. Cidon, and S. Katti, “Reducing dram footprint with nvm in facebook,” in *Proceedings of the Thirteenth EuroSys Conference*, ser. EuroSys ’18. New York, NY, USA: Association for Computing Machinery, 2018.

[2] D. Gouk, S. Lee, M. Kwon, and M. Jung, “Direct access, High-Performance memory disaggregation with DirectCXL,” in *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. Carlsbad, CA: USENIX Association, Jul. 2022, pp. 287-294.

[3] L. Liu, W. Cao, S. Sahin, Q. Zhang, J. Bae, and Y. Wu, “Memory disaggregation: Research problems and opportunities,” *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*, pp. 1664-1673, 2019.

[4] A. Khan, H. Sim, S. S. Vazhkudai, J. Ma, M.-H. Oh, and Y. Kim, “Persistent memory object storage and indexing for scientific computing,” in *2020 IEEE/ACM Workshop on Memory Centric High Performance Computing (MCHPC)*, 2020, pp. 1-9.

[5] D. Sidler, Z. Wang, M. Chiosa, A. Kulkarni, and G. Alonso, “Strom: Smart remote memory,” in *Proceedings of the Fifteenth European Conference on Computer Systems*, ser. EuroSys ’20. New York, NY, USA: Association for Computing Machinery, 2020. [Online]. Available: https://doi.org/10.1145/3342195.3387519

[6] M. K. Aguilera, N. Amit, I. Calciu, X. Deguillard, J. Gandhi, P. Subrahmanyam, L. Suresh, K. Tati, R. Venkatasubramanian, and M. Wei, “Remote memory in the age of fast networks,” in *Proceedings of the 2017 Symposium on Cloud Computing*, ser. SoCC ’17. New York, NY, USA: Association for Computing Machinery, 2017, p. 121-127.

[7] G. F. Pfister, “An introduction to the infiniband architecture,” *High performance mass storage and parallel I/O*, vol. 42, no. 617-632, p. 102, 2001.

[8] K. Koh, K. Kim, S. Jeon, and J. Huh, “Disaggregated cloud memory with elastic block

management," *IEEE Transactions on Computers*, vol. 68, no. 1, pp. 39-52, 2019.

[9] M. Zaharia, R. S. Xin, P. Wendell, T. Das, M. Armbrust, A. Dave, X. Meng, J. Rosen, S. Venkataraman, M. J. Franklin *et al.*, "Apache spark: a unified engine for big data processing," *Communications of the ACM*, vol. 59, no. 11, pp. 56-65, 2016.

[10] M. Krumnikl, P. Bainar, J. Kl´ımova´, J. Kozˇusznik, P. Moravec, Svatonˇ, and P. Tomancˇa´k, "Scijava interface for parallel execution in the imagej ecosystem," in *Computer Information Systems and Industrial Management: 17th International Conference, CISIM 2018, Olomouc, Czech Republic, September 27-29, 2018, Proceedings 17*. Springer, 2018, pp. 288-299.

[11] B. Schmid, J. Schindelin, A. Cardona, M. Longair, and M. Heisenberg, "A high-level 3d visualization api for java and imagej," *BMC bioinformatics*, vol. 11, no. 1, pp. 1-7, 2010.

[12] J. Lee, Y. Jung, S. Lee, S. Jamil, S. Park, K. Koh, H. Kim, K. Kim, and Y. Kim, "Mfence: Defending against memory access interference in a disaggregated cloud memory platform," 2023.

[13] H. Al Maruf and M. Chowdhury, "Effectively prefetching remote memory with leap," in *Proceedings of the 2020 USENIX Conference on Usenix Annual Technical Conference*, ser. USENIX ATC'20. USA: USENIX Association, 2020.

[14] Z. Guo, Y. Shan, X. Luo, Y. Huang, and Y. Zhang, "Clio: A hardware-software co-designed dis-aggregated memory system," ser. ASPLOS '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 417-433.

[15] M. Hussain, "Need for speed : Comparing fdr and edr infiniband (part 2)," 2018.

[16] "Intel® SSD 750 Series Product Specification," Intel Coporation, Tech. Rep., 2015. [Online]. Available: https://www.intel.com/content/dam/www/public/us/en/documents/product-specifications/ssd-750-spec.pdf

[17] S. Huang, J. Huang, J. Dai, T. Xie, and B. Huang, "The hibench benchmark suite: Characterization of the mapreduce-based data analysis," in *2010 IEEE 26th International conference on data engineering workshops (ICDEW 2010)*. IEEE, 2010, pp. 41-51.

[18] T. Domani, E. K. Kolodner, E. Lewis, E. E. Salant, K. Barabash, I. Lahan, Y. Levanoni, E. Petrank, and I. Yanorer, "Implementing an on-the-fly garbage collector for java," in *Proceedings of the 2nd International Symposium on Memory Management*, ser. ISMM '00. New York, NY, USA: Association for Computing Machinery, 2000, p. 155-166.

[19] H.-J. Boehm, A. J. Demers, and S. Shenker, "Mostly parallel garbage collection," *ACM SIGPLAN Notices*, vol. 26, no. 6, pp. 157-164, 1991

Yeonwoo Jeong is currently pursuing Ph.D. degrees in computer science and engineering from Sogang University, Seoul, Republic of Korea. He received his B.S. degree in computer software from Kwangwoon University and M.S. degree in computer science and engineering in Sogang University. His research interests include cloud computing, streaming system, and resource scheduling.

Gyeonghwan Jung is currently pursuing Master degree in computer science and engineering from Sogang University, Seoul, Republic of Korea. He received B.S. degree in computer science from Sangmyung University. His research interests include cloud computing and resource management.

Kyuri Park is currently pursuing Master degrees in computer science and engineering from Sogang University, Seoul, Republic of Korea. She received her B.S. degree in computer science and engineering from Sogang University. Her research interests include cloud computing, streaming system and resource management.

Youngjae Kim (Member, IEEE) received the BS degree in computer science from Sogang University, South Korea in 2001, the MS degree in computer science from KAIST in 2003, and the PhD degree in computer science and engineering from Pennsylvania State University, University Park, Pennsylvania in 2009. He is currently an associate professor with the Department of Computer Science and Engineering, Sogang University, Seoul, South Korea. Before joining Sogang University, Seoul, South Korea, he was a R\&D staff scientist at the US Department of Energy's Oak Ridge National Laboratory (2009–2015) and as an assistant professor at Ajou University, Suwon, South Korea (2015–2016). His research interests include operating systems, file and storage systems, database systems, parallel and distributed systems, and computer systems security.

Sungyong Park is a professor in the Department of Computer Science and Engineering at Sogang University, Seoul, Korea. He received his B.S. degree in computer science from Sogang University, and both the M.S. and Ph.D. degrees in computer science from Syracuse University. From 1987 to 1992, he worked for LG Electronics, Korea, as a research engineer. From 1998 to 1999, he was a research scientist at Telcordia Technologies (formerly Bellcore), where he developed network management software for optical switches. His research interests include cloud computing and systems, virtualization technologies, high performance I/O and storage systems, and embedded system software.