A Practical Approach For Workload-Aware Data Movement in Disaggregated Memory Systems

Amit Puri, Kartheek Bellamkonda, Kailash Narreddy, John Jose, Tamarapalli Venkatesh
Dept. of CSE, IIT Guwahati, Assam, India
email: {amitpuri, bkartheek, n.kailash, johnjose, t.venkat}@iitg.ac.in

***Abstract*—Memory disaggregation is a solid alternative to traditional server systems that can overcome memory scalability issues in next-generation HPC data centers. In a rack-level disaggregated system, multiple compute nodes with small local memory rely on remote memory pools (memory nodes) to fulfill their memory demands. An in-network memory manager manages remote memory address space and allocates it to compute nodes which can access the memory at cache-line granularity using coherent interconnects such as CXL (or GenZ). However, the memory access cost is significantly increased due to the presence of the network. Even though a page migration system can exploit the locality of memory accesses, accessing a remote page starves the block-level requests. Further, page migrations introduce additional overheads which combined with starvation may even degrade the performance. All these issues require systematic evaluation of disaggregated memory systems to achieve improved designs.**

This paper presents a hardware mechanism for workload-aware data movement between compute and memory pools that significantly reduces the memory access cost. Firstly, our design enables centralized hot-page migration in a multi-tiered disaggregated memory that is aware of access patterns for individual compute nodes. Secondly, we analyze the complexities of accessing a remote memory page and propose a novel solution to eliminate starvation by serving all the remote memory requests at cache block granularity and by sharing bandwidth between page and block memory requests. Lastly, we add extra hardware support to get rid of additional overheads in a page migration system. We evaluate our designs over a variety of multi-threaded benchmarks using a cycle-level simulator which is specially designed to simulate a disaggregated memory system. Our design performs 10% to 100% better than traditional RDMA-based disaggregated systems that access remote memory at page granularity and 5% to 35% better than baseline disaggregated systems that use coherent interconnects for block-level access.

***Index Terms*—Data centers, Page migration, Memory disaggregation**

I. Introduction

The memory capacity wall has introduced scalability challenges to the data center servers which run large in-memory applications such as AI, Big-data, and video/graph analytics [23]. Even though enough computing power is available in large multi-core systems, ever-increasing memory footprints of data-centric workloads fail to utilize the available memory resources [34], which remains stranded in different server units and presents issues like memory under-utilization. Memory disaggregation has recently gained the interest of the research community due to its ability to address scalability and under-utilization issues [22], [31]–[33]. A bigger motivation is the availability of coherent interconnects, such as CXL or GenZ(merged with CXL) [10], [25], that allow high-bandwidth and low-latency access to on-network memory resources. With disaggregation, the server resources are decoupled into multiple resource pools, as shown in Fig. 1, allowing on-demand resource allocation. The resources from different pools combine to create virtual server units, improving scalability and utilization. Unlike the traditional server nodes, the compute nodes have a small amount of local memory and rely on memory pools for most of their memory requirements, due to which the memory access cost is increased 2 to 3 times that of local memory.

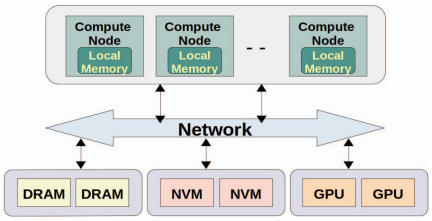


Fig. 1. Server Resource Disaggregation

A typical disaggregated memory system (software disaggregation) accesses remote memory at page-level granularity over an RDMA-enabled network. These systems swap out the memory pages to remote memory borrowed from other server nodes rather than swapping to slow disks. In hardware-based disaggregated systems, the memory-binding interconnects allow block-level access to remote memory on a cache miss, where local and remote memory (in memory pools) is organized linearly. Further, disaggregated systems feature multi-tiered memory management where a global memory manager manages the address space of memory pools beside a memory manager at compute nodes. The primary challenge in disaggregated memory systems is the high access cost to access remote memory blocks on a cache miss, significantly impacting performance. Like in other hybrid memory systems (such as DRAM-NVM), there is a scope for exploiting spatial locality in the workloads by migrating hot memory regions in slower memory to the faster memory [17], [29], [35], [44], [46]. However, moving pages in disaggregated memory systems have multiple challenges. Firstly, no centralized memory management exists in disaggregated systems like the one in a DRAM-NVM hybrid memory system, where a memory manager or TLB can predict hot memory pages in slower

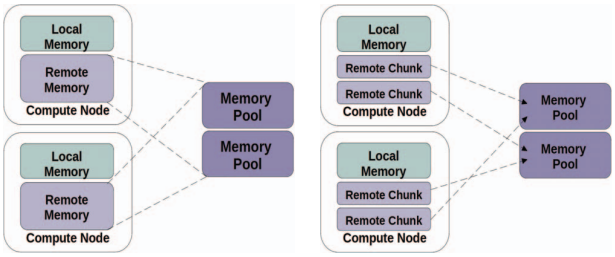
NVM. Secondly, data-centric workloads with large memory footprints expand to tera-bytes of memory, and significant metadata is required to track all the memory pages. Third and most importantly, accessing a remote memory page requires more network/memory bandwidth that starves the subsequent cache line accesses in their critical path, significantly impacting the system's performance. Further, workloads show a range of friendliness to page migration [7], and it is crucial to set the migration threshold based on the memory access patterns of each compute node. All these issues require careful investigation of disaggregated memory systems to be able to exploit the locality in remote memory pages. With efficient multi-granularity memory access, frequent block accesses to remote memory can be supported by occasional page access, reducing the memory access cost.

This paper proposes a hardware mechanism for workload-aware data movement in a disaggregated memory system that implements an epoch-based hot page migration to reduce memory access latency. Our approach is based on a centralized page migration system that supports rack-level disaggregation, where multiple compute nodes run simultaneously, each running a workload with distinct memory access patterns. The hardware support comes through an in-network memory manager [8], [22], which is extended to perform workload-aware page migration. Next, accessing a remote page takes around 1.2-1.5µs, 5 to 7 times more than a block access latency. However, it brings multiple blocks to local memory to complete future remote memory access locally. Also, reading a remote memory page (assuming a 4KB page) and sending its response back to the compute node increases the memory access cost of the subsequent block accesses and also starves them of memory/network bandwidth. Our mechanism for remote page access ensures that the starvation to block accesses is eliminated while also reducing the response time for the page access, which is now completed at block-level granularity rather than accessing a complete page altogether. Further, page migration systems (epoch-based [21] and on-the-fly [19]) have specific limitations that may even degrade the performance in a disaggregated memory system. Therefore, we add hardware support that takes advantage of both approaches to improve the performance further. Finally, we offer software support for centralized page migration that requires carefully selecting the migration parameters for each compute node based on its memory access patterns. A learning-based page migration policy is implemented that initially learns the workload behavior for each node and fixes the migration parameters. To evaluate our proposed design, we design a cycle-level disaggregated memory simulator that simulates multiple compute nodes and memory pools, a global memory manager, and remote memory access over the network interconnect. We extensively validate our simulator for measuring system performance (with one node having 100% local memory) against gem5 with good enough accuracy and integrate a standard DRAM simulator to simulate memory access. We summarize our contribution in this paper as follows:

- We propose a novel hardware mechanism for workload-aware data movement in rack-level disaggregated memory systems that occasionally fetch hot pages while performing frequent block accesses to remote memory.
- We build a cycle-approximate simulation framework to model a multi-node disaggregated memory system and implement our data movement mechanism on top of it.
- We analyze the major hurdles of multi-granularity remote memory access in a disaggregated system and implement an approach to neutralize the extra overheads and delays due to page access and migrations.
- Finally, we evaluate our proposed mechanism over various multi-threaded HPC workloads and mini-applications and shows a significant improvement in the system performance.

II. Background and Motivation

Baseline Disaggregated Memory Systems: Typical disaggregated memory systems access remote memory at page granularity and uses RDMA as the underlying mechanism. These systems utilize free memory available in other server nodes (virtual disaggregation) and swap out pages to remote memory rather than disks, speeding up future page faults to those pages. On the other hand, hardware memory disaggregation allows block-level access to remote memory, which is managed as separate pools. These systems use remote memory as an extension to local memory rather just as a swap device. Remote memory address space can be made visible to compute nodes using a shared or distributed memory approach. With a shared approach, all the remote address space is transparent to OS at compute nodes, as shown in Fig. 2(a), and a memory page can be allocated at any address. However, multiple compute nodes may try to allocate a page concurrently, causing a conflict. Thus, an in-network global memory manager must allocate a remote page on behalf of compute nodes, which may also face a bottleneck due to frequent page requests. But the approach makes it easy to share pages between nodes. Alternatively, a distributed approach may be implemented to add or remove the remote memory at run-time in larger chunks (using memory hotplug), as shown in Fig. 2(b). Once the memory is reserved and added to the compute node, it may allocate memory pages without conflict using its local memory manager. This approach removes the page allocation bottleneck at the global memory manager and significantly reduces the metadata overhead at compute nodes but requires another layer for address translation. Further, the in-network global memory manager is implemented at a programmable central switch to provide memory allocation and protection. Our approach utilizes caching structures and DRAM (to store metadata) at the switch as also has been proposed in the past for disaggregated memory [1], [2], [6], [8], [22], [39], shown in Fig. 3. Additionally, compute nodes have an address translation unit to convert between local and remote physical addresses and a network interface for remote memory access (also present at memory pools).



(a) (b)

Fig. 2. Memory Management in Fully Disaggregated Memory Systems (a) Shared Memory Approach (b) Distributed Memory Approach

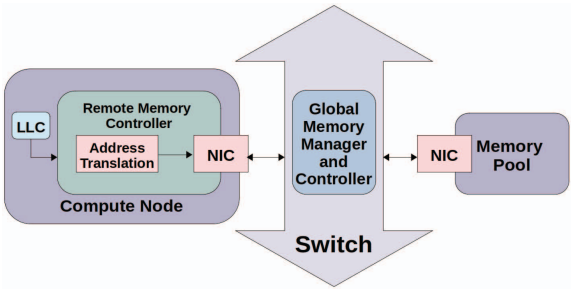


Fig. 3. Baseline Hardware Disaggregated Memory System

Our hardware support for efficient workload-aware data movement comes through the global memory controller at the centralized switch. The proposed mechanism works for both shared and distributed remote memory organizations. However, in our experimentation, we assume no page sharing among multiple nodes. Most data-centric workloads hardly share pages [16], [38], and even if application threads spread across multiple compute nodes, they follow different workflows. Further, with a large multi-core system, applications mostly fall short of memory rather than computing power and will not require shared memory access, especially with disaggregated memory.

Page Migration Overheads and Parameters: Migrating pages requires updating address translations in the page table entries (PTEs) with new mappings. While updating the PTEs, TLBs are locked to perform invalidation of migrated page entries (known as TLB shoot-down), during which OS interrupts the user application and issues an inter-processor interrupt (IPI) [28] to other cores with the same page entry (also to other compute nodes, in shared memory approach). Similarly, cache invalidation is required for the blocks with old physical tags. Performing invalidation is expensive that introduces long CPU stalls until it is performed and acknowledged by all the cores. Further, migration generates extra TLB misses for re-accessing invalidated entries, taking 60-80 cycles per page for TLB-miss on average [30]. Lastly, each page transfer from remote to local memory takes around 1.2-1.5 μs and delays the subsequent block access while the page is read from memory transferred to compute node.

Epoch-based page migration primarily requires three parameters. Firstly, an epoch length decides how often the pages should be migrated. If it is small, frequent page migrations introduce continuous CPU stalls and excessive overhead. If it is large, all the future accesses to hot pages will be complete at slower memory even before the migration. Secondly, the hotness threshold describes the minimum criteria for a page to be migrated that can be identified in various ways, such as access count/frequency to a page or other ways. If the threshold is strict, it will not migrate many probable hot pages. If it is lenient, many pages will become eligible for migration, increasing the network traffic and causing more starvation to block-level accesses. However, the threshold varies for different workloads, and the decision for migration should be taken based on the expected benefits from migration rather than a compulsion. Migrating useless pages also means the system is trying to overkill the benefits of page migration. Many pages might not even be accessed after migration, evicting more local victim pages in turn. Lastly, the number of pages to migrate (NPM) describes how many pages should be migrated together. If NPM is less, there will be frequent interrupts with CPU stalls which also invalidate TLB entries for each batch of page migration. If the batch size is large, the benefits of migration will be lost due to extra wait before the pages are brought to local memory.

Considering all the trade-offs of page migration in disaggregated memory systems, it is essential to choose migration parameters wisely for a centralized page migration system, which may only rely on small caches to track pages and can not afford to track all the pages in memory due to architectural restrictions.

III. System Design

Overview: This section discusses the proposed hardware structures to support a centralized Page-migration with workload-aware data movement that eliminates the bandwidth and starvation issues. Firstly, the central switch differentiates the memory accesses of individual nodes by reading the access packets and pass this information to the global memory controller. This allows the controller to characterize the access pattern for each node separately and fix the page migration parameters and access priorities. The global memory controller holds multiple new hardware structures whose sizes can be scaled to support any number of nodes. However, a limited number of compute nodes (C) and memory pools (M) are expected to be grouped inside a rack with specific configurations (say, at a fixed ratio of 1C:2M or 1C:4M). These configurations are unknown, as continuous research is being done in disaggregated memory space, while we assume support for a maximum of 16 nodes. The overview of the design of the global memory controller can be seen in Fig. 4. The new hardware structures are 1) Hot-Page Tracker, 2) Access Controller, 3) Pending Blocks Accesses queues, and 4) Page Buffers to store accessed memory pages

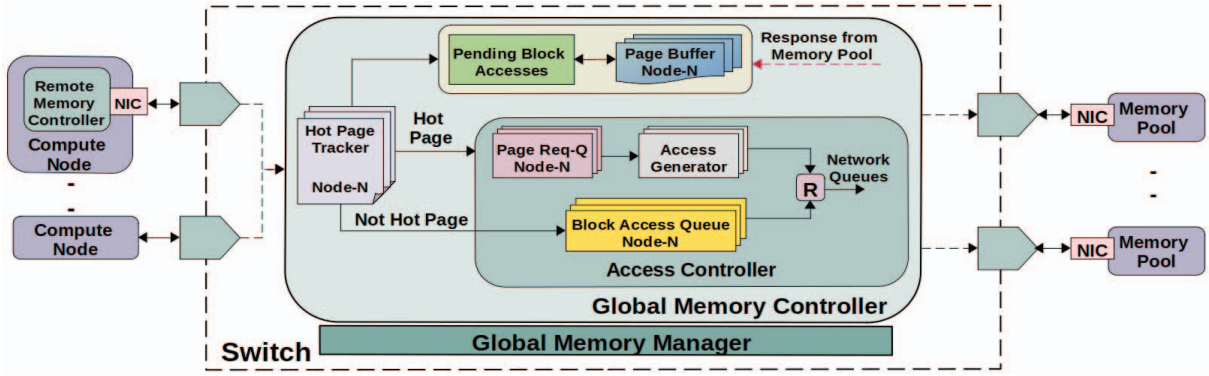
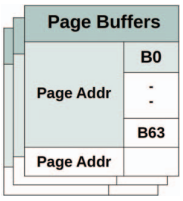


Fig. 4. Centralized Page Migration Support with Workload-Aware Efficient Data Movement, 'R' represents a Request Selector

Hot Page Tracking Table			
Page Addr	Access Count	Access Time	Access Bit
Entry1			
-			
EntryN			

(a)



(b)

Fig. 5. (a) Hot Page Tracking Table structure (b) Page Buffer structure

A. Hot Page Tracking

The controller supports predicting hot pages by tracking access count and frequency to remote pages. The hot-page tracker (HPT) consists of a table to store information on the most recently accessed pages with a maximum of 100 entries per node (also limits the number of pages migrated together). Each entry consists of the *page address* (32-bit), *access count* (16-bit), *first access time* (32-bit), and *Access_bit* (1-bit). However, there could be more active pages at a time that will not fit in the cached table. In that case, a new entry will be created by replacing the old one with minimum access count and oldest access time. The evicted entry is kept in a similar table at the switch DRAM and loaded back when that page is re-accessed. A page is identified as hot when it crosses the hotness threshold (explained later) and is based on access count and reuse frequency. The reuse frequency can be calculated using the page's access count and first access time. On identifying a hot page, a request is added to the page request queue (inside Access Control block) with its page address, and the *Access_bit* is set to 1 and remains set until it is migrated to compute node. The future memory accesses to the same page are not sent to the memory pool and are completed at the global memory controller (described in subsection III-D). However, if the page is not hot yet and *Access_bit* is '0', the block request is added to the block access queues (shown in yellow in Fig. 4).

B. Performing Migration and Using Page Buffers

Once the number of hot pages in the HPT becomes equal to NPM, the migration will be performed by swapping the same number of local victim pages at compute node to the address of migrated remote pages. One limitation of the epoch-based page migration is that many benefits are lost until the batch of hot pages is ready, especially for workloads with a high temporal locality. This can be eliminated with on-the-fly page migration, which instantly migrates a page as it is identified as hot, but has high overhead due to frequent TLB shoot-downs. We take a middle path by using page buffers (as cache) at the global memory controller with space for 100 pages per node. The page is instantly accessed as it becomes hot and kept in this cache until the whole batch is ready to migrate. When memory access to any of these pages arrives, it is completed at the controller itself through page buffers (costing less than half of a remote memory access cost), getting the best of two techniques. Further, keeping this buffer at a central controller also allows clean access to shared pages between nodes (in the case of a shared memory approach).

C. Access Controller

1) Handling Page Access: Once a page is identified as hot, its request is added in the Page Request Queue but is not send to memory queue as it is. Firstly, remote page access latency is high, which delays the pending block-level accesses to the same page. Secondly, page access occupies the available memory/network bandwidth and obstructs subsequent block accesses to other pages, adding long delays again. We overcome this problem by servicing page requests at a finer granularity and responding as soon as a block request within a page completes at the memory pool. Fig. 6 shows the detailed mechanism by which Access Controller handles the page and block-level requests. A page request eventually accesses 64 contiguous memory blocks (4KB page with 64B block). Rather than completing page access in one go, it can be accessed block-by-block. The access generator will pick a page address from the front of the page request queue and generates 64 block accesses to that page from block-0 to block-63 (using a fixed-size queue at the access generator). Access

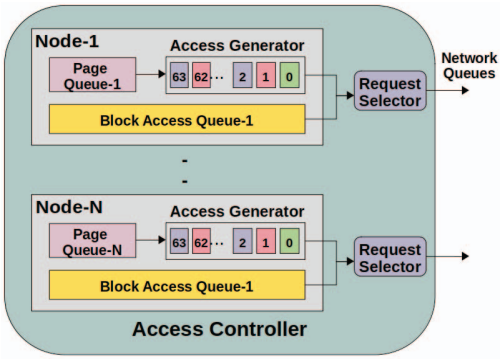


Fig. 6. Access Controller to control Multi-granularity Access

control has separate hardware structures for each node where its request selector forwards the chosen request to the network queues.

2) Handling Block Accesses and Bandwidth Allocation: When block accesses do not belong to a page request (the page is not hot), they are treated as regular requests and kept in block access queues (one for each node). Like page queues, each node has separate block access queues. We implement bandwidth partitioning between the page and block-level accesses to eliminate starvation and reduce waiting times for all types of access. In each cycle, the request selector will choose one of the requests either from the block access queue (for regular block access) or from the access generator queue (part of page access). Further, each queue can be allocated different priority levels to prioritize one type of request over another. The controller adds extra information to the packet header of selected requests before forwarding them to network queues to differentiate between regular block requests and those belonging to the page access. The response packet from the memory pool also includes the same information in its header, allowing the controller to take appropriate action when a response is received. If the response packet belongs to page access, it stores the block in the appropriate page buffer by matching the destination compute node and page address. If the response belongs to regular block accesses, it is directly forwarded to the destination compute node.

D. Pending Block Accesses

Once a page access request is sent and the page is undergoing access or is present in the page buffer, all the block accesses to that page are halted at the controller and directed to Pending Block Access queues, as shown in Fig. 4. The response is instantly sent to the compute node if the block is available in the page buffer. However, the block request waits in the queues if it is not there. The queues have separate buffers for storing reads and write. Reads queues store page address and page offset, whereas write queues also have space to store a block of data. When a new block arrives in the page buffer, it checks for a pending block access with the same address and completes the access by sending a response back to the source node. In case of a pending write request, the data block inside the page buffer (fetched from the memory pool and is dirty now) is updated with the data in pending write queues.

E. Remote Memory Access Data Path

Whenever a block request of a compute node arrives at the global memory controller, it will update the page tracking parameters. If the page becomes hot, a page access request is created using access generators, and *access_bit* is set, while the current block request is also added to pending block queues. However, the request is added to regular block access queues if the *access_bit* is '0'. The regular memory accesses will complete as usual by sending their response to the requesting compute node. On the other hand, if the response packet belonging to the page request arrives (identified by the packet header), the response is stored in the page buffer of the respective node buffer at an entry matching the page address. The pending block accesses to that page are also served if the page address matches. However, the pages are only migrated when a whole batch of hot pages is available in the page buffer. The global memory controller will then notify the respective compute node to perform page migration, for which the OS at compute node evicts an equal number of local pages. The eviction can be performed using basic memory page replacement policies such as LRU, clock replacement, or finding cold pages using access counters.

F. Hardware Overheads

We discuss the overhead of proposed hardware cache structures at the global memory controller to support a maximum of 16 nodes in any node-to-memory pool configuration. The HPT has 100 entries per node with 71 bits for each entry, approximately 14KB for 16 nodes. The page request queue inside Access Controller has only four entries per node, which is the maximum number of on-flight page requests. Each entry stores a page address (32-bit). The access generators have 64 entries (one for each page block) with a 32-bit page address and a 6-bit block address. The access controller requires 5KB in total for 16 nodes. The size of regular and pending block access queues will depend on the MSHR size of the last-level cache at the nodes. The memory requests will also be distributed among these queues, many of which are served instantly without delay. For a node with a 32-core system and 256 entries in MSHR, we assume 96 entries in the regular block access queue and only 32 entries for the pending block access queue (equally divided for pending reads and writes), as only a few on-flight page requests can be there. Each entry stores the page address (32-bit), block address (6-bit), and source node-id (4-bit) for memory access. The pending block queues additionally have 64B of data for write requests. The total size of all these queues is around 26.5KB for 16 nodes for both of these queues. Finally, the page buffers store 100 pages of size 4KB per node with its 32-bit address. This will require a slightly bigger cache of around 6.25MB but provides significant benefits by using positives from both on-the-fly and epoch-based migration.

G. Characterizing Workloads with Training

1) Setting Migration Parameters: We analyze the memory access pattern for each compute node to set the hotness threshold. An epoch-based page migration policy requires setting three migration parameters. However, if pages and hotness threshold is known, then a fixed epoch length is not required, as the system will reach a stage when the other two conditions are met. We also do not fix the NPM parameter and start migration with a small NPM (say 25) while changing it dynamically based on the feedback from the compute node, which is a more practical approach than fixed values. The hotness threshold is set using the collected information during training. When a process starts, page migration is kept off initially for a few million cycles, during which the global memory controller collects the access count and reuse frequency of all the touched pages in its DRAM (in the same way as during hot-page tracking). At the end of this phase, the pages are sorted by access count, and filtration is performed to remove less significant entries. The page entries with an access count lesser than the mean are removed. The filtration may be repeated to set a more conservative threshold until the list does not get too small (20-30% of the initial size). Finally, threshold parameters are calculated using the mean of access count and reuse frequency of the leftover page entries.

2) Migration Feedback: The OS at compute node can run a daemon program in the background to evaluate the benefits and the overheads of page migration. We track accesses to migrated and victim pages for each migration batch and evaluate it after every few batches. Based on this evaluation, a feedback score is generated and shared with the global memory controller. System overheads not only involve *tlb shootdown* (or invalidation) time but also the time to copy pages, time to re-access invalidated TLB entries (NPM multiplied by TLB miss time), and time to access victim pages in remote memory, in eq. 1. The benefits are calculated by multiplying total memory accesses to migrated pages with the difference in remote and local memory access time, in eq. 2. Finally, a migration score is produced using eq. 3, normalized on a scale of -100 to 100 and sent to the global controller. The controller modifies the NPM of a compute node based on its feedback score, which is either positive or negative according to the overheads and benefits of page migration. If the overhead is negative continuously or NPM falls below a certain threshold, re-initialization is done to reset parameters.

$$Overhead_{mig} = T_{inv} + T_{tlb\_miss} + T_{copy\_pages} + T_{Vpage\_acc} \quad (1)$$

$$Benefit_{mig} = Acc\_Count \times MAT_{Remote-Local} \quad (2)$$

$$\underset{\lim(-100 \to 100)}{\%Score_{mig}} = \frac{(Benefit_{mig} - Overhead_{mig})}{Overhead_{mig}} \times 100 \quad (3)$$

$$NPM_{new} = NPM + NPM \times \frac{Score_{mig}}{100} \quad (4)$$

TABLE I
SIMULATION PARAMETERS

CPU	3.6GHz, 4-core, 2-width 64-InsQ, 64-RS, 192-ROB, 128-LSQ
L1 Cache	32KB(I/D), 8-Way, 2-Cyc
L2 Cache	256KB, 4-Way, 20-Cyc
L3 Cache	2MB per core shared, 16-Way, 40-Cyc
Cache Type	Write-Back/Write-Allocate, Round-Robin
Memory (Local/Remote)	1200x2MHz DDR4 DRAM (19.2GB/s)
Switch	100/400Gbps, 4MB port Buffer 5ns for processing/switching
Network Interface (Nodes)	40/100Gbps, 1MB buffer 10ns (de)packetization/processing

TABLE II
BENCHMARKS

SimpleMOC(s) [15]	Light Water Reactor Simulation
miniFE [9]	Unstructured Implicit Finite Element Codes
Lulesh [20]	Unstructured Hydrodynamics
XSBench(l) [42]	Monte Carlo Neutron Transport Kernel
Testdfft [26]	Fast-Fourier Transform for HACC
Pennant [11]	Lagrangian staggered-grid hydrodynamics
NPB (bt, dc, ft, mg) [13]	Computational fluid dynamics

IV. EXPERIMENTAL METHODOLOGY AND RESULTS

A. Methodology

In the absence of a standard simulator, it is important and challenging to experiment with the new designs over a reliable platform. Therefore, we build a cycle-level disaggregated memory simulator that supports the simultaneous running of multiple compute nodes and memory pools. It includes a local memory manager at compute nodes to decide a page allocation between local and remote memory and a global memory manager that allocates remote memory to compute nodes from one of the memory pools in 4MB chunks. Lastly, an interconnect is modeled to simulate remote memory accesses in memory pools. [1]. We model our proposed hardware mechanism for the centralized page migration on our disaggregated memory simulator.

For the reliability of results, we extensively validate the performance of out-of-order CPU cores at compute node and cache miss results at the LLC for up to 4-cores (for 1 compute node and 100% local memory) against gem5 by using multi-threaded Splash3 benchmarks [37]. We encountered a mean error of 12% for IPC and 2% for LLC misses for all workloads. A small variation is expected due to implementation details, as pointed out by the past research work in simulator design [3], [4]. We use a simple queue modeling for the interconnect and show its impact in the result section by evaluating different network configurations.

We use Intel's PIN [24] platform to generate instruction-level traces at the front end. The thread-wise traces are passed to the back end to simulate an out-of-order x86 CPU and a multi-level cache hierarchy (at compute node). The local

[1]code is available at https://github.com/Amit-P89/-DRackSim/tree/main/DRACKSim-Detailed

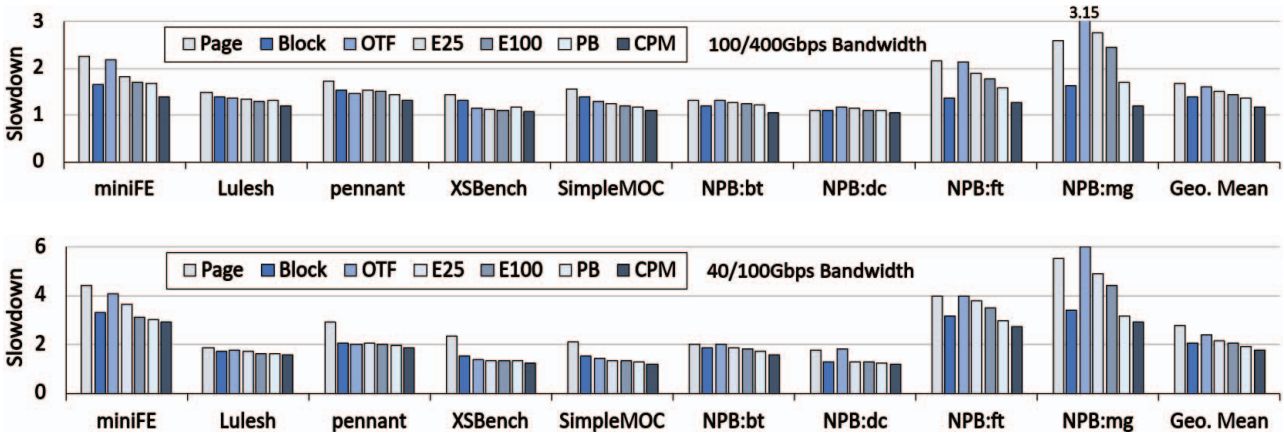


Fig. 7. Performance Slowdown for all the workloads with different data movement policies

memory manager manages address space at compute nodes and sends requests to the global memory manager on page fault if it requires more remote memory to allocate a new page. The interconnect includes a network interface at compute nodes and memory pools and a central switch for sending remote memory access. Further, we use a two-level arbitrator at the switch to select a packet from multiple input ports and virtual queues within a selected port. To simulate the memory accesses at different compute nodes and memory pools, we use cycle-accurate DRAMSim2 [36] and initialize multiple instances of DRAM units for local memory at compute node and remote memory at memory pools. At compute nodes, LLC misses belonging to remote memory are forwarded to the network interface as a network packet. The packets are pushed to the central switch queues after adding packetization delays which reach the memory pool to simulate the memory access. A response packet is generated and returned to the requesting compute node (once notified by DRAMSim2 for memory access completion). We model different latency parameters for the interconnect (packetization time, NIC delay, switch processing delay, propagation delay) and bandwidth (for transmission delay) at compute/memory nodes and the switch.

Disaggregated memory systems are to be used with multiple compute nodes and memory pools that significantly impact the performance due to memory access traffic on the network and contention at memory queues of shared memory pools. So, we design a simulator from the top down to perform multi-node simulations (with no page-sharing across nodes). We use multiple instances of Pintool to produce multiple instruction traces simultaneously at the front end (one for each node) and process them in parallel with multiple simulation threads.

Finally, we use various HPC applications and workloads that mimic different scientific applications and have a variety of memory access patterns and footprints (ranging from 50MB to 830MB). Table II mentions all the selected workloads with their functionality. We skip the initial single-threaded regions for each workload and simulate 200 million instructions only for the multi-threaded region. Table I mentions all the simulation parameters at the compute nodes, memory pools, and the interconnect. We evaluate our proposed design over two network configurations, one with 100Gbps and 400Gbps bandwidth at NIC (compute node and memory pool) and switch, respectively. The other one uses 40Gbps and 100Gbps of bandwidth.

B. Results

We evaluate our page migration system with other data movement scenarios. **Page** represents a traditional disaggregated memory system where data is only transferred at page-level granularity. However, these systems are inherently slow compared to hardware disaggregated memory systems. So we use similar page buffers (as in our design) to delay the page-table updates. **Block** represents a baseline hardware memory disaggregation with all the remote memory accesses at block-level granularity. **OTF** is an on-the-fly page migration on the same system without extra hardware support. **E25** and **E100** represent epoch-based page migration in batches of 25 and 100 pages, respectively. **PB** is the same as our proposed design, but the remote page access is made all together without any bandwidth partitioning, and the response is sent as a 4KB packet. Also, there are no pending memory access queues for in-flight page requests at the global memory controller. Finally, **CPM** is our proposed centralized hot page migration system with all its features enabled. Further, we use the same hot-page identification mechanism for OTF, E25, and E100. The memory page allocations are performed across local and remote memory at a fixed ratio of 50:50 using a round-robin policy (unless mentioned otherwise). To keep the memory ratio constant, we pre-evict the same number of victim pages from local memory for every page migration using a clock-replacement policy.

1) Impact on System Performance: We first evaluate the slowdown in system performance compared to a system using entirely local memory. We run each workload in a single node configuration using one remote memory pool. As shown

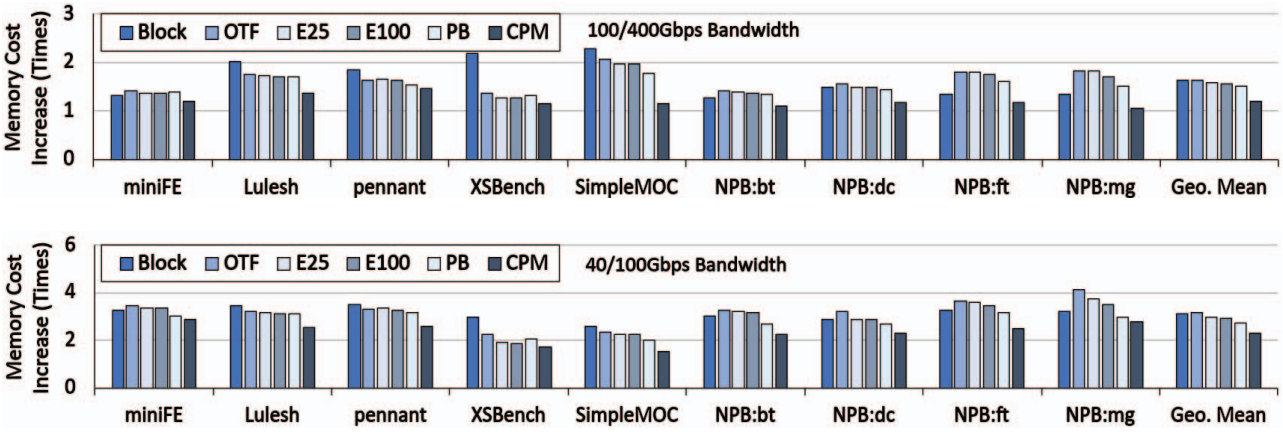


Fig. 8. Increase in memory access cost for all the workloads with different data movement policies

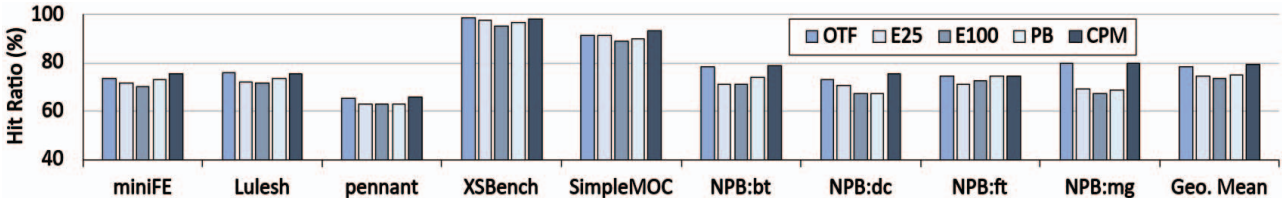


Fig. 9. Percentage of memory access at local memory due to migration of pages

in Fig. 7, CPM experiences the minimum slowdown among all the data movement schemes for all workloads at both network configurations. With 100/400Gbps bandwidth, the performance for *XSBench*, *NPB:bt*, and *NPB:dc* is very close to the system with 100% local memory even with 50% of memory footprint at local memory. As expected, PAGE suffers the maximum slowdown as it has to access all the remote pages at 4KB granularity, which increase the waiting times of pending memory accesses to those pages. Epoch-based page migration, such as E25, and E100, improved performance compared to PAGE but does not always perform better than the baseline BLOCK, as it misses out on many benefits due to the long waiting time before a batch of pages is ready to migrate. Only in a few cases, when a workload has good spatial locality, epoch-based migrations perform better than the baseline. On the other hand, OTF suffers severe slowdowns in some cases (miniFE, NPB:ft and NPB:mg), when more pages are migrated. As there are no page buffers with OTF, it performs worse or equivalent to PAGE in these cases due to regular CPU stalls during TLB shoot-downs. PB could eliminate the CPU stalls by using page buffers, not miss out on the migration benefits due to instant migration, and improve the performance for all workloads compared to baseline BLOCK. Finally, CPM further improves the performance of PB by proper bandwidth allocation to page and block requests and eliminates starvation. Further, CPM managed good enough performance even with 40/100Gbps for most workloads except miniFE, NPB:ft and NPB:mg, as they have the maximum number of cache misses.

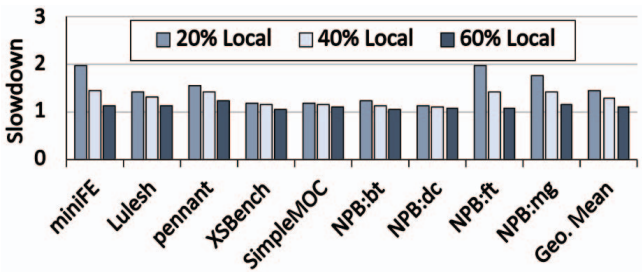
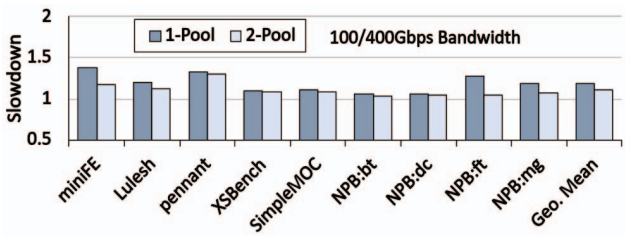
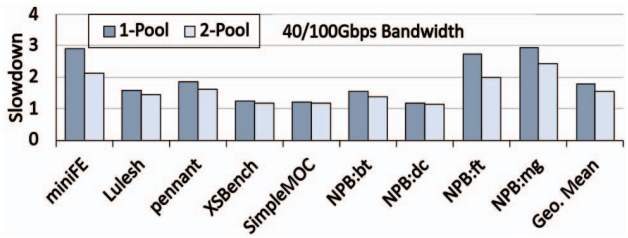


Fig. 10. Performance slowdown on changing the local memory footprint

2) Impact on Memory Access Cost: Fig. 8 shows the increase in memory access cost for all the above data movement schemes over two network configurations. As depicted by the system performance, CPM has the lowest increase in overall memory latency and averages around 1.25 times compared to local-only memory latency over a 100/400Gbps network. In the case of a 40/100Gbps network, the average increase in memory cost is around 2.25 times the local memory access latency. The baseline BLOCK and OTF suffers the most in their memory access latency. However, memory latency does not reveal the performance slowdown for page migration systems, especially for OTF, as it hides the long CPU stalls after the migration. We do not show the results for PAGE because page requests are queued up at remote memory due to significantly high page access times that could not correctly represent the waiting times for last-level cache misses.

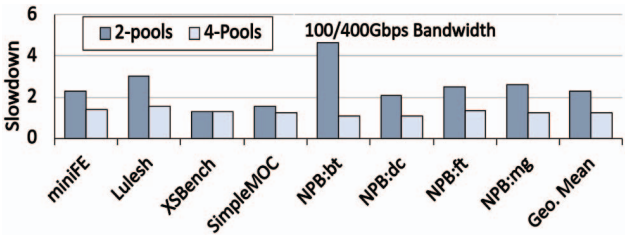


(a)

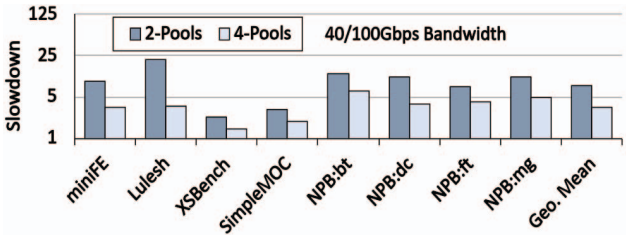


(b)

Fig. 11. Performance slowdown on using multiple memory pools



(a)



(b)

Fig. 12. Performance Slowdown with Multiple Compute and Memory Nodes

3) Impact on Hit-ratio at Local Memory: Fig. 9 shows the percentage of memory accesses completed at local memory as the consequences of page migration. For all the workloads, CPM has most of the memory accesses at local memory. The results for PB and CPM also include a few percentages of the pending memory accesses completed at the global memory controller using page buffers until a batch of pages gets ready to migrate. OTF experiences a similar percentage of local memory accesses compared to CPM, but the overheads did not allow them to experience similar performance gains. Further, CPM and PB have a significant difference in local memory hit ratio, that is because the pages are accessed at 4KB granularity in PB, which takes more time, and many block accesses to those pages are completed at remote memory before the migration.

C. Sensitivity Analysis

We further experiment by changing the memory-related parameters and different deployment configurations by changing the number of compute nodes and memory pools.

1) Local-to-Remote Memory Allocation Percentage: First, we change the memory allocation percentages at the local and remote memory by allocating pages in the same percentage (1 out of every 5 pages is allocated local memory to maintain 20% local footprint). Fig. 10 shows the performance slowdown with 20%, 40%, and 60% of local memory compared to a system with completely local memory. For XSBench, SimpleMOC, NPB:bt, and NPB:dc, the performance with even 20% of the local memory is around 80% of the local-only systems due to the high spatial locality in these workloads. Once the pages are migrated, most of the memory accesses are completed in local memory. With the novel hardware mechanism of CPM, even during page access and migrations, the overall impact of using remote memory is minimal. On the other hand, miniFE, NPB:ft, and NPB:mg faces more slowdown due to a decrease in application footprint on the local memory.

2) Multiple Memory Pools: Next, we evaluate the performance improvement when a compute node uses multiple memory pools rather than a single one. This results in an overall increase in the memory bandwidth and improves the memory access latency by reducing contention at the remote memory queues. Fig. 11 shows the performance slowdown for each workload compared to the local-only system when the remote memory pages are spread across multiple memory pools. As we can see, in both network configurations, the workloads face lesser slowdowns when the pages are spread across two memory pools compared to a single memory pool. The slowdown is more significant in the case of a 40/100Gbps network.

3) Multiple Compute Nodes and Memory Pools: Finally, we evaluate different configurations of multiple compute nodes and memory pools, which is the expected way of deployment for the hardware disaggregated memory systems. We consider 8-compute nodes and configure them in a 4:1 or 2:1 ratio with memory pools. The local-to-remote memory allocation ratio is fixed at 50:50, and the memory pool selection is done using a round-robin policy (to allocate a 4MB chunk on each request by the compute nodes). Fig. 12 shows the performance slowdown for all the workloads (one on each node) running together with different network configurations and node-to-pool ratios. As we can see, the performance impact is minimal with four memory pools, making it an optimal choice for a node-to-pool configuration ratio. Whereas, over a slower

network and a 4:1 ratio, the slowdown is significant and is around 9.6x of the local-only system.

V. Related Work

Page migration has been used for hybrid DRAM-NVM [5], [18], [35], [41], [44], [45] memory systems to bring frequently accessed pages. Taekyung et al. [18] proposed a system to migrate huge pages targeting the tired memory systems which suffer from excessive misses in translation look-aside buffers (TLB). Wang [44] proposed a system to support super-pages in NVM but still perform the migrations in light-weight pages to the DRAM. Shuang et al. [45] apply hot-page migration to cloud computing platforms and devise a hot-page capturer for virtual machine migration to reduce the remote page faults during a restart at the remote node. Yujuan et al. [40], [41] proposed 'UIMigrate' that selects hot pages from NVM and dynamically adjusts the hotness for page migration using access counters. These techniques do not work for hardware disaggregated memory with a multi-tiered memory system making tracking pages difficult. Further, data-center applications expand to tera-bytes of memory and without a centralized manager, require a dedicated hot-page tracker which uses the least amount of meta-data. Page migration has also been proposed for systems with software/virtual disaggregated memory [5], [12], [14], [27], [43], which allow only page-based remote memory access and does not support cache-based access. These systems use RDMA to exploit the free memory in other server nodes and replace slow disk paging with comparatively faster remote memory paging and are not a replacement for hardware disaggregation and are just the older solutions to improve memory scalability. Komareddy et al. [21] proposed a global memory controller hosted at the rack switch. Although page migration in disaggregated memory was proposed for the first time, the controller delays the migrations to miss out on the potential benefits and uses fixed parameters for migration without any intelligence. Finally, there is little opportunity to translate the available designs for page migration in hybrid memory, or software disaggregated systems to fully disaggregated memory systems that support multi-granularity memory access

VI. Conclusion

Disaggregated memory systems solve the problem of the under-utilization problem and improve memory scalability by allowing on-demand memory allocation from the remote memory pools. However, coherent interconnects allow cache-based to remote memory, the high memory latency in such a system is the real concern that largely impacts the performance. A page-migration system may bring down the latency but has multiple issues and cannot be implemented as such in disaggregated memory. Firstly, hot page migration is difficult in multi-tiered disaggregated systems. Secondly, page access introduces long delays, consumes more bandwidth, and starves block-level accesses. This paper proposes a centralized hot-page migration system that eliminates those issues by accessing remote memory at block granularity, even for page requests. We further reduce the waiting times for both page and regular block accesses by bandwidth partitioning between different types of requests to give them equal opportunities. Our proposed design improves the system performance between 10% to 100% compared to traditional disaggregated systems (page sharing) and 5% to 35% compared to the baseline hardware disaggregated systems.

References

[1] [Online]. Available: https://www.juniper.net/us/en/products/switches/ex-series/ex9200-programmable-network-switch.html

[2] [Online]. Available: https://www.intel.com/content/www/us/en/products-/network-io/programmable-ethernet-switch.html

[3] J. H. Ahn, S. Li, S. O, and N. P. Jouppi, "Mcsima+: A manycore simulator with application-level+ simulation and detailed microarchitecture modeling," in *2013 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2013, pp. 74–85.

[4] A. Akram and L. Sawalha, "A survey of computer architecture simulation techniques and tools," *IEEE Access*, vol. 7, pp. 78 120–78 145, 2019.

[5] 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.

[6] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker, "P4: Programming protocol-independent packet processors," *SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 3, p. 87–95, jul 2014. [Online]. Available: https://doi.org/10.1145/2656877.2656890

[7] R. Cheveresan, M. Ramsay, C. Feucht, and I. Sharapov, "Characteristics of workloads used in high performance and technical computing," in *Proceedings of the 21st Annual International Conference on Supercomputing*, ser. ICS '07. New York, NY, USA: Association for Computing Machinery, 2007, p. 73–82. [Online]. Available: https://doi.org/10.1145/1274971.1274984

[8] S. Chole, A. Fingerhut, S. Ma, A. Sivaraman, S. Vargaftik, A. Berger, G. Mendelson, M. Alizadeh, S.-T. Chuang, I. Keslassy, A. Orda, and T. Edsall, "Drmt: Disaggregated programmable switching," in *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, ser. SIGCOMM '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 1–14. [Online]. Available: https://doi.org/10.1145/3098822.3098823

[9] P. S. Crozier, H. K. Thornquist, R. W. Numrich, A. B. Williams, H. C. Edwards, E. R. Keiter, M. Rajan, J. M. Willenbring, D. W. Doerfler, and M. A. Heroux, "Improving performance via mini-applications." 9 2009. [Online]. Available: https://www.osti.gov/biblio/993908

[10] C. Express. (2023, May) Compute express link: The breakthrough cpu-to-device interconnect cxl. [Online]. Available: https://www.computeexpresslink.org/

[11] C. R. Ferenbaugh, "Pennant: an unstructured mesh mini-app for advanced architecture research," *Concurrency and Computation: Practice and Experience*, vol. 27, no. 17, pp. 4555–4572, 2015. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.3422

[12] P. X. Gao, A. Narayan, S. Karandikar, J. Carreira, S. Han, R. Agarwal, S. Ratnasamy, and S. Shenker, "Network requirements for resource disaggregation," in *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'16, USA, 2016, p. 249–264.

[13] D. Griebler, J. Loff, G. Mencagli, M. Danelutto, and L. G. Fernandes, "Efficient nas benchmark kernels with c++ parallel programming," in *2018 26th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP)*, 2018, pp. 733–740.

[14] J. Gu, Y. Lee, Y. Zhang, M. Chowdhury, and K. G. Shin, "Efficient memory disaggregation with infiniswap," in *Proceedings of the 14th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI'17. USA: USENIX Association, 2017, p. 649–667.

[15] G. Gunow, J. Tramm, B. Forget, K. Smith, and T. He, "SimpleMOC – a performance abstraction for 3D MOC," in *ANS & M&C 2015 - Joint International Conference on Mathematics and Computation (M&C), Supercomputing in Nuclear Applications (SNA) and the Monte Carlo (MC) Method*, 2015.

[16] Z. Guo, Y. Shan, X. Luo, Y. Huang, and Y. Zhang, "Clio: A hardware-software co-designed disaggregated memory system," in *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 417–433. [Online]. Available: https://doi.org/10.1145/3503222.3507762

[17] A. Hassan, H. Vandierendonck, and D. S. Nikolopoulos, "Energy-efficient hybrid dram/nvm main memory," in *2015 International Conference on Parallel Architecture and Compilation (PACT)*, 2015, pp. 492–493.

[18] T. Heo, Y. Wang, W. Cui, J. Huh, and L. Zhang, "Adaptive page migration policy with huge pages in tiered memory systems," *IEEE Transactions on Computers*, vol. 71, no. 1, pp. 53–68, 2022.

[19] M. Islam, S. Adavally, M. Scrbak, and K. Kavi, "On-the-fly page migration and address reconciliation for heterogeneous memory systems," *J. Emerg. Technol. Comput. Syst.*, vol. 16, no. 1, jan 2020. [Online]. Available: https://doi.org/10.1145/3364179

[20] I. Karlin, J. Keasler, and J. R. Neely, "Lulesh 2.0 updates and changes," 7 2013. [Online]. Available: https://www.osti.gov/biblio/1090032

[21] V. R. Kommareddy, S. D. Hammond, C. Hughes, A. Samih, and A. Awad, "Page migration support for disaggregated non-volatile memories," in *Proceedings of the International Symposium on Memory Systems*, ser. MEMSYS '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 417–427. [Online]. Available: https://doi.org/10.1145/3357526.3357543

[22] S.-s. Lee, Y. Yu, Y. Tang, A. Khandelwal, L. Zhong, and A. Bhattacharjee, "Mind: In-network memory management for disaggregated data centers," in *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, ser. SOSP '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 488–504. [Online]. Available: https://doi.org/10.1145/3477132.3483561

[23] 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 *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ser. ISCA '09. New York, NY, USA: Association for Computing Machinery, 2009, p. 267–278. [Online]. Available: https://doi.org/10.1145/1555754.1555789

[24] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: Building customized program analysis tools with dynamic instrumentation," *SIGPLAN Not.*, vol. 40, no. 6, p. 190–200, jun 2005. [Online]. Available: https://doi.org/10.1145/1064978.1065034

[25] H. A. Maruf, H. Wang, A. Dhanotia, J. Weiner, N. Agarwal, P. Bhattacharya, C. Petersen, M. Chowdhury, S. Kanaujia, and P. Chauhan, "Tpp: Transparent page placement for cxl-enabled tiered-memory," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ser. ASPLOS 2023. New York, NY, USA: Association for Computing Machinery, 2023, p. 742–755. [Online]. Available: https://doi.org/10.1145/3582016.3582063

[26] P. Messina, "The exascale computing project," *Computing in Science & Engineering*, vol. 19, no. 3, pp. 63–67, 2017.

[27] R. Mittal, A. Shpiner, A. Panda, E. Zahavi, A. Krishnamurthy, S. Ratnasamy, and S. Shenker, "Revisiting network support for rdma," in *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, ser. SIGCOMM '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 313–326. [Online]. Available: https://doi.org/10.1145/3230543.3230557

[28] D. Mosberger and S. Eranian, *IA-64 Linux Kernel: Design and Implementation*. USA: Prentice Hall PTR, 2001.

[29] N. Niu, F. Fu, B. Yang, Q. Wang, X. Li, F. Lai, and J. Wang, "Pfha: A novel page migration algorithm for hybrid memory embedded systems," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 29, no. 10, pp. 1685–1692, 2021.

[30] C. H. Park, I. Vougioukas, A. Sandberg, and D. Black-Schaffer, "Every walk's a hit: Making page walks single-access cache hits," in *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 128–141. [Online]. Available: https://doi.org/10.1145/3503222.3507718

[31] A. Patke, H. Qiu, S. Jha, S. Venugopal, M. Gazzetti, C. Pinto, Z. Kalbarczyk, and R. Iyer, "Evaluating hardware memory disaggregation under delay and contention," in *2022 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 2022, pp. 1221–1227.

[32] C. Pinto, D. Syrivelis, M. Gazzetti, P. Koutsovasilis, A. Reale, K. Katrinis, and H. P. Hofstee, "Thymesisflow: A software-defined, hw/sw co-designed interconnect stack for rack-scale memory disaggregation," in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2020, pp. 868–880.

[33] J. V. Quiroga, M. Torrents, N. Sonmez, D. Theodoropoulos, F. Zyulkyarov, and M. Nemirovsky, "Evaluation of a rack-scale disaggregated memory prototype for cloud data centers," in *Proceedings of the 30th International Workshop on Rapid System Prototyping (RSP'19)*, ser. RSP '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 15–21. [Online]. Available: https://doi.org/10.1145/3339985.3358496

[34] C. Reiss, A. Tumanov, G. R. Ganger, R. H. Katz, and M. A. Kozuch, "Heterogeneity and dynamicity of clouds at scale: Google trace analysis," in *Proceedings of the Third ACM Symposium on Cloud Computing*, ser. SoCC '12. New York, NY, USA: Association for Computing Machinery, 2012. [Online]. Available: https://doi.org/10.1145/2391229.2391236

[35] T. Repantis, C. Antonopoulos, V. Kalogeraki, and T. Papatheodorou, "Dynamic page migration in software dsm systems," in *2004 IEEE International Conference on Cluster Computing (IEEE Cat. No.04EX935)*, 2004, pp. 494–.

[36] P. Rosenfeld, E. Cooper-Balis, and B. Jacob, "Dramsim2: A cycle accurate memory system simulator," *IEEE Comput. Archit. Lett.*, vol. 10, no. 1, p. 16–19, jan 2011. [Online]. Available: https://doi.org/10.1109/L-CA.2011.4

[37] C. Sakalis, C. Leonardsson, S. Kaxiras, and A. Ros, "Splash-3: A properly synchronized benchmark suite for contemporary research," in *2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2016, pp. 101–111.

[38] Y. Shan, Y. Huang, Y. Chen, and Y. Zhang, "Legoos: A disseminated, distributed os for hardware resource disaggregation," in *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'18. USA: USENIX Association, 2018, p. 69–87.

[39] A. Sivaraman, C. Kim, R. Krishnamoorthy, A. Dixit, and M. Budiu, "Dc.p4: Programming the forwarding plane of a data-center switch," in *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research*, ser. SOSR '15. New York, NY, USA: Association for Computing Machinery, 2015. [Online]. Available: https://doi.org/10.1145/2774993.2775007

[40] Y. Tan, B. Wang, Z. Yan, Q. Deng, X. Chen, and D. Liu, "Uimigrate: Adaptive data migration for hybrid non-volatile memory systems," in *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2019, pp. 860–865.

[41] Y. Tan, B. Wang, Z. Yan, W. Srisa-an, X. Chen, and D. Liu, "Apmigration: Improving performance of hybrid memory performance via an adaptive page migration method," *IEEE Transactions on Parallel and Distributed Systems*, vol. 31, no. 2, pp. 266–278, 2020.

[42] J. R. Tramm, A. R. Siegel, T. Islam, and M. Schulz, "XSBench - the development and verification of a performance abstraction for Monte Carlo reactor analysis," in *PHYSOR 2014 - The Role of Reactor Physics toward a Sustainable Future*, Kyoto, 2014. [Online]. Available: https://www.mcs.anl.gov/papers/P5064-0114.pdf

[43] R. Wang, J. Wang, S. Idreos, M. T. Özsu, and W. G. Aref, "The case for distributed shared-memory databases with rdma-enabled memory disaggregation," *Proc. VLDB Endow.*, vol. 16, no. 1, p. 15–22, sep 2022. [Online]. Available: https://doi.org/10.14778/3561261.3561263

[44] X. Wang, H. Liu, X. Liao, J. Chen, H. Jin, Y. Zhang, L. Zheng, B. He, and S. Jiang, "Supporting superpages and lightweight page migration in hybrid memory systems," *ACM Trans. Archit. Code Optim.*, vol. 16, no. 2, apr 2019. [Online]. Available: https://doi.org/10.1145/3310133

[45] S. Wu, B. Wang, C. Yang, Q. He, and J. Chen, "A hot-page aware hybrid-copy migration method," in *2016 IEEE International Conference on Cloud Engineering (IC2E)*, 2016, pp. 220–221.

[46] Y. Zhang, J. Zhan, J. Yang, W. Jiang, L. Li, L. Zhu, and X. Tang, "Dynamic memory management for hybrid dram-nvm main memory systems," in *2016 13th International Conference on Embedded Software and Systems (ICESS)*, 2016, pp. 148–153.