# Enforcing Fairness in Disaggregated Non-Volatile Memory Systems

Vamsee Reddy Kommareddy[1], Amro Awad[1], Clayton Hughes[2], and Simon David Hammond[2]

[1] University of Central FLorida, Orlando FL 32816, USA
vamseereddy8@Knights.ucf.edu, amro.awad@ucf.edu
[2] Sandia National Laboratories, Albuquerque, NM 87185, USA
chughes,sdhammo@sandia.gov

**Abstract.** As many applications have growing demands for memory, the memory system is expected to become the bottleneck of next-generation computing systems. Sharing memory system across processor sockets and nodes becomes a compelling trend because memory is scaling at a slower rate than processor technology. Moreover, as many applications rely on shared huge data, e.g., graph applications and database workloads, having a large number of nodes accessing shared memory allows for efficient use of resources and avoids duplicating huge files, which can be infeasible for large graphs or scientific data.
Upgrading memory modules and maintenance become a challenging task when dealing with thousands of computing nodes. Thus, placing all memory modules in a centralized location, i.e., memory blade, instead of coupling them with specific compute nodes, can enable more flexibility in upgrading memory and maintenance. Hence, disaggregated memory systems are more suitable for large scale systems that get upgraded frequently. However, due to the nature of disaggregated memory systems, different users and applications compete for the available shared memory bandwidth, and therefore can result in severe contention due to memory traffic from different SoCs. In this paper, we discuss the contention problem in disaggregated memory systems and suggest mechanisms to ensure memory fairness and enforce QoS. Our simulation results show that employing our proposed QoS techniques can speed up memory response time by up to 55%.

**Keywords:** Disaggregated memory systems · Non-volatile memory · Quality of service.

## 1 Introduction

The memory system is becoming a major bottleneck in conventional High-Performance Computing (HPC) systems. HPC systems are usually designed to have per-node memory that can accommodate the largest memory footprint of the expected set of applications. Unfortunately, due to the diversity of applications run on HPC systems, especially with the emergence of cloud computing, the

memory requirements can vary a lot. This can lead to situations where the memory subsystem is underutilized. Perhaps more importantly, because the memory subsystem is typically constructed with DRAM, each node can incur very high cooling costs in addition to significant power consumption [27, 24, 10, 11, 22, 25, 23, 32]. Additionally, the emergence of workloads that process huge shared files or large graphs makes private memory for a node less attractive. These workloads are expected to become more common in the future, [6, 18], pushing future computing systems to become *memory-centric*.

In memory-centric systems, a memory blade contains all global/shared memory across the nodes and can be accessed by any of the computing nodes. Compute nodes can be as simple as System-on-Chip (SoC) nodes. Such systems are also typically referred to as *disaggregated memory systems* (DMS) [26]. Each node is connected to the shared memory through a high speed interconnect fabric, e.g., GenZ or CCIX [9, 8]. An example of such a system is The Machine project from HP Labs [7]. These memory-centric systems promise scalable shared memory applications and significant reductions of communication overhead by relying on shared memory instead of ethernet interfaces to facilitate message passing between compute nodes. Moreover, applications that access shared large files or data-sets concurrently can benefit from having these files resident in the globally accessible shared memory. Such a trend becomes more compelling with emerging Non-Volatile Memories (NVMs) [5, 20] which promise terabyte capacities per memory module, and can be leveraged to keep shared files and large data-sets persistent with ultra-low idle power. And, unlike DRAM, NVMs do not require frequent refresh operations and can retain their data even after power loss.

Unfortunately, as DMS are expected to be used in multi-tenant environments, e.g., cloud systems or data centers, contention can become a significant problem due to competition for the shared global (centralized) memory. As the number of compute nodes that share the global memory increases, the more likely the average global memory access time will increase. Additionally, the worst-case access latency becomes much higher and mainly depends on the access patterns of other nodes and their memory intensity. This potential slowdown can certainly affect the adoption of such systems in environments where users and applications are guaranteed some level of quality assurance through Service-Level Agreements (SLAs), such as in cloud systems. To this end, ensuring *Quality-of-Service (QoS)*, is essential for designing and using DMS.

In this paper, we investigate the impact of QoS on application performance when running on DMS. Specifically, we propose a hierarchical dynamic priority-based approach to support QoS in disaggregated NVM systems. Two levels of priorities are maintained - static and dynamic. Static priority is fixed at run-time. Dynamic priority is adjusted over the lifetime of the application. We divide the shared memory into memory pools to improve performance and study the effect of our approach. To the best of our knowledge, our work is the first to investigate QoS on DMS in addition to investigating novel solutions for this purpose.

The rest of the paper is organized as follows: Section 2 discusses the background on QoS and DMS. Section 3 details the design and approach of our

hypothesis. Evaluation and results are discussed in Section 4. Finally, we conclude in Section 6.

## 2 Background, Related Work and Motivation

In this section, we briefly explain the most relevant concepts to our paper, e.g., DMS and QoS. Later, we present motivational results that demonstrate the contention issue on memory-centric systems and the importance of QoS in such systems.

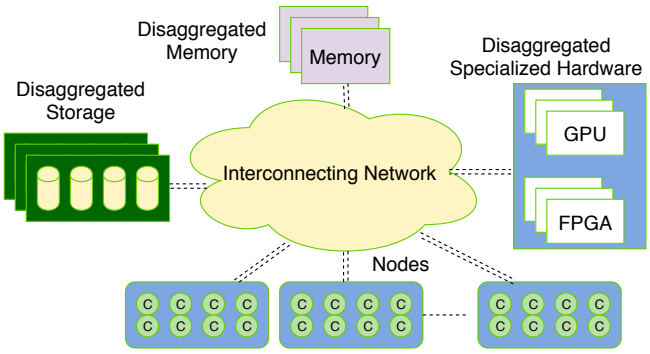


**Fig. 1.** Disaggregated Memory System

### 2.1 Disaggregated Memory Systems (DMS)

As memory-driven computing becomes the leading trend when designing high-performance computing (HPC) and high-performance data analytics (HPDA) systems, disaggregated memory is considered one of the most promising architectures to enable efficient sharing and concurrent access to huge data [21, 7]. Moreover, the emergence of non-volatile, dense and fast memory technologies, e.g., 3D Xpoint [14], makes building large shared memory systems a more appealing option when hosting large shared memory-mapped files and data. Additionally, upgrading memory and maintenance is much easier(by replacing memory blades[21]) with such systems than manually upgrading hundreds of thousands of nodes. Finally, DMS allow seamless integration of new compute nodes by just connecting them to the high-speed interconnect and allowing them to share data with the help of conventional load/store operations instead of using costly OpenMPI framework calls and access large memory capacities. Recent studies demonstrate that approximately 80% of the jobs on HPC systems overestimate their memory prerequisites [1]. By committing memory to particular jobs, HPC frameworks often under-utilize the available memory. With DMS, each node can request as much memory as it needs from the shared space while the rest can be utilized by other nodes.

However, a lot of challenges need to be addressed while dealing with DMS, which opens a path for designing future computing systems. One of them is, even though DMS can provide better bandwidth by scaling the number of channels to

shared memory, at the expense of speed, fair allocation of memory bandwidth to all the applications should be handled in heterogeneous systems to provide Quality of Service (QoS).

### 2.2 Quality of Service

The more compute units deployed in a system, the more contention there will be in the memory subsystem(shared resource). Contention on memory banks and shared request queues become more aggressive as the number of compute nodes accessing the shared (global) memory increases. However, many modern applications are memory-driven; computing components accessing the same memory units are more common. Different memory scheduling schemes are implemented in memory controllers to enforce QoS. Zhou et al. [34] implemented a fine-grained QoS scheduling for PCM memory using pre-emption methodologies at the cost of performance. Subramanian et al. [30] worked on designing a model to accurately estimate memory-interference-induced slowdowns. They also proposed a memory scheduler that meets hardware accelerator deadlines along with maximizing CPU performance [31]. Jeong et al. [16] proposed QoS aware memory controller which can dynamically balance bandwidth between CPUs and GPUs. Zhao et al. [33] proposed a memory control scheme called FIRM which can fairly run persistent and non-persistent applications. While all the above-mentioned schemes works for the respective targeted systems, they need to be a carefully studied for DMS as the contention at the shared memory controller is huge in such systems. We explore possible solutions that can enhance QoS and study the possible outcomes to improve the performance of disaggregated non-volatile memory system.

### 2.3 Motivation

Since shared memory in DMS is accessed by multiple nodes, the contention at the shared memory is expected to be high and could cause significant slowdowns. Moreover, as DMS use a remote memory, there is an incremental delay in accessing memory due to the network. Figure 2 shows the delay in accessing memory per request in disaggregated memory system while varying the number of nodes in the system[3]. As expected, there is a significant increase on the average memory response time when increasing the number of nodes running on a disaggregated memory system. Such slowdowns mainly depend on the memory-intensity of the applications running on the compute nodes and the sensitivity of the applications to memory latency. Surprisingly, we observe that such contentions can lead to multiple times increase in global memory access latency. Pennant, due to its memory intensity and streaming access pattern, incurs more than 3 times increase in access latency when there are 4 nodes sharing the same memory; from $380ns$ up to $1400ns$. Note that row buffer locality can be severely impacted and the chances of finding row buffer hits would decrease with receiving many memory requests from other nodes. In this paper, we explore novel hierarchical, static and dynamic priority schemes as QoS assurance mechanisms on DMS.

[3] More details about the methodology and benchmarks are explained in section 4 in detail.

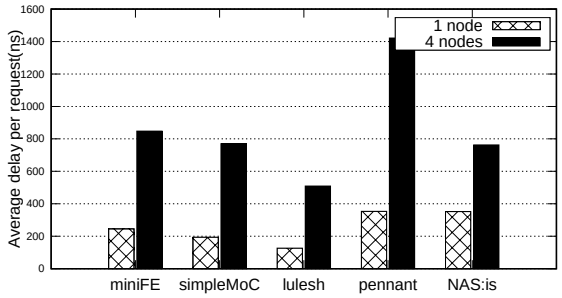


**Fig. 2.** Delay per request in accessing memory for generic and disaggregated memory systems

## 3 Design

In this section, we discuss our proposed QoS support for disaggregated memory systems. Our scheme, *hierarchical priority*, implements two levels of priorities - static (defined before run-time) and dynamic (changes based on memory intensity).

### 3.1 Hierarchical Priority

In this section, we focus on explaining how a hierarchical priority scheme works in disaggregated memory systems. Hereinafter, the systems discussed consist of multiple nodes that run simultaneously and share a global memory that is accessible through a fast interconnect. Moreover, each node runs a single application.

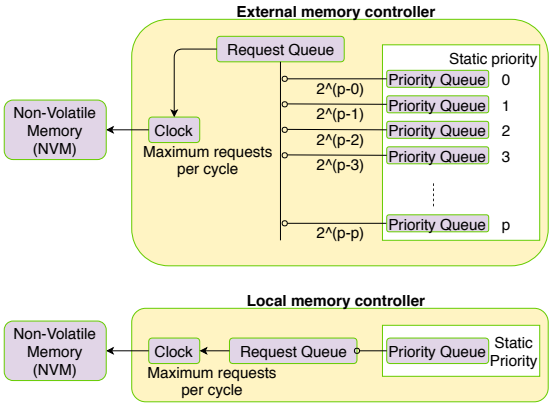


**Fig. 3.** Hierarchical priority based QoS implementation in Local and External memory controllers

**Static Priority** This is a fixed priority for each node which is configured during the initialization phase of the application[4]. The memory controller maintains a queue for each static priority level. However, to keep the number of queues practical, we limit this to a maximum of 8 static priority levels. Requests from

[4] Assigning static priorities to nodes is in accordance with service level agreement.

applications with similar static priorities are placed in the same queue. Note that this is, to some extent, similar to the state-of-the-art storage protocol, NVM Express [29], where multiple I/O submission queues are configured with different priorities but in the context of disaggregated memory systems.

Figure 3 depicts the implementation of static priorities. The disaggregated memory system has two types of memories: shared memory and local memory. Based on our design, the shared memory controller is required to maintain a priority queue for each static priority supported in our system. Each local memory controller maintains only one priority queue that can perform dynamic priority, as explained in section 3.1, if the node executes multiple applications. Request batches are pulled from priority queues based on their static priority levels. For instance, if we consider four static priority levels, for every batch of requests the memory controller serves up to 16 requests from the node with static priority of 0, up to 8 requests from the node with static priority of 1 and so on. So if we assume $p$ as the number of static priority levels and $i$ as the static priority of the node then number of requests addressed by memory controller in a batch can be represented as

$$R_{batch/node} = 2^{p-i} \tag{1}$$

**Dynamic Priority** In general, applications can be categorized into memory-bound and compute-bound applications. Memory-bound applications are prone to read and write data to memory frequently and require more memory bandwidth. Compute-bound applications are dependent on computation power of the system and are less prone to memory accesses.

Irrespective of the type, applications need to compete for the limited bandwidth and because of the wide gap between computing power and memory accessing capacity, memory bandwidth is limited for the applications. Due to this scenario, compute-bound applications, need to wait for the memory bandwidth to fetch small amount of information and this wait can be costly and can severely affect the performance of the applications. Memory-bound applications are less memory sensitive and can be delayed for memory accesses. By taking the variance in memory sensitivity into consideration, we define the dynamic priority based on the characteristics of the applications running in the nodes with a specific static priority. Applications with less memory requests should have higher priority and applications with more memory requests can bare less priority. Therefore, dynamic priority can be expressed as

$$P = \frac{R_{node}}{AR_{staticpriority}} \tag{2}$$

$P$ is the dynamic priority of the node, $R_{node}$ is number of requests per node and $AR_{staticpriority}$ is average number of requests per static priority which is represented as

$$AR_{staticpriority} = \frac{TR_{staticpriority}}{N_{staticpriority}} \tag{3}$$

where $TR_{staticpriority}$ is total number of requests from a specific static priority and $N_{staticpriority}$ is number of nodes with the same static priority.

According to Equation 2, low priority applications can starve if dominated by high priority applications. Thus, we use the request rate per period (epoch) for each node to calculate its dynamic priority. When the application has a smaller number of requests in the prior period, its dynamic priority will be higher and hence its requests do not get pre-empted by memory-intensive applications with the same static priority. The dynamic priority can be expressed as:

$$P = \frac{R_{node} * RR_{node}}{AR_{staticpriority}} \tag{4}$$

where $RR_{node}$ is the rate of requests per node which is calculated for every epoch[5]

Once the dynamic priority is calculated, the requests are prioritized within same static priority and then they are added to the request queue in accordance with the static priority. For every clock cycle, the memory controller serves requests relative to the priority. The combination of dynamic and static priorities meets the requirements of disaggregated memory system architectures and promises to enforce QoS by allocating more bandwidth to high priority applications while meeting the requirements of low priority applications, however, also ensuring fairness across applications with similar priority levels.

### 3.2 Splitting Shared Memory

Contention at the shared memory increases exponentially with the increase in number of compute units sharing it. Also, it is very unlikely and inefficient to maintain the entire shared memory in a single huge memory pool. Considering this, we explore the option of dividing the shared memory into multiple memory pools. By doing so, we can dedicate shared memory pools for high static priority nodes. In other words, high static priority nodes can have a dedicated memory pool that cannot be accessed by low static priority nodes or any other high static priority nodes. In this scenario, the memory bandwidth of each dedicated pool is devoted to the corresponding node running a high-priority application, which might lead to better performance. In contrast, memory pages of low static priority nodes can be allocated from a range of memory pools that are classified as low priority shared memory pools, i.e., not dedicated but rather shared between all nodes, including high static priority nodes. An example is illustrated in Figure 4. It can be seen that the shared memory is divided into a number of memory pools and nodes with high static priority have dedicated shared memory pools. Also, a chunk of the low priority pools is marked for high priority static nodes, which indicates that the low priority pools are accessed between all the nodes irrespective of their priorities. This way we assume to achieve better QoS along with improving performance.

If there are more nodes with high static priority then it would be difficult to divide shared memory into an appropriate number of memory pools. We address

[5] Time interval for which the rate of requests per node is calculated.

this by dividing shared memory proportionally. That is, some of the low priority memory pools are converted to high priority memory pools and also high static priority nodes can share dedicated memory pools.

The downside of this approach is that each node will have a portion of shared memory rather than entire shared memory. For high priority nodes, this can be manageable by allocating memory from low priority shared memory pools or free shared memory pools when the dedicated remote memory pool is full. But for low priority nodes, this option is narrow and can lead to starvation.

Centralized manager can allocate shared pages, from single shared memory pool, to the applications sharing information between them. Pages of the applications that are not shared with other applications can be assigned from different memory pools to avoid contention. With this we enable applications to share data with less contention from applications that do not share huge information.

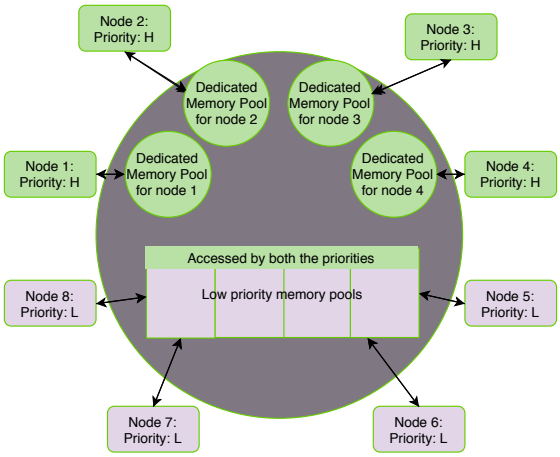


**Fig. 4.** Dedicating shared memory pools for high priority nodes.

## 4 Evaluation

To evaluate our QoS support for disaggregated memory systems, we extend the Structural Simulation Toolkit (SST) [28]. We have created an external memory with its respective memory controllers and connected it to compute nodes through a fast network modelled after GenZ [9] as described in [19]. As the local memory is expected to be very small in such systems [7], we deploy an alternating memory allocation policy [19], where memory is allocated alternatively from shared and local memory. The modules that we used to simulate disaggregated memory system is described in [19].

Table 1 describes the simulation parameters used to evaluate the design. L1, L2 and L3 caches are non-inclusive type and each are of sizes 32KB, 256KB and 16MB respectively. We used NVM as shared memory and considering the density of NVM, we maintained NVM size as twice the local memory per node.

We understand that NVM density is much higher compared to DRAM. Future disaggregated systems with higher densities can also benefit from our QoS approach. As explained in section 4.2, 4 nodes are simulated to study our approach. Hence, we used 16GB ($4nodes * 2 * localmemorysize$) of shared NVM. A maximum of 100 million instructions are executed in each core.

**Table 1.** Simulation Parameters

| Element | Parameters |
|---|---|
| CPU | 8 Out-of-Order cores, 2GHz, 2 issues/cycles, 32 max. outstanding requests |
| L1 | private, 64B blocks, 32KB, LRU |
| L2 | private, 64B blocks, 256KB, LRU |
| L3 | shared, 64B blocks ,16MB, LRU |
| Local memory | 2GB, DDR4-based DRAM |
| Global memory | 16GB, NVM-based DIMM (PCM), 128 max. outstanding requests, 16 banks, 300$ns$ Read Latency, 1000$ns$ Write Latency |
| External network latency | 20ns[4] |

For the simulation model, we use the alternate memory allocation policy in which memory is allocated alternatively from shared and local memories. For example, if the disaggregated memory system is configured to have a local memory and a shared memory pool, then for every two-page faults one page is allocated from shared memory and one from local memory. If shared memory is divided into pools, then every time a page is to be allocated from shared memory, it is allocated from a different shared memory pool, ensuring that the shared memory is evenly allocated.

Note that this a state-of-the-art model to demonstrate the advantages of dividing shared memory into multiple pools. We assume that each shared memory pool should have memory more than local memory (2GB). Since we are using a global memory of 16GB and if it is divided into 4 shared memory pools, each pool will have 4GB of memory, which is greater than local memory size. Hence we confine to dividing shared memory up to a maximum of 4 memory pools.

**Simulated Applications** Considering that our focus is on HPC applications in disaggregated memory environment, we choose 5 memory intensive HPC proxy applications to evaluate our design. Lulesh [17], a mini-app for hydrodynamics. Pennant [12] is an unstructured mesh physics mini-app designed for advanced architecture research. SimpleMOC [13], mini-app is to demonstrate the performance characteristics and viability of the Method of Characteristics (MOC) for 3D neutron transport calculations in the context of full scale light water reactor simulation. NASA IS [2, 3] mimic the computation and data movement characteristics of large scale computational fluid dynamics (CFD) applications. IS is an integer sort kernel which performs a sorting operation that is important in particle method codes. MiniFE [15] is a proxy application for unstructured implicit finite element codes. We decided upon these specific applications as these are memory intensive.

For the rest of the evaluation, $N$ indicates number of nodes. $SM$ is shared memory along with local memory wherein SM1 indicates 1 shared memory pool. For example, $N2$ with $SM2$ indicates 2 nodes with a local memory each

and 2 shared memory pools are available for the nodes to utilize. Mixes are a combination of applications running in each node which are explained in Table 2. *noqos* indicates experiments without any QoS. *hp* indicates experiments with hierarchical priority. Application running in each node is expressed as $a - n$ wherein $n$ indicates node number.

**Table 2.** Applications-Mixes

| Name | Description |
|---|---|
| mix-1 | miniFE-SimpleMoC-lulesh-pennant |
| mix-2 | SimpleMoC-lulesh-pennant-NAS:IS |
| mix-3 | lulesh-pennant-NAS:IS-miniFE |
| mix-4 | pennant-NAS:IS-miniFE-SimpleMoC |

### 4.1 The Impact of Number of Shared Memory Pools

In a disaggregated memory architecture, as the number of nodes in a system increases, the contention at memory increases exponentially, and thus the response time from the shared memory is expected to be slower with increase in the number of sharing nodes. Such delays affect the performance of individual nodes in addition to the overall system throughput. We calculate the performance of each node in terms of relative response time per memory request (RRT). An average of all the nodes is taken into consideration. RRT is relative to running the application in a node on the same system but without any applications running on the other nodes.

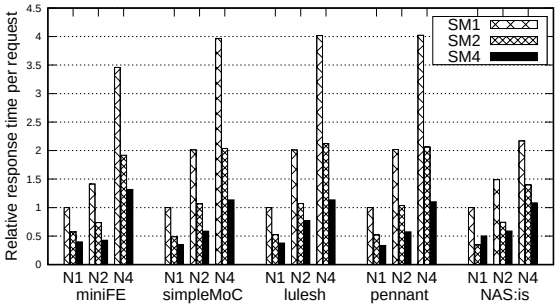


**Fig. 5.** Relative response time per memory request of disaggregated memory system model

Figure 5 show the impact of using multiple shared memory pools along with the resulting performance degradation under multiple nodes scenario without any kind of priorities. For instance, for *Pennant* application, RRT is 3x times with 4 nodes due to heavy contention at the single shared memory pool from other nodes, Figure 5.

Note that the memory concurrency is limited by the number of banks at the memory. Hence, when the entire shared memory is maintained in one shared memory pool, the parallelism is limited to 16 banks, according to our simulation parameters. When multiple shared memory pools are used, the level of parallelism is higher due to the increase in the number of banks, $4 * 16$ banks for 4 shared memory pools. It can be also observed that, for the same application, *Pennant*,

when shared memory is maintained in 4 memory pools, RRT is almost equal to the RRT when only one node is running in the system with one shared memory pool. Therefore, contention due to multiple nodes can be reduced by maintaining multiple shared memory pools.

It should also be noted for the same application, *Pennant*, if multiple shared memory pools are used with only one node in the system, the performance increases immensely (0.4x times RRT). This is due to huge memory concurrency and no contention at shared memory from other nodes as the system has only one node. Also, RRT of the system decreases as the number of shared memory pools increases due to less contention at each shared memory pool.

### 4.2 QoS using Hierarchical Priority

As the focus of this paper is to provide a proof-of-concept and due to the constraints of simulation time, we limit our evaluation to only 4 nodes with nodes 1 and 2 as high priority nodes and nodes 3 and 4 as low priority nodes. For every mixed workload, as shown in Table 2, the first 2 mentioned benchmark applications would be running in high priority nodes, nodes 1 and 2, and the remaining 2 benchmark applications run in the last 2 nodes, nodes 3 and 4, with low priority. Request frequency is calculated for each epoch period - every 1 million cycles (we varied the epoch size and empirically found that 1 million gives the most suitable epoch length).

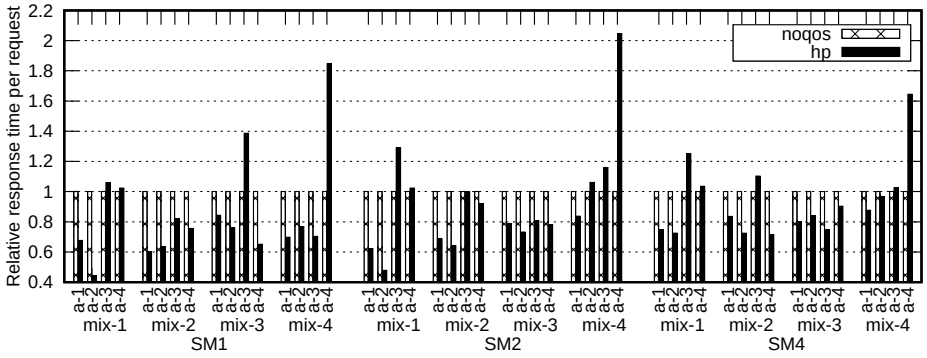


**Fig. 6.** Relative response time per request wherein global memory is divided into shared memory pools in disaggregated memory system.

Figure 6 depicts the performance of disaggregated memory systems under the hierarchical priority based QoS method and without QoS. It can be seen that in every mix the 2 nodes with high static priority modeled in hierarchical priority out performs the no QoS model. For example, in *mix-2* the RRT for nodes 1 and 2, when shared memory is not divided into multiple pools *SM1*, is reduced to around 0.6x each, as observed from Figure 6. We observed a maximum of 55% improvement in RRT (node 2 *mix-1*) with a single shared memory pool.

For low priority nodes the RRT is reduced for some mixes and it increases for some other mixes. This is due to less contention at shared memory from high priority nodes as they are addressed as soon as possible and different memory

footprints of the applications. For instance, in *mix-2*, RRT of nodes 3 and 4 is reduced to 0.8x and 0.7x respectively using hierarchical priority with one shared memory pool *SM1*. At the same time for *mix-3*, RRT for node 3 increases to 1.4x times.

We evaluated our design by dividing shared memory into 2 and 4 shared memory pools, Figure 6. We observed similar pattern using multiple shared memory pools but due to more bank parallelism the performance of the system is better as shown in Figure 5. An RRT improvement of up to 50% (node 2 in *mix-1*) for 2 shared memory pools and up to 28% (node 2 *mix-1*) for 4 shared memory pools can be observed in Figures 6.

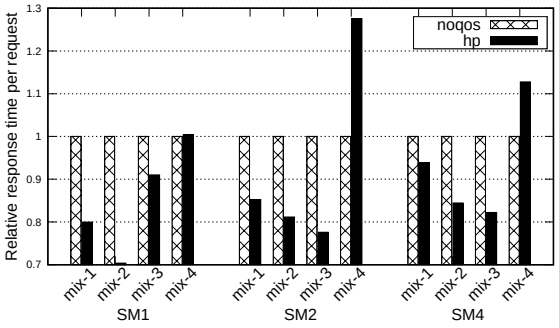


**Fig. 7.** Overall relative response time per request of disaggregated memory system using no QoS and hierarchical priority based QoS

Overall system performance in terms of RRT per memory request is shown in Figure 7. We could observe an overall maximum performance improvement of 30% using single shared memory with *mix-2*.

## 5 Conclusion

Memory-driven HPC applications and factors such as memory under-utilization, memory upgradability and information sharing make disaggregated memory systems a better choice but come at the cost of speed and performance. There are many factors that still need to be understood to effectively improve the performance of such systems. QoS is a crucial aspect that demands careful attention in disaggregated memory systems and is inversely proportional to the number of nodes. In this paper we carefully analyzed the performance of disaggregated memory system, proposed and studied QoS approaches.

We proposed hierarchical priority, a combination of static and dynamic priority-based QoS for disaggregated memory systems. We determined that assigning priorities to nodes and modifying priorities at run time can greatly improve the performance of the system. To improve the performance further, we divided the shared memory into pools and dedicated pools for specific nodes. Dedicating shared memory pools to specific nodes reduced the overall memory per node and not produce the anticipated performance gains due to less concurrency when accessing memory. Our conclusion opens up a new research direction to

explore more aspects related to disaggregated memory systems. We would like to extend this work by simulating more number of nodes and dividing shared memory into feasible number of pools.

## References


1. Adaptive resource optimizer for optimal high performance compute resource utilization. pp. 1–5. Synopsys Inc, silicon to software, Mountain View (2015)
2. Bailey, D.H.: Nas parallel benchmarks. In: Encyclopedia of Parallel Computing, pp. 1254–1259. Springer (2011)
3. Bailey, D.H., Barszcz, E., Barton, J.T., Browning, D.S., Carter, R.L., Dagum, L., Fatoohi, R.A., Frederickson, P.O., Lasinski, T.A., Schreiber, R.S., et al.: The nas parallel benchmarks. The International Journal of Supercomputing Applications **5**(3), 63–73 (1991)
4. Borkar, S.: Networks for multi-core chips–a controversial view. In: Workshop on on-and off-chip interconnection networks for multicore systems (OCIN), Stanford (2006)
5. Chen, A.: Emerging nonvolatile memory (nvm) technologies. In: Solid State Device Research Conference (ESSDERC), 2015 45th European. pp. 109–113. IEEE (2015)
6. Chowdhury, M., Zaharia, M., Ma, J., Jordan, M.I., Stoica, I.: Managing data transfers in computer clusters with orchestra. ACM SIGCOMM Computer Communication Review **41**(4), 98–109 (2011)
7. Comperchio, D., Stevens, J.: Emerging computing technologies: Hewlett-packard's "the machine" project. In: HP Discover 2014 conference held in Las Vegas June 10-12. pp. 1–4. Willdan Energy Solutions (2014)
8. Consortium, C., et al.: Cache coherent interconnect for accelerators (ccix). Online]. http://www. ccixconsortium. com (2017)
9. Consortium, G.Z., et al.: Gen-z–a new approach to data access (2017)
10. David, H., Fallin, C., Gorbatov, E., Hanebutte, U.R., Mutlu, O.: Memory power management via dynamic voltage/frequency scaling. In: Proceedings of the 8th ACM international conference on Autonomic computing. pp. 31–40. ACM (2011)
11. Deng, Q., Meisner, D., Ramos, L., Wenisch, T.F., Bianchini, R.: Memscale: active low-power modes for main memory. In: ACM SIGPLAN Notices. vol. 46, pp. 225–238. ACM (2011)
12. Ferenbaugh, C.R.: Pennant: an unstructured mesh mini-app for advanced architecture research. Concurrency and Computation: Practice and Experience **27**(17), 4555–4572 (2015)
13. Gunow, G., Tramm, J., Forget, B., Smith, K., He, T.: Simplemoc-a performance abstraction for 3d moc (2015)
14. Handy, J.: Understanding the intel/micron 3d xpoint memory. In: Proc. SDC (2015)
15. Heroux, M.A., Doerfler, D.W., Crozier, P.S., Willenbring, J.M., Edwards, H.C., Williams, A., Rajan, M., Keiter, E.R., Thornquist, H.K., Numrich, R.W.: Improving performance via mini-applications. Sandia National Laboratories, Tech. Rep. SAND2009-5574 **3** (2009)
16. Jeong, M.K., Erez, M., Sudanthi, C., Paver, N.: A qos-aware memory controller for dynamically balancing gpu and cpu bandwidth use in an mpsoc. In: Proceedings of the 49th Annual Design Automation Conference. pp. 850–855. ACM (2012)
17. Karlin, I., Keasler, J., Neely, J.: Lulesh 2.0 updates and changes. Tech. rep., Lawrence Livermore National Laboratory (LLNL), Livermore, CA (2013)

18. Khrabrov, A., De Lara, E.: Accelerating complex data transfer for cluster computing. In: HotCloud (2016)
19. Kommareddy, V., Awad, A., Hughes, C., Hammond, S.: Opal: A centralized memory manager for investigating disaggregated memory systems
20. Li, H., Chen, Y.: An overview of non-volatile memory technology and the implication for tools and architectures. In: Proceedings of the Conference on Design, Automation and Test in Europe. pp. 731–736. European Design and Automation Association (2009)
21. Lim, K., Chang, J., Mudge, T., Ranganathan, P., Reinhardt, S.K., Wenisch, T.F.: Disaggregated memory for expansion and sharing in blade servers. In: ACM SIGARCH Computer Architecture News. vol. 37, pp. 267–278. ACM (2009)
22. Lin, C.H., Yang, C.L., King, K.J.: Ppt: joint performance/power/thermal management of dram memory for multi-core systems. In: Proceedings of the 2009 ACM/IEEE international symposium on Low power electronics and design. pp. 93–98. ACM (2009)
23. Liu, J., Jaiyen, B., Veras, R., Mutlu, O.: Raidr: Retention-aware intelligent dram refresh. In: ACM SIGARCH Computer Architecture News. vol. 40, pp. 1–12. IEEE Computer Society (2012)
24. Liu, S., Leung, B., Neckar, A., Memik, S.O., Memik, G., Hardavellas, N.: Hardware/software techniques for dram thermal management (2011)
25. Liu, S., Pattabiraman, K., Moscibroda, T., Zorn, B.G.: Flikker: saving dram refresh-power through critical data partitioning. ACM SIGPLAN Notices **47**(4), 213–224 (2012)
26. Meyer, H., Sancho, J.C., Quiroga, J.V., Zyulkyarov, F., Roca, D., Nemirovsky, M.: Disaggregated computing. an evaluation of current trends for datacentres. Procedia Computer Science **108**, 685–694 (2017)
27. Pawlowski, J.T.: Hybrid memory cube: breakthrough dram performance with a fundamentally re-architected dram subsystem. In: Hot Chips. vol. 23 (2011)
28. Rodrigues, A.F., Hemmert, K.S., Barrett, B.W., Kersey, C., Oldfield, R., Weston, M., Risen, R., Cook, J., Rosenfeld, P., CooperBalls, E., et al.: The structural simulation toolkit. ACM SIGMETRICS Performance Evaluation Review **38**(4), 37–42 (2011)
29. Strass, H.: An introduction to nvme. Tech. rep., Seagate Technology LLC (2016)
30. Subramanian, L., Seshadri, V., Kim, Y., Jaiyen, B., Mutlu, O.: Predictable performance and fairness through accurate slowdown estimation in shared main memory systems. arXiv preprint arXiv:1805.05926 (2018)
31. Usui, H., Subramanian, L., Chang, K., Mutlu, O.: Squash: Simple qos-aware high-performance memory scheduler for heterogeneous systems with hardware accelerators. arXiv preprint arXiv:1505.07502 (2015)
32. Wu, D., He, B., Tang, X., Xu, J., Guo, M.: Ramzzz: rank-aware dram power management with dynamic migrations and demotions. In: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis. p. 32. IEEE Computer Society Press (2012)
33. Zhao, J., Mutlu, O., Xie, Y.: Firm: Fair and high-performance memory control for persistent memory systems. In: Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture. pp. 153–165. IEEE Computer Society (2014)
34. Zhou, P., Du, Y., Zhang, Y., Yang, J.: Fine-grained qos scheduling for pcm-based main memory systems. In: Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on. pp. 1–12. IEEE (2010)