End Performance SLA Support for Disaggregated Memory

Kwangwon Koh
ETRI
kwangwon.koh@etri.re.kr

Kangho Kim
ETRI
khk@etri.re.kr

Changdae Kim
ETRI
cdkim@etri.re.kr

Jaehyuk Huh
KAIST
jhhuh@kaist.ac.kr

Abstract

This paper proposes a new service-level agreement (SLA) support for disaggregated memory, which consists of fast direct DRAM and slow indirect memory in remote machines or local non-volatile memory. Based on a new performance prediction model, the SLA support provides the bounded performance as compared to the ideal configuration only with the direct memory. For the SLA fulfillment, we propose an efficient performance predictor using indirect memory traces, and a memory management system to adjust the local memory capacity as necessary. We implement the proposed SLA support in a disaggregated memory system integrated to Linux KVM, and show its feasibility with real machine experiments. By providing the performance bound, the proposed SLA support facilitates the adoption of disaggregated memory in cloud computing, where the performance consistency is critical.

Keywords Disaggregated Memory, Service-Level Agreement, Cloud Computing, Memory Management

1. Introduction

Disaggregated memory provides the memory capacity expansion from the direct local DRAM to indirect memory either in remote machines or local non-volatile memory attached via PCIe. It has emerged to accommodate the capacity demands arising from the memory-intensive workloads, such as in-memory databases, in-memory data caching, bioinformatics, and graph processing. In addition, disaggregated memory can provide a cost-effective way to scale memory capacity while improving resource provisioning flexibility and power efficiency for cloud providers [12, 14–16]. Numerous prior studies have investigated the memory extension across node boundaries. Some of them investigated remote paging techniques on OS or hypervisor, and others proposed the hardware architectures of the memory blade [1, 2, 4, 5, 7, 8, 11, 13–18].

However, disaggregated memory can pose a new challenge to cloud providers. Depending on the capacity allocation of fast direct and slow indirect memory, the application performance can significantly vary. Such performance inconsistency can potentially hamper the adoption of disaggregated memory in cloud systems.

To facilitate the utilization of disaggregated memory systems on clouds, this study proposes a new SLA support for disaggregated memory system guaranteeing the end performance based on a performance model (§2). To the best of our knowledge, this is the first study to support quality-of-service (QoS) for disaggregated memory systems on clouds. To support the SLA, this study proposes an efficient performance predictor by profiling accesses to the tail part of direct local memory and accesses to indirect memory. The proposed model can estimate local memory misses using the extension of *Counter Stack* [22] (§3). With the new SLA support manager on a disaggregated memory system integrated to Linux KVM, this paper shows its feasibility by real machine experiments (§4).

2. SLA for Disaggregated Memory

To support the bounded performance with disaggregated memory on clouds, SLA must be changed to include the effect of memory allocation of direct and indirect memory. Even with memory heterogeneity, the system must provide a certain level of performance consistency.

We define SLA-α as an agreement supporting the performance degradation within $(1 - \alpha)$ fraction of performance of a virtual machine (VM) with the contracted resources. For instance, when a customer contracts to use 128 GB of DRAM memory with SLA-0.9, the cloud provider can use any combination of direct and indirect memory capacities, as long as it can provide the performance within a performance degradation of up to 10% from the direct memory-only performance. If the cloud system is unable to allocate the required capacity of direct memory to meet SLA-α, the VM must be migrated to a physical node that has sufficient memory capacity. To support such a new SLA, a performance prediction model is required to identify the performance degradation due to the involvement of indirect memory.

2.1 Performance Prediction Model

The execution time is delayed by the total latency of indirect memory accesses (direct memory misses), and thus, we define the execution time as follows:

$$T(n) = L + \Sigma P, \quad (1)$$

where $T(n)$ is the execution time of an application running on a VM on the disaggregated memory system. L is its execution time on an ideal VM using only the direct memory for the entire VM memory. P is a latency for handling a direct memory miss. The latency penalty ΣP determines the total performance penalty of applications running on the disaggregated memory.

Let α be the target performance bound for SLA ($0 < \alpha \leq 1$). For a given α, the allowed number of direct memory misses, n_{SLA}, is defined as follows:

$$n_{SLA} = \frac{(1/\alpha - 1)L}{P_{avg}} (\because \Sigma P = P_{avg} \times n), \quad (2)$$

where P_{avg} is the average value for the entire Ps, and n is the number of direct memory misses. To meet SLA, n must become equal to or less than n_{SLA}. Otherwise, the SLA is violated. In such a situation of SLA violation, the direct memory must be additionally allocated to the VM, and the minimum required capacity of direct memory (Δc) is calculated as follows:

$$\Delta c = H^{-1}(n \rightarrow n_{SLA}),$$

where H^{-1} is the inverse function of a reuse distance histogram which estimates the reduced number of direct memory misses for the direct memory capacity.

SLA fulfillment With the known values of variables, $T(n)$ and ΣP, the ideal execution time L can be estimated from equation 1 *without knowledge of the workload*. When n is larger than n_{SLA} from equation 2 with the estimated L, the additional required direct memory capacity (Δc) must be calculated to identify how much more direct memory is necessary for fulfilling the SLA-α. The key prediction model to obtain Δc is H^{-1} which is based on the profiling of reuse distance. However, a naïve reuse distance profiling incurs a significant amount of cost since it requires an analysis of full memory traces. In the next section, an efficient reuse distance profiling for the disaggregated memory is presented.

3. Reuse Distance Profiling

A *reuse distance* is the number of distinct memory pages accessed between two memory accesses to the same memory page. If the direct memory capacity is larger than the reuse distance, the latter memory access between the two accesses will be a hit on the direct memory. Therefore, by using the reuse distance histogram of a memory trace, we can estimate the number of direct memory misses with a given capacity for the trace.

To profile the reuse distances online, there are two challenges. First, tracing all accesses to memory becomes a significant burden on the system. To reduce the overhead while focusing on the SLA, we propose to use `partial memory traces`, which do not record

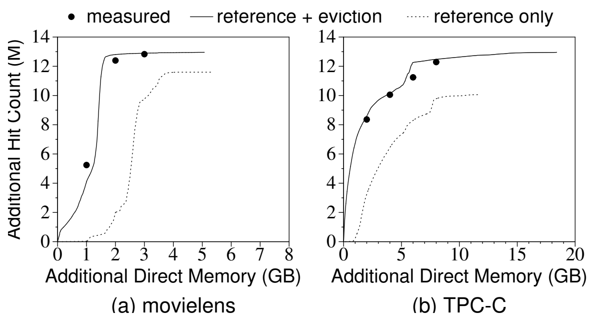

Figure 1: Additional direct memory hits with increasing direct memory sizes. Two curves are estimated with page reuse histograms with `reference-only` and `reference+eviction` traces. Dots denote the measured ones with a given size (`measured`).

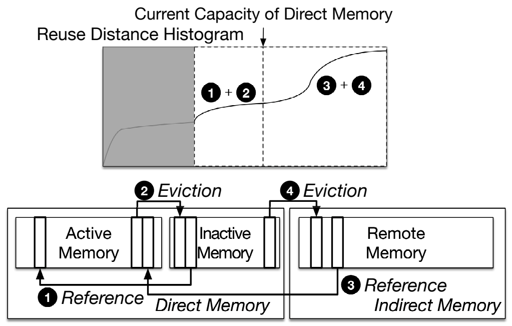

Figure 2: Partial memory profiling and reuse histogram

accesses to the pages in the hot memory region (§3.1). Second, analyzing the profiled accesses should have acceptable overheads in terms of time and space costs. Among possible approaches on memory access profiling [9], this paper proposes `Dual Counter Stack` by extending the prior *Counter Stack* technique [22] (§3.2).

3.1 Partial Memory Traces

Since tracking every access to memory is not feasible, sampling techniques have been proposed for the reuse distance profiling. *StatCache* [3] samples every N-th access, and *RapidMRC* [20] samples a small portion of time period. However, such temporal sampling is not appropriate for DRAM and remote memory profiling since it requires profiling of very long reuse distances. *SHARDS* [21] proposed a spatial sampling mechanism for I/O accesses. However, it requires all accesses trapped to determine whether each access needs to be sampled, which incurs significant overheads when applied to memory tracing.

For reducing memory tracking cost, we adopt *partial memory traces* introduced by *Geiger* [9], and we also revise a *reuse distance* as *the number of pages evicted to and remained in remote memory between eviction and subsequent reference to the same page*.

access:	a	b	c	**a**	**b**
from a	1	2	3	**3**	3
from b		1	2	3	**3**
from c			1	2	3
from **a**				1	2
from **b**					1

Figure 3: *HyperLogLog* counters example

Figure 1 shows the increased direct memory hit counts when additional direct memory capacity is added. Two curves are estimated from the reuse histograms with `reference-only` and `reference+eviction`. Dots show the increased hits from the real runs with the given additional memory capacity. The results show that the estimation corrected with eviction traces is more accurate than the estimation with only references.

Figure 2 presents the overview of monitoring system with *partial memory profiling*. We manage the memory pages in three regions. First, `active memory` consists of hot data, and any referenced page is moved to the region (❶ and ❸). Second, `inactive memory` contains warm data, evicted pages from the active memory (❷). Third, `remote memory` covers the pages evicted from `inactive memory` (❹) and located in remote memory. Except for `active memory`, we capture the memory accesses by unmapping, and collect reference and eviction traces in the page fault handler. Profiling between the direct and remote memory, obtained from ❸ and ❹, is used to estimate whether the current performance is within the SLA on the disaggregated memory. In addition, the reuse histogram between the active and inactive memory, obtained from ❶ and ❷, can be used to identify how much direct memory is over-provisioned after satisfying the SLA.

3.2 Counter Stack for Partial Memory Traces

Counter Stack is an efficient reuse distance estimation technique using full access traces [22]. In this section, we briefly describe *Counter Stack* and our extension for supporting `partial memory traces`.

Counter Stack with Full Memory Traces: For every access, *Counter Stack* creates a counter using *HyperLogLog* which estimates the number of distinct items based on the stochastic averaging of hashed values of accesses. The consecutive accesses are added to all *HyperLogLog* counters, to estimate the number of distinct accesses since each counter has been created.

Figure 3 presents an example of *Counter Stack*. If a reuse exists in a given memory access, the counter does not increase horizontally. For the non-increasing counters, vertically non-increasing counters in a column indicate reuses and their distances marked in the bold numbers in the figure. Then, *Counter Stack* detects reuse, and the counter value indicates the estimated reuse distance.

Table 1: Workloads: performance and memory footprint

Workload	local DRAM-only Perf.	Memory Footprint
movielens	1,170.485s	25,607MB
twitter	287.086s	16,637MB
snap	375.217s	69,525MB
TPC-C	42796.1TPS	26,556MB

Since the number of counters determines the computation and memory overheads, *Counter Stack* adopts two optimizations. First, it creates a counter for a group of accesses instead of every access. This reduces the resolution of reuse distances, but the number of counters are also reduced as the size of the group increases. Second, it drops old counters. This restricts the distance of detectable reuse, while *Counter Stack* can focus on the reasonable range of reuse distances.

Dual Counter Stack for Partial Memory Traces: To support `partial memory traces` with reference and eviction traces, we present `Dual Counter Stack` by extending *Counter Stack*. It creates two counters for every remote memory accesses: $\text{Counter}_{R\&E}$ takes both references and evictions. The counter is used to detect reuses, because a new reference to the recently evicted address does not increase this counter. Counter_{RO} takes references only. Since the reuse distance is the number of evicted but not referenced pages, as described in §3.1, it can be estimated by ($\text{Counter}_{R\&E}$ - Counter_{RO})

However, when a reference and eviction on a page are counted, the order of accesses matters. Assume that a page is referenced and then evicted between a reuse. This contributes to the reuse distance by its definition. On the other hand, if the page is evicted and then referenced, it does not contribute to the reuse distance since it is finally referenced. However, the counting based mechanism cannot distinguish two cases. To address this problem, we add a reference id for each page, increasing the id for every eviction. This makes the reuse distance estimation correct by distinguishing the cases.

4. Evaluation

4.1 Implementation

We implement the proposed SLA support manager on a KVM-based disaggregated memory system, *dcm* [12]. The implementation is based on Linux kernel 4.10.12. We use the LRU-chain management of *dcm* to manage three memory regions, and *dcm* traps accesses on `inactive` and `remote memory` regions. The memory profiling with *HyperLogLog* is executed during the page fault handler, and estimating the reuse distance histogram is periodically executed every 5 seconds as a user process. The SLA manager also notifies the kernel to expand or shrink the direct memory capacity for each VM based on the estimation results.

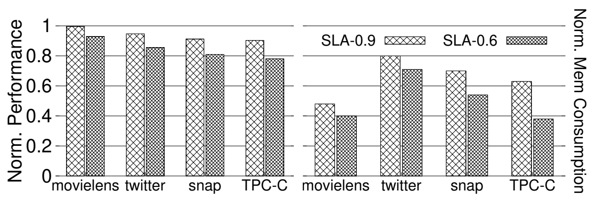

Figure 4: Performance/direct memory consumption normalized to local DRAM-only VM

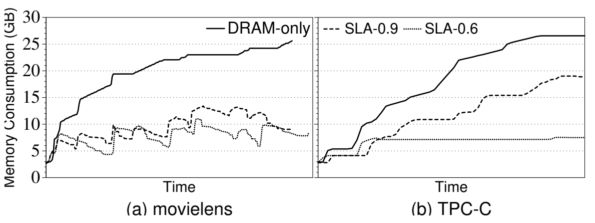

Figure 5: Direct memory consumption over time

4.2 Experimental Methodology

For evaluation, we use a system with two Intel Xeon E5-2670v3@2.3GHz, DDR4 96 GB memory. For a remote memory, we equip a PCRAM-based Intel Optane SSD with 375GB capacity. A VM is configured to have 24 vCPUs and 160 GB memory. The baseline experiments use a vanilla KVM configured to allow memory overcommits. For evaluating the SLA manager, the proper local DRAM capacity is determined by the SLA manager, and then the capacity is adjusted in *dcm* [12].

Table 1 lists the workloads and their characteristics. For the performance metric, we use throughput per second (TPS) for TPC-C and execution time for the rest. The memory footprint is measured using Resident Set Size (RSS). We chose four workloads widely used in the in-memory computing on clouds: collaborative-filtering (movielens), graph processing (twitter), in-memory database (TPC-C on VoltDB), and gene sequence alignment (snap) [6, 19, 23].

4.3 Experimental Results

The primary goal of the proposed SLA manager is to fulfill SLA-α and the secondary goal is to reduce the capacity of the direct memory as much as possible, while SLA-α is retained.

Figure 4(a) shows the performance results of our SLA manager. The x-axis indicates target workloads and the y-axis represents the normalized performance to the ideal VM, which can contain the entire memory footprint in the local DRAM (local DRAM-only VM). All results show that they achieve the target performance. Figure 4(b) shows the direct memory consumption normalized to the local DRAM-only VM. Under SLA-0.9, *movielens* shows up to 99.6% performance of the local DRAM-only VM with 47.8% of the direct memory capacity.

Figure 5 presents the direct memory consumption during the execution for two workloads. In Figure 5(a), allocation and reclamation of the direct memory are repeated to accommodate the memory demand changes of *movielens*. Figure 5(b) shows that a much smaller direct memory capacity is enough to meet the SLA for *TPC-C*. Infrequently, the memory consumption with SLA-0.6 is larger than that with SLA-0.9since our SLA manager aggressively enlarges and conservatively shrinks the direct memory capacity. However, the overall consumption with SLA-0.6 is much smaller than that with SLA-0.9.

Finally, our SLA manager shows negligible time and space costs. The *HyperLogLog*-based profiling adds 78ns latency to the page fault handler on average. Note that the latency is added infrequently, with only accesses to tail parts of direct memory and indirect memory. In addition, `Dual Counter Stack` takes 4.7ms for every 5s time interval. Its CPU utilization is about 0.004% running as a background process. The memory footprint by the SLA manager is 5.2-5.8MB for the workloads.

5. Conclusion and Future Work

This paper defined a new SLA for disaggregated memory systems and proposed an SLA support manager for a single system. We are exploring the following directions as our future work.

Multi-tiered memory: Our framework needs to be extended to support multi-tiered memory systems. A system can have both of the network-attached remote memory and PCIe-attached non-volatile memory (NVM). Moreover, network latencies and bandwidth can vary by the network topology, creating diverse remote memory access costs.

Compensation for temporary SLA violation: Although the proposed technique can support the overall performance guarantee, temporary SLA violation can occur for a short period of time by sudden VM behavior changes. The violation can be quickly compensated at the next time period, by over-provisioning memory capacity temporally.

Sequential access: Disaggregated memory system needs to be further optimized for sequential access patterns which incur remote memory accesses for consecutive addresses. Prefetching or *Elastic Block Management* [12] can be employed to address such patterns.

Memory Scheduling Policy: With the proposed performance model and dynamic memory allocator, memory becomes a predictable and flexible resource. Thus, memory capacity becomes a schedulable resource, and various scheduling policies are possible, such as limiting the performance variance, maximizing system-wide throughput, and so on [10].

Revisiting VM placement: A VM placement policy should be revisited as our work provides more scheduling options. As a VM does not have to fully allocate its memory within the system, more VMs can be consolidated into a system without SLA violation.

Acknowledgments

This work was supported by Ministry of Science and ICT (MSIT), Korea through the R&D program of NST (grant B551179-12-04-00) and ETRI R&D program (grant 19ZS1220). Jaehyuk Huh was supported by IITP (No.2017-0-00466) funded by MSIT, Korea.

References

[1] 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 Proc. Symposium on Cloud Computing (SoCC), pages 121–127, 2017.

[2] E. A. Anderson and J. M. Neefe. An Exploration of Network RAM. Technical report, 1998.

[3] E. Berg and E. Hagersten. StatCache: A Probabilistic Approach to Efficient and Accurate Data Locality Analysis. In Proc. International Symposium on Performance Analysis of Systems and Software (ISPASS), pages 20–27, March 2004.

[4] H. Chen, Y. Luo, X. Wang, B. Zhang, Y. Sun, and Z. Wang. A Transparent Remote Paging Model for Virtual Machines. In Proc. International Workshop on Virtualization Technology (IWVT), 2008.

[5] S. Dwarkadas, N. Hardavellas, L. Kontothanassis, R. Nikhil, and R. Stets. Cashmere-VLM: Remote Memory Paging for Software Distributed Shared Memory. In Proc. International Parallel Processing Symposium (IPPS), pages 153–159, Apr 1999.

[6] M. Ferdman, A. Adileh, O. Kocberber, S. Volos, M. Alisafaee, D. Jevdjic, C. Kaynak, A. Daniel Popescu, A. Ailamaki, and B. Falsafi. Clearing the Clouds: A Study of Emerging Scale-out Workloads on Modern Hardware. In Proc. International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), pages 37–48, 2012.

[7] M. D. Flouris and E. P. Markatos. The Network RamDisk: Using Remote Memory on Heterogeneous NOWs. Cluster Computing, 2(4):281–293, Oct 1999.

[8] J. Gu, Y. Lee, Y. Zhang, M. Chowdhury, and K. G. Shin. Efficient Memory Disaggregation with INFINISWAP. In Proc. USENIX Conference on Networked Systems Design and Implementation (NSDI), pages 649–667, 2017.

[9] S. T. Jones, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Geiger: Monitoring the Buffer Cache in a Virtual Machine Environment. In Proc. International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), pages 14–24, 2006.

[10] C. Kim and J. Huh. Exploring the design space of fair scheduling supports for asymmetric multicore systems. IEEE Transactions on Computers, 67(8):1136–1152, Aug 2018.

[11] K. Kim, J. Kim, and S. Jung. GNBD/VIA: A Network Block Device over Virtual Interface Architecture on Linux. In Proc. International Parallel and Distributed Processing Symposium (IPDPS), pages 1–7, 2002.

[12] K. Koh, K. Kim, S. Jeon, and J. Huh. Disaggregated Cloud Memory with Elastic Block Management. IEEE Transactions on Computers, 68(1):39–52, Jan 2019.

[13] S. Liang, R. Noronha, and D. K. Panda. Swapping to Remote Memory over Infiniband: An Approach using a High Performance Network Block Device. In Proc. International Conference on Cluster Computing (CLUSTER), pages 1–10, Sept 2005.

[14] K. Lim, J. Chang, T. Mudge, P. Ranganathan, S. K. Reinhardt, and T. F. Wenisch. Disaggregated Memory for Expansion and Sharing in Blade Servers. In Proc. International Symposium on Computer Architecture (ISCA), pages 267–278, 2009.

[15] K. Lim, Y. Turner, J. Chang, J. R. Santos, and P. Ranganathan. Disaggregated Memory Benefits for Server Consolidation. HP Laboratories, Palo Alto, CA, USA, 2011.

[16] K. Lim, Y. Turner, J. R. Santos, A. AuYoung, J. Chang, P. Ranganathan, and T. F. Wenisch. System-level Implications of Disaggregated Memory. In Proc. International Symposium on High-Performance Computer Architecture (HPCA), pages 1–12, 2012.

[17] E. P. Markatos and G. Dramitinos. Implementation of a Reliable Remote Memory Pager. In Proc. USENIX Annual Technical Conference (ATC), pages 15–15, 1996.

[18] T. Newhall, S. Finney, K. Ganchev, and M. Spiegel. Nswap: A Network Swapping Module for Linux Clusters. In Proc. European Conference on Parallel Processing (Euro-Par), pages 1160–1169, 2003.

[19] M. Stonebraker and A. Weisberg. The VoltDB Main Memory DBMS. IEEE Data Engineering Bulletin, 36(2):21–27, 2013.

[20] D. K. Tam, R. Azimi, L. B. Soares, and M. Stumm. RapidMRC: Approximating L2 Miss Rate Curves on Commodity Systems for Online Optimizations. In Proc. International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), pages 121–132, 2009.

[21] C. A. Waldspurger, N. Park, A. Garthwaite, and I. Ahmad. Efficient MRC Construction with SHARDS. In Proc. USENIX Conference on File and Storage Technologies (FAST), pages 95–110, 2015.

[22] J. Wires, S. Ingram, Z. Drudi, N. J. A. Harvey, and A. Warfield. Characterizing Storage Workloads with Counter Stacks. In Proc. USENIX Conference on Operating Systems Design and Implementation (OSDI), pages 335–349, 2014.

[23] M. Zaharia, W. J. Bolosky, K. Curtis, A. Fox, D. A. Patterson, S. Shenker, I. Stoica, R. M. Karp, and T. Sittler. Faster and More Accurate Sequence Alignment with SNAP. arXiv:1111.5572, 2011.