2021

Improving Performance and Flexibility of Fabric-Attached Memory Systems

Vamsee Reddy Kommareddy
University of Central Florida

IMPROVING PERFORMANCE AND FLEXIBILITY OF FABRIC-ATTACHED MEMORY SYSTEMS

by

VAMSEE REDDY KOMMAREDDY
M.S. University of Central Florida, 2018

A dissertation submitted in partial fulfilment of the requirements
for the degree of Doctor of Philosophy
in the Department of Electrical and Computer Engineering
in the College of Engineering and Computer Science
at the University of Central Florida
Orlando, Florida

Spring Term
2021

Major Professor: Amro Awad

ABSTRACT

As demands for memory-intensive applications continue to grow, the memory capacity of each computing node is expected to grow at a similar pace. In high-performance computing (HPC) systems, the memory capacity per compute node is decided upon the most demanding application that would likely run on such a system, and hence the average capacity per node in future HPC systems is expected to grow significantly. However, diverse applications run on HPC systems with different memory requirements and memory utilization can fluctuate widely from one application to another. Since memory modules are private for a corresponding computing node, a large percentage of the overall memory capacity will likely be underutilized, especially when there are many jobs with small memory footprints. Thus, as HPC systems are moving towards the exascale era, better utilization of memory is strongly desired. Moreover, as new memory technologies come on the market, the flexibility of upgrading memory and system updates becomes a major concern since memory modules are tightly coupled with the computing nodes.

To address these issues, vendors are exploring fabric-attached memories (FAM) systems. In this type of system, resources are decoupled and are maintained independently. Such a design has driven technology providers to develop new protocols, such as cache-coherent interconnects and memory semantic fabrics, to connect various discrete resources and help users leverage advances in-memory technologies to satisfy growing memory and storage demands. Using these new protocols, FAM can be directly attached to a system interconnect and be easily integrated with a variety of processing elements (PEs). Moreover, systems that support FAM can be smoothly upgraded and allow multiple PEs to share the FAM memory pools using well-defined protocols. The sharing of FAM between PEs allows efficient data sharing, improves memory utilization, reduces cost by allowing flexible integration of different PEs and memory modules from several vendors, and makes it easier to upgrade the system.

However, adopting FAM in HPC systems brings in new challenges. Since memory is disaggregated and is accessed through fabric networks, latency in accessing memory (efficiency) is a crucial concern. In addition, quality of service, security from neighbor nodes, coherency, and address translation overhead to access FAM are some of the problems that require rethinking for FAM systems. To this end, we study and discuss various challenges that need to be addressed in FAM systems. Firstly, we developed a simulating environment to mimic and analyze FAM systems. Further, we showcase our work in addressing the challenges to improve the performance and increase the feasibility of such systems; enforcing quality of service, providing page migration support, and enhancing security from malicious neighbor nodes.

ACKNOWLEDGMENTS

I have received immense support and assistance to pursue the degree and throughout this dissertation.

I thank my advisor, Dr. Amro Awad, for his immense belief and trust in me. Your guidance and expertise were precious in addressing and formulating various research questions and solving them. Your support and feedback motivated and drove me in improving my ability to think and solve critical problems.

I would also like to thank Dr. Clayton Hughes and Dr. Simon David Hammond from Sandia-National Laboratories for their collaboration and guidance. I thank my committee members, Dr. Rickard Ewetz, Dr. Zhishan Guo, Dr. Paul Gazzillo, and Dr. Clayton Hughes for serving in my defense panel.

Finally, I would like to thank my family and friends for their constant support and motivation.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1: INTRODUCTION

This chapter discusses the drawbacks of HPC systems with tightly coupled in-node memories (resources) and emphasizes on the need for fabric-attached memory (FAM) systems for HPC workloads.

With the ever increasing demand for larger memory capacities, many high-performance computing (HPC) systems nowadays have their nodes equipped with hundreds of gigabytes of memories. For instance, Oak Ridge National Lab's Summit supercomputer has 512GB of DRAM and 96GB of HBM2 per compute node. The catalyst for increased memory capacity needs per compute node is increasing memory requirements for current new age and emerging workloads like data analytics, graph analytics, machine learning, artificial intelligence et al. However, maintaining such huge memory within a node leads to following drawbacks. Most HPC systems typically run many different applications from a variety of domains, each of which will have its own unique resource requirements; some applications may use the whole memory in the node while others may only use a few gigabytes. Nevertheless, most current HPC schedulers allocate resources at the node granularity, and applications with extremely low memory demands can end up reserving nodes with large memories. Unfortunately, the current approach to choose the size of memory per node is based on the maximum footprint (per node) of the applications of interest, which can lead to significant under-utilization of the memory system. 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 [1, 2, 3, 4, 5, 6, 7, 8]. A recent study shows that about 80% of the jobs on HPC systems overestimate their memory requirements [9]; thus, HPC systems under-utilize memory slots by dedicating them to specific jobs. Moreover, applications that are not able to fit their memory needs into one node incur additional communication overhead because their computation must be split across nodes. Ideally, compute nodes should have direct access to memories that meet their demands without the need to incur expensive software stack overhead due to message passing libraries. As asserted since memory subsystem is mostly constructed with DRAM, considering its fast access rates, power consumption is high. In addition to the under-utilization and sharing overheads of the systems that couple memory with computing nodes, upgrading memory can be challenging. To take advantage of the fast-evolving memories and adapt to the new requirements of applications, system memory should have the ability to be augmented with evolving memory technologies. In systems that deploy petabytes of storage, it is important to be able to flexibly extend the data stores and ensure their robustness. Further, migrating jobs is one of the crucial requirements for hybrid cloud systems [10] and with the existing memory arrangement it is complicated since the data pertaining to the job, which is to be migrated to the target system (node), has to be moved completely.

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, [11, 12], pushing future computing systems to become *memory-centric*. Hence, to mitigate the scalability challenges and data sharing complexities of coupled memory systems, a new memory design direction, decoupled memory systems in which memory is decoupled from the compute engines, is evolving as a result of memory-driven applications [13]. To support such systems, recent standards (consortium's), such as Gen-Z[14], Compute Express Lanes (CXL)[15] and Cache Coherent Interconnect for Accelerators (CCIX)[16], define protocols and interface requirements for accessing memory modules attached to the fast system interconnect. Memory modules that implement memory-semantic protocols and can be readily integrated with the system fabric are typically referred to as FAMs. Protocols defining how to integrate FAMs are being developed through a consortium of major vendors, such as Intel, HPE, AMD, IBM, Lenovo, and VMware[15, 14, 16]. FAMs promise a new HPC architecture where compute nodes can potentially access shared physical memory pools through fast interconnects. In particular, there has been recent industrial interest in architectures where memory modules can be disaggregated from compute nodes, and hence, allows nodes to scale up its memory allocation to the requirements of the workloads run on the node. Such architectures are typically referred to as *memory-centric architectures*. Memory-centric architectures leverage memory semantic protocols to communicate with FAM pools over high-speed interconnects. Owing to memory disaggregation, they promise efficiency, flexibility, and reduced costs. Examples of architectures that resemble memory-centric systems include Facebook's Disaggregated Rack[17], HPE Labs' *The Machine*[13], and Intel's Rack Scale Architecture[18].

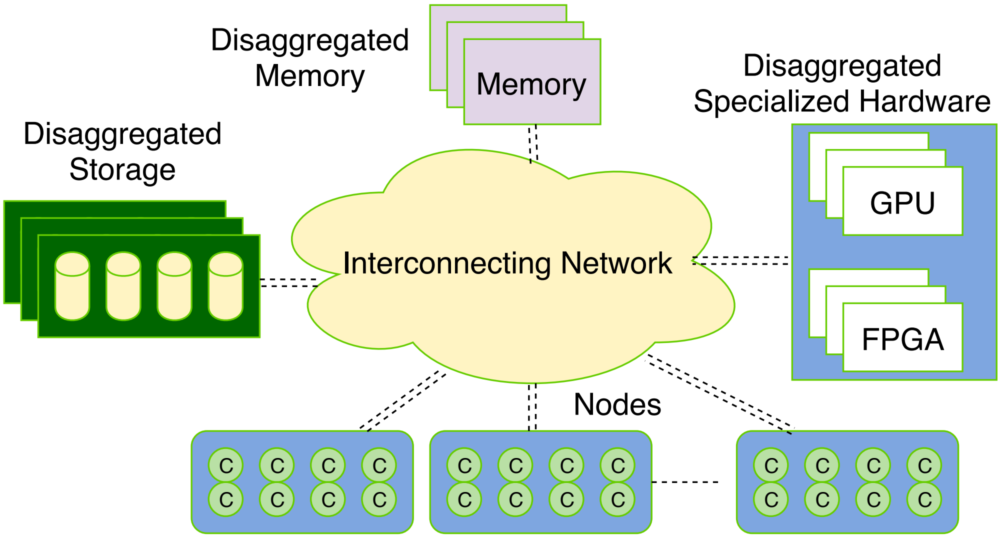

Figure 1.1: Disaggregated memory system

As shown in Figure 1.1, FAM decouples memory from computing nodes (FPGAs, GPUs or system-on-chips (SoCs)). In such systems, the applications can use traditional shared memory interfaces to operate on shared data by utilizing the large shared memory space. Most importantly, the shared memory space can be accessed by traditional load/store operations instead of explicitly communicating between computing 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 (NVM) [19, 20].

NVMs, e.g., Intel's and Micron's 3D XPoint[21, 22, 23] are considered among the best candidates for building FAM systems for several reasons. First, with NVMs expected to have capacities in terabytes per processor socket [24, 25, 26, 27], the under-utilization problem becomes more severe if a processor-centric approach is adopted. Second, NVMs can be used to host directly-accessible filesystems, e.g., Linux's Direct Access for Files (DAX) support [28]. Having NVM modules shared across computing units allows for more efficient operation on shared files. Third, due to the slow and limited endurance of NVM writes, they are expected to be used as an additional layer in the memory hierarchy; having small DRAM-based local memory within nodes while NVM used as a lower-tier memory/storage, attached over fabrics, is a more natural design point [29]. Fourth, when scaling up the total memory capacity of data centers to petabytes, idle power becomes a major concern, and thus DRAM becomes a less practical solution given its significant refresh power and hence high cooling and operational costs.

Memory is maintained in a rack-scaled manner and can be accessed by scores of the compute nodes. Several nodes are connected to the shared memory through a high speed interconnect fabric, e.g., Gen-Z, CXL, and CCIX, [30, 31, 15, 32], are key enablers for disaggregated memory systems, in addition to scalable, dense and ultra-low power memory devices such as emerging NVMs. For instance, Gen-Z[30], is a new interconnect standard that defines a new memory semantic where memory units are directly attached to a system interconnect. In such design, memory becomes the center of the system, where other components, such as accelerators, processor nodes, and SoCs are integrated into the system by interfacing them with a shared interconnect that is attached to the memory.

While accessing memory traditionally, data is copied at multiple locations before storing in memory. For instance, as shown in Figure 1.2(a) data is copied at three places; file system, input output buffers and drivers. Such data copies are not necessary while accessing FAM. This is due to the direct connection between compute nodes to storage class memory with universal memory controllers. This saves both power and also time while maintaining data at multiple locations. Further, with the intervention of software stack in traditional memory access approach, the number of instructions issued by applications is significantly greater than the actual (useful) instructions. Since, PEs can be directly connected to the storage class memory, the number of instructions that are required to access memory now is drastically reduced to only useful instructions [33]. Although, PEs can access memory directly, to connect to the fabric network, PEs, currently, need to understand the underlying protocols provided by the fabric providers to access fabric attached storage class memory. To make PEs or applications oblivious of the underlying protocols, Symmetrical Hierarchical Memory (SHMEM), a PGAS library is used. SHMEM is a point-to-point, one-sided communicative library. Sandia OpenSHMEM (SOS) [34], OpenSHMEM [35] and CraySHMEM [36] are different implementations of SHMEM to communicate with the underlying communicative primitives. OpenFabrics Interfaces (OFI) [37] is a framework which is aimed at exporting fabric communication services or primitives to the upper layers (applications) through SHMEM interface. Libfabric [38] and unified communication X (UCX) [39] are two different framework variants of OFI. These frameworks export user-space application interfaces of the fabric interface to connect to the fabric provider hardware unit.

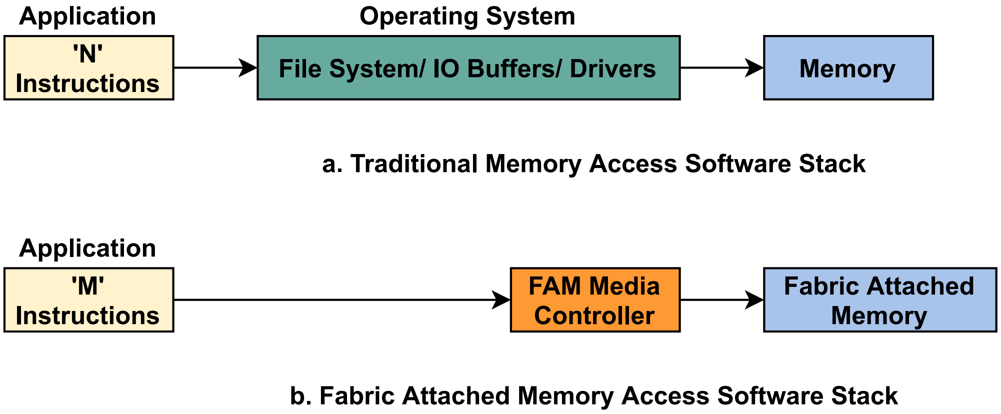
Application
'N' Instructions
Operating System
File System/ IO Buffers/ Drivers
Memory
a. Traditional Memory Access Software Stack
Application
'M' Instructions
FAM Media Controller
Fabric Attached Memory
b. Fabric Attached Memory Access Software Stack

Figure 1.2: Memory access approach in FAM and tranditional memory

Such memory-centric systems promise scalable shared memory applications and significantly reduce communication overhead by relying on shared memory and accessing memory directly with almost zero data copies instead of message passing interfaces between compute nodes and multiple data copies with interference from software stack as discussed. Moreover, disaggregated memory enables fluid division of the shared memory resource among all nodes resulting in a highly scalable system, yet cost efficient. Finally, upgrading and migrating the memory system would require much less efforts. Considering the advantages, many major vendors are considering system designs that utilize FAM, which can be accessed by a large number of processing nodes.

While FAM systems are a promising direction for designing future computing systems, adopting such systems requires rethinking to address various challenges raised with such systems. The drawbacks or challenges while adopting FAM systems are

- *High memory access latency*: Memory is accessed via fabric network interface and hence, memory access latency also includes fabric network latency apart from latency due to various memory types (DRAM or NVM).
- *Security*: FAM (centralized memory) is accessed by number of nodes and without proper memory protection mechanisms, data can be leaked to the neighbour nodes.
- *Fairness*: FAMs are expected to be used in multi-tenant environments and hence, multiple applications from different nodes access FAM. To this end, critical applications need to be prioritized to avoid starvation from applications running in the neighbour nodes .
- *Data coherency*: FAM allows for data sharing between the nodes and hence, data needs to be synchronized between the nodes. Data coherency should be scaled at node-level to achieve data synchronized between the nodes.
- *Memory management*: Remote FAM pages are allocated with the help of global memory

manager[40]. The global memory manager keeps track of the status of the remote memory, for instance, free and allocated pages. However, accessing remote memory manager require modifications to the operating systems (OSes).

Thesis statement: *Our thesis thrives to study a promising use-case for FAMs, which is adopting FAMs in High-Performance Compute (HPC) systems, where the underutilization of memory is a major challenge. For such a use-case, drawbacks like increased memory access latency and security from the neighbour nodes should be addressed. Hence, we address these drawbacks and provide solutions to improve the performance and feasibility of adopting FAMs in HPC systems.*

The summary of our contributions are as follows:

1.1 Fabric-Attached Memory Simulation Setup

To study the impact of decoupling memory and maintaining centralized memory we need a proper simulating environment. FAM system is explored using QEMU [41] by HP laboratories [42]. However, scaling the simulation setup to study number of nodes accessing FAM with QEMU is not practical or takes non-realistic time. Hence, first we design a simulation model for evaluating disaggregated memory architectures using a publicly available simulator, Structural Simulation Toolkit (SST). SST is a cycle-level architectural simulator that is widely used, based on open-source licensing and is publicly available. SST has been proven to be one of the most reliable simulators for large-scale systems due to the scalability and modular design of its components. This makes SST the perfect candidate for simulating disaggregated memory systems at scale. To this end, we developed a new evaluating platform to explore FAM architectures and we explored various memory allocation policies for FAMs. Further details are explained in detailed in chapter 2.

1.2 Investigating Performance by Enforcing Fairness with Hierarchical Priority

As asserted, FAMs 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 FAM increase, 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 FAMs.

We investigate the impact of QoS on application performance when running on FAM architectures. 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 and to propose novel solutions to ensure QoS is enforced. Our simulation results show that by employing proposed hierarchical priority based QoS techniques, a speed up of 55% in accessing FAM is achieved. Hierarchical priority approach and necessary optimizations are discussed in chapter 3

1.3 Page Migration Support

For disaggregated memory systems, it is expected that each computing node would have a small local memory that is based on either HBM or DRAM, whereas a large shared NVM memory would be accessible by all nodes [29]. However, given the contention that results from memory sharing among nodes, proper management of the shared memory resource is a key design requirement. Managing such systems with global and local memory requires a novel hardware/software co-design.

Because off-node memory accesses are expensive, page migration will become a frequent operation on heterogeneous and disaggregated memory systems [43, 44]. During page swapping, physical addresses assigned to the virtual addresses can change, hence page table entry (PTE) update is required. The core initiating a PTE update needs to send Inter-Processor Interrupt (IPI) to other cores to force them to invalidate any copies from the updated PTE on their Translation Look-aside Buffers (TLBs). This process is called TLB shootdown [45]. To reduce the costs of such an interruption, several TLB shootdown optimization algorithms have been proposed [46, 44] Before these systems can be deployed, it will be important to analyze the impact the page migration will have on disaggregated memories. To this extent we provide a detailed page migration support to initiate page migration between global and local memory to maximize performance while enabling access to huge shared memory. Further we investigate such memory management aspects and the major system-level aspects that can affect design decisions in disaggregated NVM systems. Hence, monitoring memory accesses at the shared memory level would require careful design and implementation due to its direct impact on overall system performance. Triggering page migrations between shared memory and local memory would require special handling to ensure invalidating the affected and possibly cached memory mappings on each node. Moreover, identifying when and how often to migrate pages from global memory to local memory is challenging due to many aspects: temporal reuse of page, cost of page migration, network latency and shared memory latency. All of these aspects together should be considered to determine if page migration is useful or not for FAM architectures.

Hence we systematically analyze the impact of various memory management aspects including

TLB shootdowns, page migration latencies, page migration frequency and initiation mechanisms, and global memory latency, on the overall system performance for several applications. Counter-intuitively, we observe that for some applications, accessing remote memory instead of migrating pages to local memory would lead to a better performance. We identify what system configurations and parameters would make page migration more useful. Finally, we propose a novel page migration mechanism that relies on minimal hardware changes to track hot pages at the global memory, where such information can be periodically accessed by system software to initiate page migrations at defined epoch boundaries.

Our evaluation results reveal that for a system with unoptimized TLB shootdown costs, page migration latency, and memory-centric latency, applications do not benefits from page migration, since migrating pages to the relatively fast local memories is amortized by the costs of page migration and TLB shootdown. To mitigate this, we first investigate the impact of optimized TLB shootdown latency to understand to what level TLB shootdown cost is acceptable in disaggregated memory systems, i.e., the point on which such migrations no longer burden memory management. Later, we vary page migration latency and global memory latency to understand their impact on the effectiveness of memory management. Based on these investigations, we propose a novel mechanism that removes much of these overheads from the critical path. Chapter 4 provides a detailed analysis about page migration support in FAM systems.

1.4 Architecture-Aware Virtual Memory Support For Fabric Attached Memory Systems

Since memory-centric architectures leverage FAMs as physically shared memory pools, multiple compute nodes, potentially running applications from different users, can access pages in the same FAM memory modules. This access model is different from conventional HPC architectures where each compute node has its own memory modules and applications' memory accesses are limited to its own nodes, unless explicitly requested from other nodes through software interface. Therefore, a new question arises: who is responsible for access control of FAMs? Without strict access control mechanisms, malicious OSes, applications, and PEs can potentially compromise the entire system by accessing the data of other users in the shared FAMs. Note that in this system architecture, there could be compute nodes containing PEs from different vendors. Even if not malicious, these PEs could contain bugs in their internal virtual memory implementation, which can compromise the whole system. Obviously, with such a wide attack surface, accesses to shared FAM modules need to be vetted externally, at the system-level, and not rely solely on internal access control within PEs. Pages in shared FAM pools can be managed in two different ways. The first approach is through transparently allocating FAM pages to nodes on-demand, i.e., each compute node has the illusion that it has a contiguous large physical space [47]. Such an approach is similar in spirit to how hypervisors give virtual machines (VMs) the illusion that each VM has a contiguous guest physical memory, which eventually gets translated into the real system physical address through the hypervisor. In this case, a memory broker node is dedicated to set up such translations for each node at the system level. The second approach is to expose each node to the real physical addresses (FAM addresses) and modify the OS kernel running on each node to communicate with the external memory broker to allocate FAM pages [47, 48].

Transparent management of FAMs' pages eliminates the need to modify the OS kernel and, most importantly, allows system-level vetting of accesses to FAMs through a second level of memory translation, from the node guest address to the FAM physical address. However, while this is similar to two-level translation in virtualized environments, flexibility, transparency, and security come at the cost of significant performance overheads due to the additional level of translation. In conventional x86 systems, each memory access can require up to four memory accesses for translation, however, when a second level is added, the number of memory accesses can be up to 24 [49]. We observe that significant performance overheads can be incurred when naively implementing state-of-the-art implementations of two-level translation in the context of FAM architectures. Hence, in this paper, we focus on optimizing the implementation of virtual memory support for memory-centric systems.

To minimize the performance overheads of transparent access control and management support for shared FAM pools, we propose an efficient and secure architecture aware virtual memory management support for FAM systems. Chapter 5 studies the existing memory management schemes and discusses out proposed novel memory management approach for FAM systems.

CHAPTER 2: FABRIC-ATTACHED MEMORY SIMULATION

In this chapter, we discuss the implementation of FAM simulation setup in SST. A FAM manager is developed which maintains the status of the global FAM. This FAM manager is used to connect multiple nodes to the global memory.

2.1 Background

Disaggregated memory performance can be emulated on the real systems using device drivers and dividing physical memory to evaluate the remote memory [50, 51, 52]. In this setup, remote memory latency is emulated through a device driver. Unfortunately, relying on real-system emulation restricts the design space exploration to a narrow space that is constrained by the real-system configurations. Trace-driven simulations [53, 54] oversimplify the impact of system-level operations and the out-of-order nature of processing cores and memory systems. They are rarely scalable beyond a few cores, even with very simple memory and processor models. Moreover, it is difficult to model disaggregated memory as it requires multiple nodes to be simulated at the same time. Real-system prototyping takes a significant amount of time and limits the conclusions to the available hardware and software stack, which reduces the flexibility of design exploration. FAM system is explored using QEMU [41] by HP laboratories [42]. However, scaling the simulation setup to study number of nodes accessing FAM with QEMU is not practical (simple) or takes non-realistic time. To the best of our knowledge there is currently no simulation platform that can properly simulate and model disaggregated memory systems. Thus, to facilitate research efforts in disaggregated memory systems, we develop a disaggregated memory emulation environment that takes into consideration many important system-level aspects.

We choose SST to implement FAM system since, SST [55] has been proven to be one of the most reliable simulators for large-scale systems due to the scalability and modular design of its components. Further, SST is a discrete-event simulation model that is modular and easy to customize. This makes SST the perfect candidate for simulating disaggregated memory systems at scale. Implementing a disaggregated memory system design in SST opens up opportunities to explore and examine many challenges. One of the current limitations of SST is the lack of a centralized memory management entity that correctly models page faults and requests for physical frames from the simulated machine. Such a limitation becomes more relevant when there are a large number of shared resources (pools). Thus as an initial step we implemented a centralized memory management entity for disaggregated memory, Opal [56]. Opal is developed to investigate memory allocation policies, page placement, page migration, the impact of TLB shootdown, and other important aspects that are related to managing disaggregated memory systems.

2.2 Opal: A Centralized Memory Manager

Disaggregated memory systems require global memory managers to handle the system shared memory, initiate and broadcast TLB shootdown requests, implement page migration and allow for sharing memory between nodes. To model these aspects of the system, we propose Opal, a centralized memory manager that is implemented as a part of SST to help researchers in studying the functionalities, bottlenecks and optimizations for managing disaggregated memory systems. For the rest of this section, we describe the Opal framework and how it can be utilized to investigate disaggregated memory systems.

Opal can be thought of as the OS memory manager and, in the case of a disaggregated memory system, the system memory allocator/manager. In conventional systems with a single level memory, once a process tries to access a virtual address, a translation is triggered to map the virtual address to a physical address. If a translation is not found, and the hardware realizes that either there is no mapping to that virtual address or the access permissions would be violated, it triggers a page fault that is handled by the OS. The page fault handler maps the virtual page to a physical page that is chosen from a list of free frames (physical pages). Once a physical page is selected, its address is inserted in the page table along with the corresponding access permissions. Any successive accesses to that virtual address will result in a translation process that concludes with obtaining the physical address of the selected page. Since SST aims for fast simulation of HPC systems, it does not model the OS aspects of this sequence of events. However, the memory allocation process will have a major impact on performance for heterogeneous memory systems and disaggregated memory, simply because of the many allocation policies that an OS can select from. Moreover, allocation policies are not well understood on disaggregated memory systems, making it important to investigate them to discover the best algorithm or heuristics to be employed for both performance and energy efficiency. Opal is proposed to fill this role; facilitating fast investigation and exploration of allocation policies in heterogeneous and disaggregated memory systems.

Each component in SST typically represents a subsystem in a real system. SST models a wide range of components such as cores, MMU units, memory hierarchy, routers, and different memory models like DRAM and NVM. Components are ticked according to the component clock frequency set up during configuration. Links are used to communicate between components. Each link can be configured with a latency. We used the Ariel, Samba [57], Messier [58] and Merlin components in SST to simulate CPU cores, MMU unit, NVM memory and network respectively to implement disaggregated memory system design with the help of Opal component.

As shown in Figure 2.1, Opal and the external memory are maintained remotely and each node is connected to Opal and external memory through external links. Processing cores and memory management units are connected to global memory manager, Opal. In our design, we maintained an internal router that helps in communicating between cache and memory components. Likewise, an external router is maintained to connect the internal router with external memory through a network bridge that has its latency modelled after GenZ [30]. This way, communication between nodes and external memory takes place though internal and external routers. To make it realistic, links to external memory is configured with high latency and links to internal memory is configured with low latency.

Processing cores are connected to Opal to pass hints about memory allocations. For instance, calls to `malloc` or `mmap` do not immediately allocate physical pages, but are allocated at the time of mapping, during a page fault. Opal can use hints sent from cores to decide where to allocate the physical page. This is similar to libNUMA malloc hints, which will be recorded and used later by the kernel at the time of on-demand paging. CPU cores can trigger TLB shootdown events to all the other cores, including cores on other nodes. It is cumbersome to create links between each core to send events like TLB shootdown. Hence, we facilitate a communication medium between nodes through Opal. CPU cores communicate with Opal, sending TLB shootdowns events, using a core to Opal link.

The hardware MMU units have links to Opal, so that once a TLB miss and page table walk conclude with a page fault request (unmapped virtual address), a request for physical frame allocation is sent to Opal. Allocation requests come from the page table walker when the accessed virtual page has never been mapped, which resembles the minor page fault and on-demand paging on the first access to virtual pages in real systems. Opal searches for any hints associated with the page fault. If the hints are available, memory is allocated according to the hints from a specific memory region, if not, Opal checks for free frames according to the allocation policies, described in Section 2.2.2, and allocates a frame to the corresponding memory request. Apart from this, during TLB shootdown, Opal sends invalid addresses to all the MMU's through the MMU unit to Opal link and the MMU unit responds with an acknowledge event to Opal after invalidating the addresses.

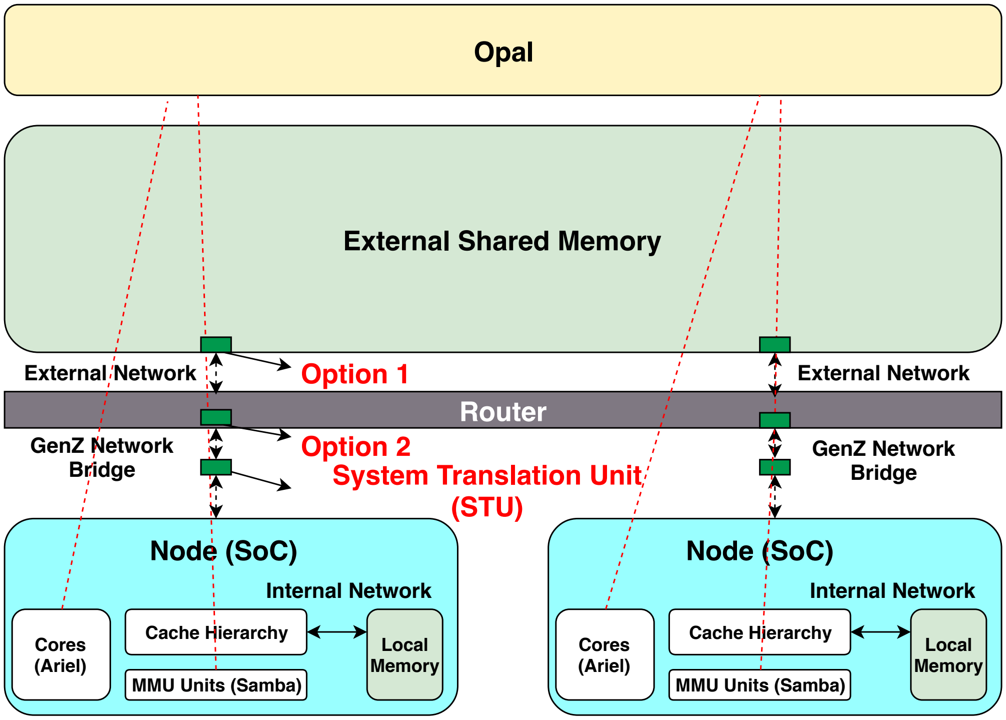

Figure 2.1: A simulated system that uses Opal for centralized memory management.

Hence, Opal must be connected to both a MMU unit, such as Samba, for receiving page fault requests and a processing element, such as Ariel. To allow this, Ariel cores and Samba units should connect to their respective ports in Opal, *coreLink_n* and *mmuLink_n*. For example, *coreLink*_0 port of Opal can be connected to the *opal_link*_0 port of an Ariel core and *mmuLink*_0 port of Opal can be connected to the *ptw_to_opal*0 port of Samba.

Before diving into the details of Opal, we will start with discussing different ways of managing disaggregated memory systems:

2.2.0.1 Exposing External Memory Directly to Local Nodes

In this approach a local node OS (or Virtual Machine) sees both the local memory and external memory, however, it needs to request physical frames from a central memory manager to be able to access external memory legitimately. To enforce access permission, and to achieve isolation between data belonging to different nodes/users, the system must provide a mechanism to validate the mappings and the validity of physical addresses being accessed by each node. To better understand the challenges of this scheme, Figure 2.1 depicts different options to implement access control on shared resources in such management scheme.

As shown in the Figure 2.1, Option 1 would be to check if the requesting node is eligible to access the requested address at the memory module level. This implementation requires a bookkeeping mechanism at the memory module level (or in the memory blade) to check the permission of every access. If the access is valid, then the request will be forwarded to the memory, otherwise either random data is returned or an error packet (access violation) is sent back to the requesting core. Since the external memory is shared between nodes, the system memory manager must have a consistent view of allocated pages and their owning nodes. One way to implement this is through a device driver (part of the local nodes' OS) that can be used to communicate, either through the network or predefined memory regions, with the external memory manager. Option 2 is similar but instead of relegating the permission check to the memory module, the router will have mechanisms to check if the accessed physical addresses are granted to the requesting node. In both options, nodes will not have direct access or modification privileges for permission tables, only the system memory manager will have such access. Such a guarantee can be implemented by encrypting requests with some integrity and freshness verification mechanisms. There are many benefits of these schemes, such as: page table walking process is not modified and it is much faster than virtualized environments (4 steps vs. 26 steps). Also, node-level memory manager optimizations and page migrations are feasible (unlike virtualized environments). But the operating system must be patched with a device driver to communicate with external memory manager and the centralized memory manager becomes a bottleneck if not scalable.

2.2.0.2 Virtualizing External Memory

In this approach, each node has an illusion that it owns all of the system memory. In fact, in this scheme, the OS does not need to be aware of the current state of the actual system physical memory. Figure 2.1 depicts the virtualized system memory scheme.

As shown in Figure 2.1, the system translation unit (STU) must be added to support translation from the *node physical address* to the *system physical address*. The STU can be implemented as an ASIC-based or FPGA-based unit that takes a physical address from the node and translate it into the corresponding system physical address. In case the address has never been accessed, an on-demand request mechanism is initiated by the STU to request system physical page. The STU might need to do a full system page table walk to obtain the node to system translation. Most importantly, the STU can be updated only through the system memory manager. This scheme is better if OS does not need to be changed. But the STU will need to walk the system level page table in addition to walking the node's page table at the node level. Also, there is no guarantee of where the system physical pages that back up the node physical pages exist.

2.2.1 Opal Configuration

Opal should be configured with the component-specific, node-specific and shared memory-specific information as shown in Table 2.1. Component-specific information includes clock frequency, maximum instructions per cycle, etc. Node-specific information includes number of nodes, number of cores per node, clock frequency per node, per node network latency to access the Opal component, node memory allocation policy as explained in section 2.2.2 and local memory information. Shared memory-specific information includes the number of memory pools that shared memory is divided into and the respective memory pool parameters. Both per-node local memory and per-shared memory pool parameters are related to memory and they are explained separately in Table 2.2. Each of these parameters should be appended with memory related parameters as shown in Table 2.1. Table 2.2 describes the memory pool-specific parameters. Each memory pool, whether shared or local, needs a starting address, pool size, frame or page size, and memory technology.

Table 2.1: Opal Parameters

Parameter	Description
clock	frequency of Opal component
max_inst	maximum instructions processed in a cycle.
num_nodes	number of nodes.
node_i_cores	number of cores per node.
node_i_clock	frequency of each node.
node_i_latency	latency to access Opal component per node.
node_i_allocation_policy	memory allocation policy per node.
node_i_memory.	local memory-specific information per node. These come under memory parameters and are shown in Table 2.2
shared_mempools	number of shared memory pools to maintain shared memory.
shared_mem.mempool_i.	global memory-specific information per shared memory pool. These come under memory parameters and are shown in Table 2.2

We show a basic configuration used to test a disaggregated memory system with Opal in Figure 2.2. According to the example configuration, the system has 4 nodes (”*num_nodes*” : 4) with a private memory each and shared global memory is divided into 4 memory pools (”*shared_mempools*” : 4). The private memory uses *DRAM* (”*node*0.*memory.mem_tech*” : 0) technology with a size of

Table 2.2: Memory Pool Parameters

Parameter	Description
start	starting address of the memory pool.
size	size of the memory pool in KB’s.
frame_size	frame size of each frame in memory pool in KB’s. This is equivalent to page size.
mem_tech	memory pool technology $(0 : DRAM, 1 : NVM)$.

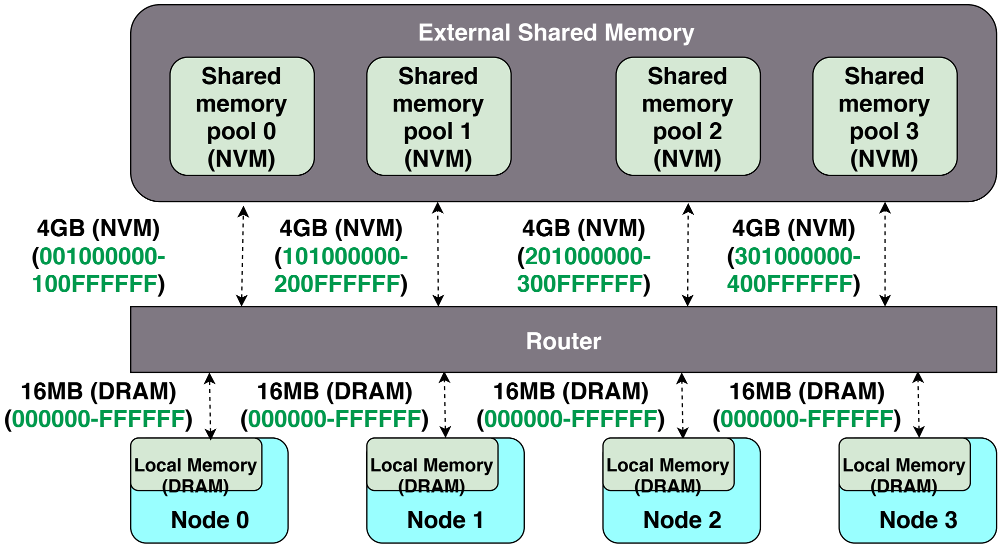
External Shared Memory
Shared memory pool 0 (NVM)
Shared memory pool 1 (NVM)
Shared memory pool 2 (NVM)
Shared memory pool 3 (NVM)
4GB (NVM) (001000000-100FFFFFF)
4GB (NVM) (101000000-200FFFFFF)
4GB (NVM) (201000000-300FFFFFF)
4GB (NVM) (301000000-400FFFFFF)
Router
16MB (DRAM) (000000-FFFFFF)
16MB (DRAM) (000000-FFFFFF)
16MB (DRAM) (000000-FFFFFF)
16MB (DRAM) (000000-FFFFFF)
Local Memory (DRAM)
Node 0
Local Memory (DRAM)
Node 1
Local Memory (DRAM)
Node 2
Local Memory (DRAM)
Node 3

Figure 2.2: Example configuration

16MB (”$node0.memory.size$” : 16384), a starting address of 0 (”$node0.memory.start$” : 0). The total global or shared memory is 16GB, which is divided into 4 memory pools each of 4GB (”$shared\_mem.memp\ ool0.size$” : 4194304). The starting address of shared memory pool 0 is 001000000 (”$shared\_mem.mempool0.start$” : 001000000) which is equivalent to local memory (16MB) + 1; the starting address of memory pool 1 is 101000000 (”$shared\_\ mem.mempool1.start$” : 101000000), which is equal to the starting address of shared memory pool 0 + shared memory pool 0 size. Figure 2.2 shows the starting address of each memory pool, from which the size of each memory pool can be deduced. Each shared memory pool is of NVM type ("$shared\_mem.mempool0$ $.mem\_type$" : 1). Frame size or page size in both DRAM and NVM is $4KB$ ("$node0.memory.frame$ $\_size$" : 4 "$shared\_mem.mempool0.frame\_size$" : 4). Memory allocation policy can be configured using "$node0.allocation\_policy$" parameter, which is explained in the Section 2.2.2. The network latency to communicate with Opal can be configured with "$node3.latency$" parameter. In our case we used 2 micro seconds latency to communicate with Opal.

2.2.2 Memory Allocation Policies

Multiple memory allocation policies are implemented in our design, which are described below.

2.2.2.1 Local Memory First Policy:

Local memory is given more priority than shared memory, that is, memory is searched in local memory and if local memory is full then shared memory is searched for memory. If shared memory is spread into different memory pools, then a shared memory pool is chosen randomly among different memory pools until some space is found. If none of the memory pools are available, that is total memory is full, then an error message is thrown. This memory allocation policy can be chosen by setting "$allocation\_policy$" parameter of a node to 0.

2.2.2.2 Alternate Memory Allocation Policy:

For every two memory requests, one frame is allocated from local memory and the other from shared memory. For example, if two shared memory pools are maintained, first page is allocated from the local memory, second page is allocated from shared memory pool one, third page is allocated from the local memory, fourth page is allocated from the shared memory pool two and so on. This memory allocation policy can be chosen by setting ”*allocation_policy*” parameter of a node to one.

2.2.2.3 *Round Robin Memory Allocation Policy:*

Memory frames are scheduled to be allocated from shared and local memory based on the total number of memory pools, which includes local memory pool of a node and total shared memory pools in a round robin fashion. If two shared memory pools are maintained, then for the 1st memory request, memory is allocated from local memory, for the 2nd memory request, memory is allocated from shared memory pool 1, for the 3rd memory request, memory is allocated from shared memory pool 2, for the 4th memory request, memory is allocated from local memory and so on. This memory allocation policy can be chosen by setting ”*allocation_policy*” parameter of a node to 2.

2.2.2.4 *Proportional Memory Allocation Policy:*

The proportion at which memory frames are allocated from shared and local memory is based on the fraction of local memory size to total shared memory size. For example, if the local memory size is 2GB and shared memory size is of 16GB, then, for the 1st memory allocation request, memory is allocated from local memory while the next 8 memory requests are allocated from shared memory in sequential order. For the next memory request, which is the 10th memory request, memory is allocated from local memory and so forth. This memory allocation policy can be chosen by setting ”*allocation_policy*” parameter of a node to 3.

2.2.3 Communication Between Nodes

Opal also allows nodes to communicate directly with one another by sending hints with the same *fileID* to Opal using Ariel *ariel_mmap_mlm* and *ariel_mlm_malloc* calls. Opal checks if the received *fileID* is registered with any memory. If it is, then the specific page index is sent to the requesting node. If the *fileID* is not registered with any memory page, then memory is allocated based on the requested size. The allocated memory region is now registered with the requester *fileID*. Nodes can share information just by writing information to the specific pages. This reduces costly OpenMPI calls to share information between nodes.

2.3 Evaluation

We validated our design by calculating the performance of the system in-terms of instructions per cycle (IPC). We vary the number of nodes, number of shared memory pools and memory allocation policies. The average number of instructions per cycle is taken into consideration. Simulation parameters and applications that we used along with application parameters are shown in Tables 5.2 and 2.4 respectively.

Table 5.2 depicts simulation parameters for our experiments. According to this, each node has 8 cores and each core can serve up to 2 instructions per cycle. The clock frequency of the cores is 2GHz. Each core is configured to service up to 100 million instructions. Three levels of cache are used, L1, L2, and L3, with sizes of 32KB, 256KB and 16MB respectively and each are of non-inclusive type. Local memory size is 2GB and is of DRAM type. External memory is of NVM type with 16GB size. Network latency is critical in disaggregated memory system. For the external network latency, we use 20*ns* for input and 20*ns* for output buffer latency (in total 40ns latency) which has been modelled after the GenZ network latency.

Table 2.3: Exploring Allocation Policies 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 input and 20ns output latency[59]

Table 2.4: Applications used to Exploring Allocation Policies

Application	Value
XSBench [60]	-s large -t 8
Lulesh [61]	-s 120
SimpleMoC [62]	-t 8 -s
Pennant [63]	leblancbig.pnt
miniFE [64]	-nx 140 -ny 140 -nz 140
NAS:IS [65, 66]	class C

Since our focus is on HPC applications we evaluated our design using 6 HPC mini applications. XSBench [60], a mini-app representing a key computational kernel of the Monte Carlo neutronics application, OpenMC. Lulesh [61], a mini-app for hydrodynamics. Pennant [63] is an unstructured mesh physics mini-app designed for advanced architecture research. SimpleMOC [62], 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 [65, 66] mimics 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. MiniFE [64] is a proxy application for unstructured implicit finite element codes. Applications and their parameters are shown in Table 2.4. We decided upon these specific applications as these are memory intensive.

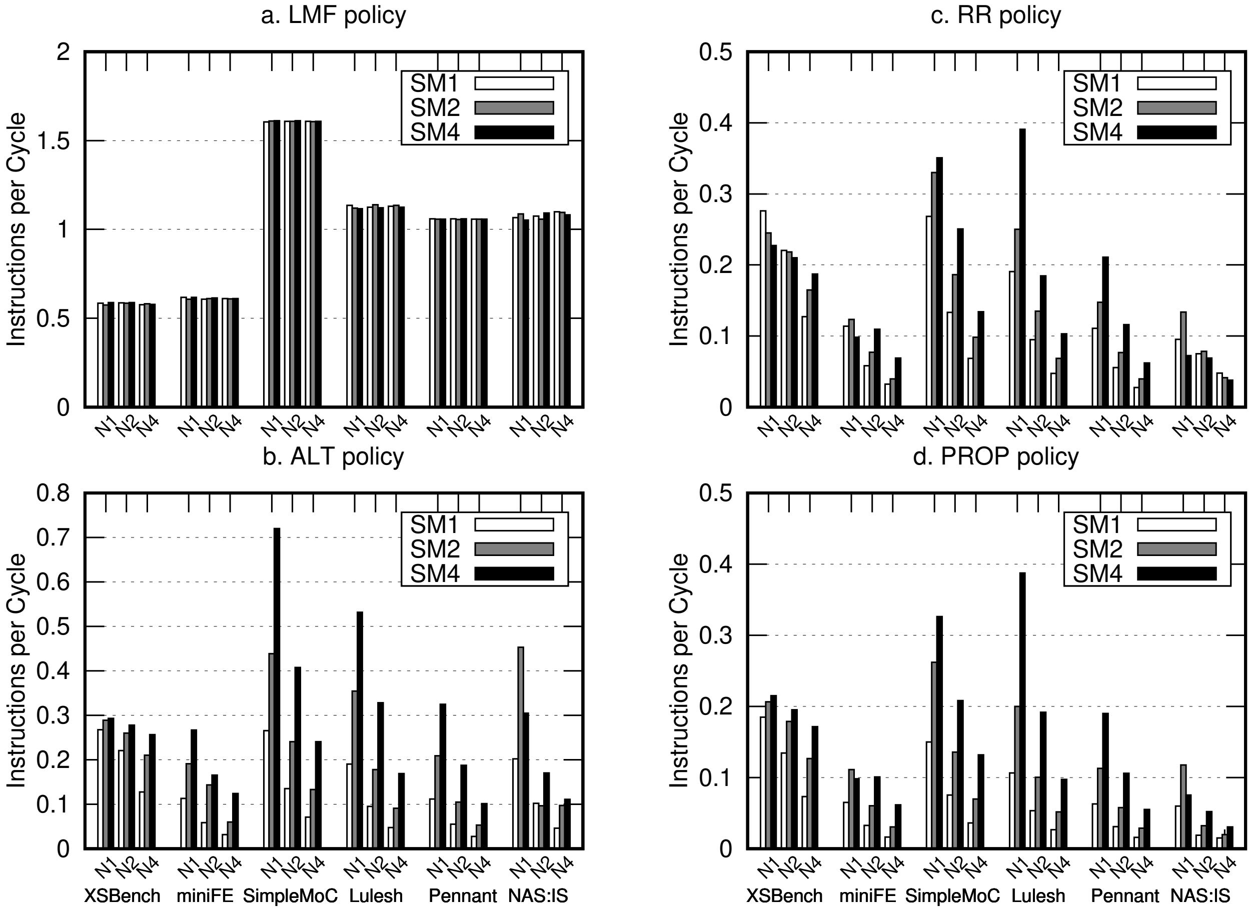

Figure 2.3: Performance in instructions per cycle of disaggregated memory system with different memory allocation policies. *N* indicates number of nodes. *SM* indicates number of shared memory pools. *LMF*, *ALT*, *RR* and *PROP* indicate local memory first, alternate memory, round robin and proportional memory allocation policies.

2.3.1 Opal Memory Allocation Policies

If more memory is allocated from shared memory, the performance of the system worsens as the delay in accessing shared memory is high. Memory allocation policies, explained in section

2.2.2, control allocation of local and shared memory. Contention at shared memory is one of the key factors that contributes to the performance in disaggregated memory systems. The more the contention at the memory, the more will be the delay in accessing memory. Contention at memory is higher if more nodes are accessing memory at a given time. Accordingly, we observed the following traits for each memory allocation policy.

2.3.1.1 Local Memory First (LMF) Policy:

According to the local memory first allocation policy, memory is allocated in private memory first and if there is no space in private memory then memory is allocated from global memory. The benchmark applications that we used occupy a maximum of approximately 500MB of memory to generate 100 million instructions. Because each node has its private memory of 2GB, all of the memory pages should be allocated form local memory and the performance of the nodes should be same as there is no contention at local memory due to other nodes. Our results in Figure 2.3(a) reflect this. Irrespective of the number of nodes and number of shared memory pools the performance of each node, i.e., number of instructions per cycle is equal. We show this to understand the memory intensity of the benchmarks. According to Figure 2.3(a), the IPC of XSBench and MiniFE is around 0.6. Lulesh, Pennant and NAS:IS have an IPC of around 1.1. Wherein the IPC of SimpleMoC is 1.6. From this it can be understood that XSBench and MiniFE are more memory intensive, SimpleMoC is less memory intensive, and Lulesh and Pennant are moderately memory intensive among the set of benchmarks that we experimented with.

2.3.1.2 Alternate (ALT) Policy:

In this memory allocation policy, for every other page fault, a page is allocated from the shared memory. Accordingly, almost half of the pages are from the shared memory, i.e., among 500MB of memory that the applications use, 230MB of memory is from shared memory. From Figure 2.3(a) the IPC of Lulesh is 1.1 while Figure 2.3(b) shows an IPC of 0.2. The performance decreases by 81% when shared memory is used. It further decreases if there are a greater number of nodes accessing the shared memory. For the same benchmark, the IPC is 0.05 when 4 nodes share the external memory. As the number of nodes making use of the shared memory increases, contention at the shared memory increases and the individual node performance decreases. Contention can be reduced by dividing the shared memory into number of memory pools and hence the performance of the system increases. From Figure 2.3(b) it can be seen that for Pennant, the IPC is 0.12 and 0.34 with 1 node when shared memory is maintained in 1 shared memory pool and 4 shared memory pools respectively. With 4 nodes, the IPC is 0.1 when shared memory is maintained in 4 shared memory pools, which is almost equivalent to the performance of the system with 1 node when shared memory is maintained in only 1 shared memory pool.

2.3.1.3 Round Robin (RR) Policy:

Memory is allocated based on the number of shared and local memory pools. The more the number of shared memory pools, the more memory addresses are allocated from the shared memory and the performance decreases. Figure 2.3 shows that, for the 4 node SimpleMoC benchmark with 4 shared memory pools, the IPC is 0.15 for the RR policy and 0.25 for the ALT policy. This is due to more memory is allocated from shared memory in the RR memory allocation policy with 4 shared memory pools. When shared memory is maintained only in 1 memory pool, RR memory allocation policy is same as ALT memory allocation policy. From Figure 2.3(c) it can also be observed that, for some applications, when shared memory is maintained in more shared memory pools, the performance decreases due to more memory being allocated from shared memory. For instance, the IPC of XSBench drops from 0.28 to 0.23 when shared memory is divided into 4 shared memory pools compared to when shared memory is maintained in 1 shared memory pool.

2.3.1.4 Proportional (PROP) Policy:

Memory allocation in based on the proportion of local and shared memory. From the configuration that we used, 16GB of shared memory and 2GB local memory, the proportion at which shared and local memory are allocated is 8:1, i.e., for every 9 memory allocations 8 memory allocations are from shared memory and 1 memory allocation is from local memory. According to this, more memory is allocated from shared memory in comparison with RR and ALT allocation policies. From Figure 2.3(d) it can be clearly observed that, for miniFE, the IPC is 0.06 with 1 node and when shared memory is maintained in only 1 shared memory pool. This is less than the IPC of ALT memory allocation policy and RR memory allocation policy which is around 0.11 each, from Figures 2.3(b) and 2.3(c). As the nodes increased from 1 to 4, the IPC of the system further decreased from 0.06 to 0.02. When shared memory is divided into 4 shared memory pools the IPC of the system increased to 0.06.

We observe that dividing shared memory into more shared memory pools does not always improve the performance of the system. The performance depends on the application characteristics as wellFrom Figure 2.3 it can be seen that for NAS:IS benchmark, for several memory allocation policies, the IPC, when shared memory is maintained in 2 shared memory pools, is more when compared with IPC of the system when shared memory is maintained in 4 shared memory pools with 1 node in the system. We suspect that NAS:IS is latency sensitive and performs better when local memory is used even though it has limited memory-level parallelism. When shared memory is divided into 2 shared memory pools, this can lead to a increase in the number of memory accesses serviced by global memory as in round-robin allocation policy, however, this can also improve the bandwidth and memory-level parallelism. Meanwhile, increasing the number of pools to 4 can lead to performance degradation as the increase in memory access latency due to accessing global memory is no longer amortized by the increase in bandwidth.

2.4 Conclusion

While FAM systems are a useful system design, before fully adopting such architectures, there are a lot of challenging design parameters that must be fully understood such as latency, memory management policies,virtual to physical address translation, page migration, and quality of service. To this end, we proposed a new FAM system simulating model to examine and explore various aspects related to such an architecture. Specifically, we implemented a centralized memory manager in SST which has capability to manage memory in disaggregated memory systems.

CHAPTER 3: INVESTIGATING PERFORMANCE IMPROVEMENTS BY ENFORCING FAIRNESS IN FABRIC-ATTACHED MEMORY SYSTEMS

FAM systems opens a path for designing future computing systems. However, as asserted, memory access latency is a concern in such systems. One of the reasons for the latency is, the global memory is accessed by various applications from a number of nodes. This leads to contention at the global memory not just from the applications from within the node, but from applications from a number of nodes accessing FAM. Using the FAM simulation setup that is explained in chapter 2, Figure 3.1 shows the delay (observed by the core) in accessing memory per request in disaggregated memory systems when the FAM is dedicated to a single node and is shared between four nodes. These results are with respect to system configuration shown in Table 3.1. More details about the methodology and benchmarks are discussed in section 3.3 in detail.

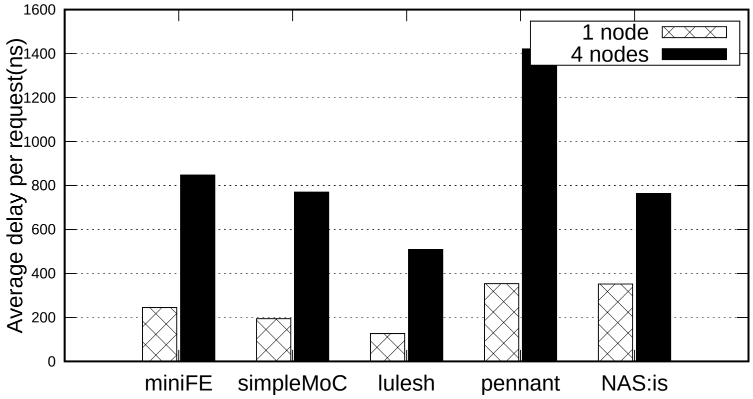

Figure 3.1: Average memory access delay per request observed by the core in dedicated single node FAM system (FAM is accessed by only one node) and shared four node FAM system (FAM is shared by four nodes).

As expected, there is a significant increase in the average memory response time when increasing the number of nodes running on 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 this contention can lead to a multi-x increase in global memory access latency. Pennant benchmark, due to its memory intensity and streaming access pattern, incurs more than 3x increase in access latency when there are four nodes sharing the same memory; from $\approx 380ns$ up to $\approx 1400ns$. Note that row buffer locality can be severely impacted and the chances of a row buffer hit decreases with increasing numbers of memory requests from other nodes. Hence, to improve the response time (overall memory access delay), in this chapter, we enforce fairness (QoS) to the applications from different nodes by modifying the global memory manager. Precisely, we explore novel hierarchical, static and dynamic priority schemes as QoS assurance mechanisms on FAM systems.

3.1 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. [67] implemented a fine-grained QoS scheduling for PCM memory using pre-emption methodologies at the cost of performance. Subramanian et al. [68] worked on designing a model to accurately estimate memory-interference-induced slowdowns. They also proposed a memory scheduler that meets hardware accelerator deadlines while maximizing CPU performance [69]. Jeong et al. [70] proposed a QoS-aware memory controller which can dynamically balance bandwidth between CPUs and GPUs. Zhao et al. [71] proposed a memory control scheme called FIRM, which can fairly run persistent and non-persistent applications. While all of the above-mentioned schemes work for their respective systems, they need to be carefully studied for FAM systems as the contention at the shared memory controller can be quite high. We explore possible solutions that can enhance QoS and study the possible outcomes to improve the performance of disaggregated non-volatile memory system.

3.2 Fairness in Fabric-Attached Memory Systems

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).

3.2.1 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.

3.2.1.1 Static Priority

This is a fixed priority for each node, which is configured during the initialization phase of the application[1]. 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 applications with similar static priorities are placed in the same queue. Note that this is,

[1] Assigning static priorities to nodes is in accordance with service level agreement (SLA).

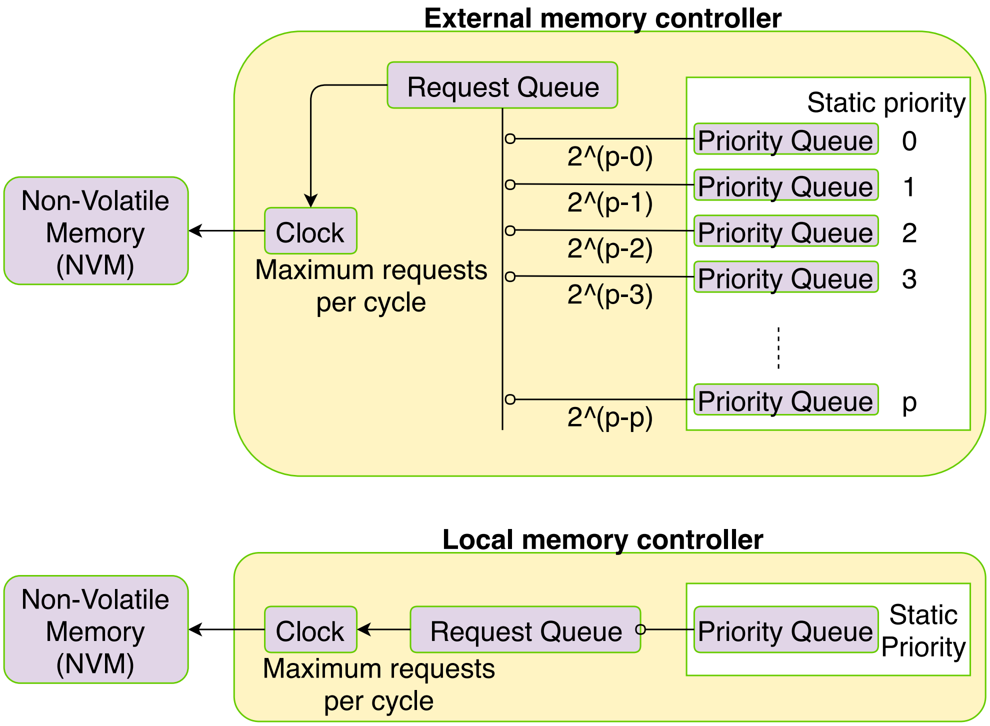

Figure 3.2: Hierarchical priority based QoS implementation in Local and External memory controllers

to some extent, similar to the state-of-the-art storage protocol, NVM Express [72], where multiple I/O submission queues are configured with different priorities but in the context of disaggregated memory systems.

Figure 3.2 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 3.2.1.2, if the node executes multiple applications. Request batches are pulled from priority queues based on their static priority levels. For instance, if we consider 4 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:

$$R_{batch/node} = 2^{p-i} \tag{3.1}$$

3.2.1.2 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 the 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 capacity, memory bandwidth is limited for the applications. Due to this scenario, compute-bound applications need to wait for the memory bandwidth to fetch a small amount of information. This wait can be costly and may 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 few memory requests should have higher priority and applications with more memory requests can have less priority. Therefore, dynamic priority can be expressed as:

$$P = \frac{R_{node}}{AR_{staticpriority}} \tag{3.2}$$

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:

$$AR_{staticpriority} = \frac{TR_{staticpriority}}{N_{staticpriority}} \tag{3.3}$$

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 3.2, 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. If the application has a smaller number of requests in the prior period, the dynamic priority of application will be higher and its requests will not get pre-empted by memory-intensive applications with the same static priority. The dynamic priority can be expressed as:

$$P = \frac{R_{node} * RR_{node}}{AR_{staticpriority}} \tag{3.4}$$

where RR_{node} is the rate of requests per node which is calculated per epoch (time interval).

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. On each 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.

3.2.2 Splitting Shared Memory

Contention at the shared memory increases exponentially with the 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 to high priority nodes. In other words, high priority nodes can have a dedicated memory pool that cannot be accessed by low priority nodes or any other high 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 3.3. 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 this by dividing shared memory proportionally. That is, some of the low priority memory pools are converted to high priority memory pools and 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 managed 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.

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

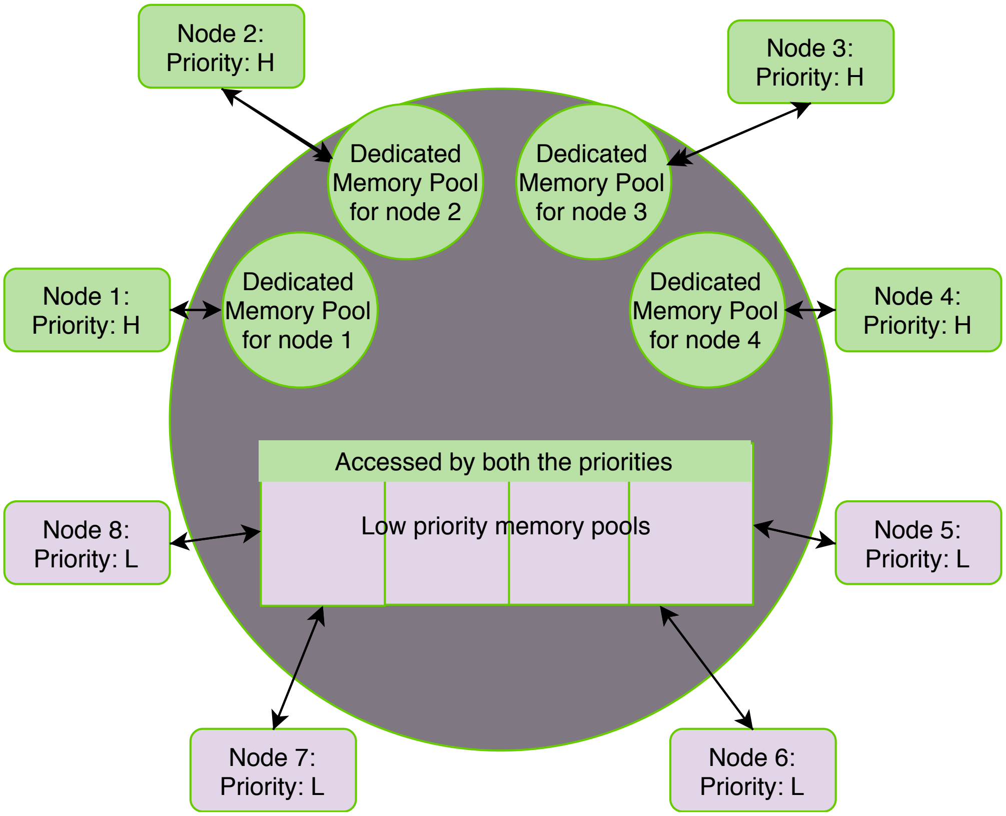

Figure 3.3: Dedicating shared memory pools for high priority nodes.

3.3 Evaluation

To evaluate our QoS support for disaggregated memory systems, we extend SST [55]. We have created an external memory with its respective memory controllers and connected it to compute nodes through a fast network modeled after GenZ [30], as described in [56]. As the local memory

Table 3.1: System Simulation Parameters used to Examine Hierarchical Priority Approach

Element	Parameters
CPU	8 Out-of-Order cores, 2GHz, 2 issues/cycles, 32 max. outstanding requests
L1	private, 64B blocks, 32KB, LRU, inclusive
L2	private, 64B blocks, 256KB, LRU, inclusive
L3	shared, 64B blocks ,16MB, LRU, inclusive
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	40ns[59]

is expected to be very small in such systems [13], we deploy an alternating memory allocation policy [56], where memory is allocated alternatively from shared and local memory. The modules that we used to simulate disaggregated memory system are described in [56].

Table 3.1 lists the simulation parameters used to evaluate the design. L1, L2 and L3 caches are non-inclusive and are of sizes 32KB, 256KB and 16MB respectively. We used NVM as shared memory and, considering the density of NVM, held the NVM size to be twice the size of the local memory, per node. We understand that NVM density is much higher when compared to DRAM. Future disaggregated systems with higher densities can also benefit from our QoS approach. As explained in section 3.3.2, 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.

For the simulation model, we use the alternate memory allocation policy in which memory is allocated alternatively from shared and local memories. 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 are confined to maximum of 4 memory pools.

Considering that our focus is on HPC applications in disaggregated memory environment, we chose 5 memory intensive HPC proxy applications to evaluate our design. Lulesh [61], a mini-app for hydrodynamics. Pennant [63] is an unstructured mesh physics mini-app designed for advanced architecture research. SimpleMOC [62], 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 [65, 66] mimic the computation and data movement characteristics of large scale computational fluid dynamics (CFD) applications. IS is an integer sort kernel that performs a sorting operation, which is important in particle method codes. MiniFE [64] is a proxy application for unstructured implicit finite element codes.

For the rest of the evaluation, N indicates the number of nodes. SM indicates a node has local and shared memory and includes an identifier representing the number of memory pools. 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 3.2. *noqos* indicates experiments without any QoS. *hp* indicates experiments with hierarchical priority. Applications running in each node is expressed as $a-n$ wherein n indicates node number.

Table 3.2: Applications-Mixes used to Examine Hierarchical Priority Approach

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

3.3.1 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.

Figure 3.4 shows the impact of using multiple shared memory pools along with the resulting performance degradation under the multiple nodes scenario without any priority. For instance, for *Pennant*, RRT is 3x times with 4 nodes due to heavy contention at the single shared memory pool from other nodes, Figure 3.4.

Note that 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 configuration. 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

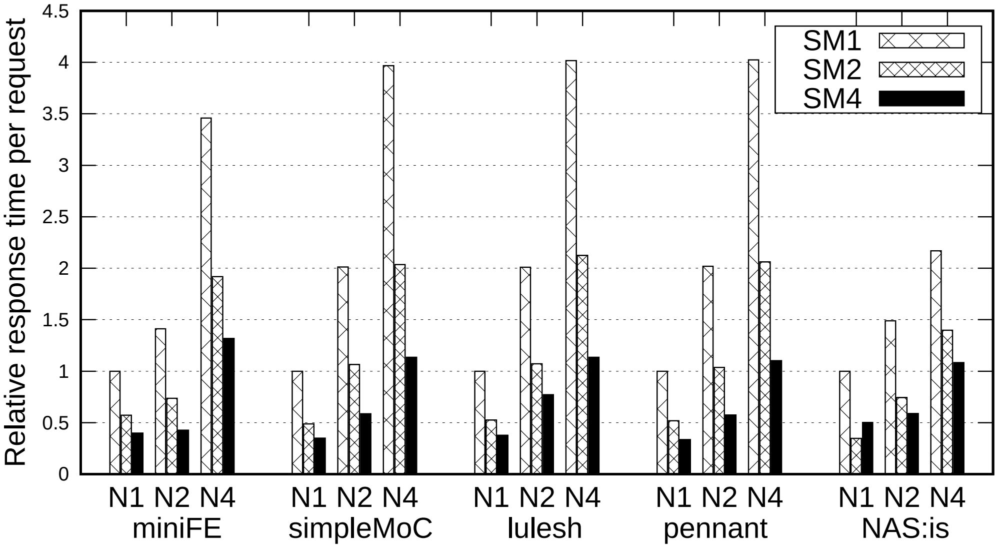

Figure 3.4: Relative response time per memory request of disaggregated memory system model.

in the system with one shared memory pool. Therefore, contention due to multiple nodes can be reduced with 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.

3.3.2 QoS using Hierarchical Priority

As the focus 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 3.2, 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. The 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).

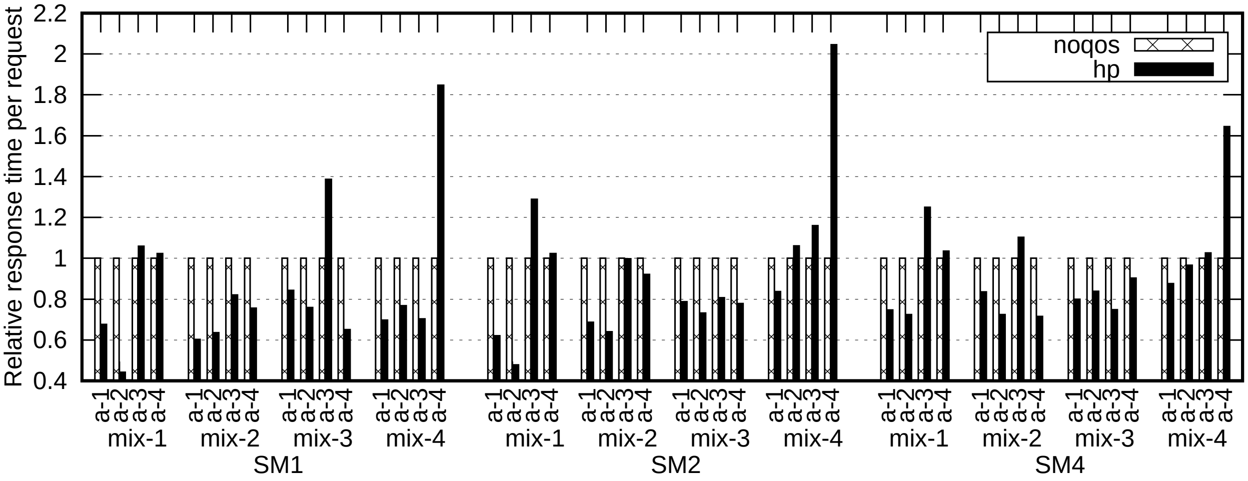

Figure 3.5: Relative response time per request wherein global memory is divided into shared memory pools in disaggregated memory system.

Figure 3.5 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 3.5. 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.

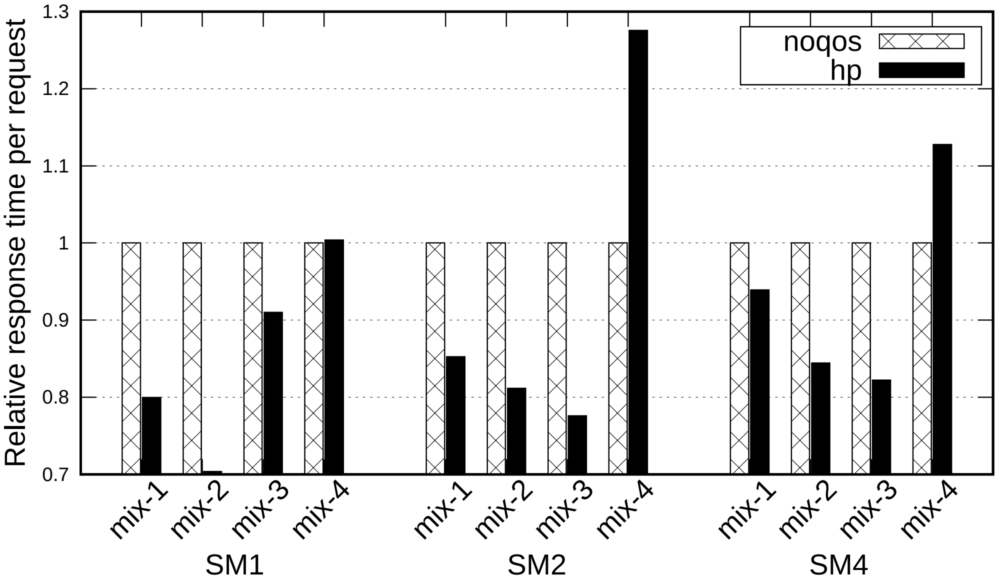

Figure 3.6: Overall relative response time per request of disaggregated memory system using no QoS and hierarchical priority based QoS.

We evaluated our design by dividing shared memory into 2 and 4 shared memory pools, shown in Figure 3.5. We observed a similar pattern as when using multiple shared memory pools but, due to more bank parallelism, the performance of the system is better, as shown in Figure 3.4. 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 Figure 3.5.

The overall system performance in terms of RRT per memory request is shown in Figure 3.6. We could observe an overall maximum performance improvement of 30% using single shared memory with *mix-2*.

3.4 Conclusion

Considering that the memory access latency is impacted by the applications running in the neighbour nodes sharing the same FAM, in this chapter, we study potential improvements that can be achieved by enforcing fairness (QoS) to the nodes. We observe that QoS is a crucial aspect in FAM systems and is inversely proportional to the number of nodes sharing FAM. To this end, we proposed hierarchical priority, a combination of static and dynamic priority-based QoS for FAM 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 propose dividing the shared memory into pools and dedicate FAM pools for specific nodes. However, dedicating shared memory pools to specific nodes reduces the overall memory per node and does not show the anticipated performance gains due to less contention at 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.

CHAPTER 4: PAGE MIGRATION SUPPORT FOR FABRIC-ATTACHED MEMORY SYSTEMS

As discussed in chapter 2, many major vendors are considering FAM system designs by retaining a portion of memory on the node instead of full memory disaggregation for fast processing [29]. This leaves us an opportunity to migrate memory pages from FAM to the local memory to improve overall memory access latency. Hence, in this chapter we provide page migration support and analyze the performance improvements with page migrations.

4.1 Motivation: Impact of Number of Nodes Accessing FAM on Performance

Figure 4.1 shows performance of individual nodes in disaggregated memory systems and traditional systems, with NVM as memory for a range of HPC-relevant benchmarks (see Section 4.4 for additional discussion). As the number of nodes accessing global memory increase, the performance of the applications decreases due to more contention at global memory. We consider performance when global memory is not shared among multiple nodes (Disag-NVM-N1) as the optimal case (no contention) in disaggregated systems. As expected, the performance worsens when the number of nodes sharing global memory increases, e.g., 4 and 8 nodes. For instance, for a memory intensive workload like Lulesh, we can notice a decrease in the performance by 83.8% with 8 nodes normalized to no contention case. Meanwhile, some other workloads, which are less memory sensitive, e.g., NAS IS, encounter 39% slowdown when moving from 1 node to 8 nodes sharing the same memory module. As shown in Figure 4.1, the memory response time increases differently for each workload, mainly due to the variable levels of memory request rate and contention.

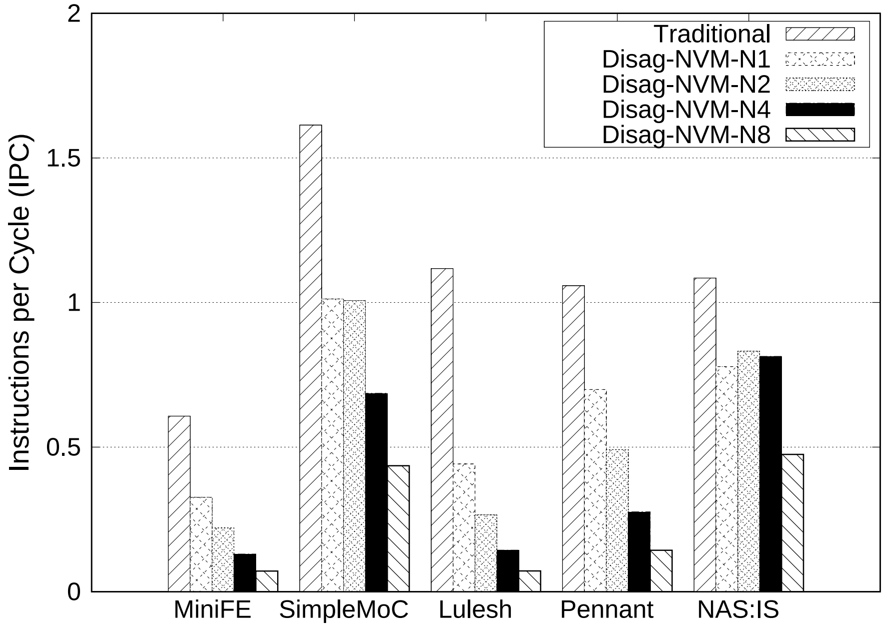

Figure 4.1: Average performance of traditional (non-FAM) memory system and disaggregated memory systems with one to eight nodes accessing a single FAM module.

4.2 Page Migration

Moving frequently accessed pages from global memory to local memory would benefit both node and system overall performance, due to the reduced contention at the global memory and the fast response time of local memory. However, as system-level management deals with each node's physical page mappings, it is an obvious task for system-level management to dynamically migrate hot pages to the local memories accessing them more frequently. While naively we can assume such pages can be identified at the start time of application and then migrated to local memory, the reality is that many challenges and application behaviors typically render such simple solutions ineffective. In particular, the dynamic nature and time-variable pages popularity (access frequency) render static solutions irrelevant. Moreover, given the limited capacity of local memory, an accurate and dynamic profiling of page behavior is needed, otherwise more popular pages may get evicted from local memory.

4.2.1 Translation Look-Aside Buffer (TLB) Shootdown:

TLBs are fast caches that hold page table translations within each core. However, TLB coherence with the full system page table must be maintained explicitly through inter-core interrupts and explicit invalidation commands. When performing page migration, processor cores have to be stalled to invalidate page table entries in respective TLB structures [73, 45, 46]. The process of clearing stable page entries is called TLB shootdown [44, 74] and can become an expensive operation.

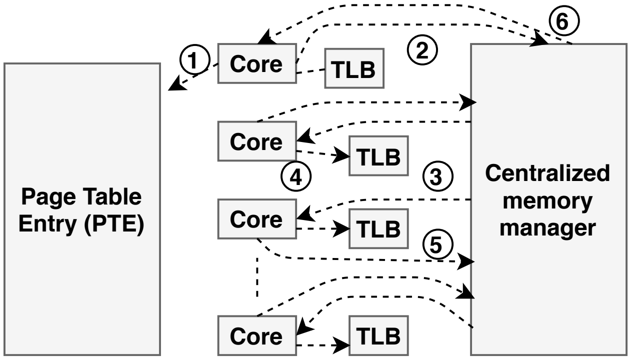

Figure 4.2: TLB shootdown process

Figure 4.2 shows the process of TLB shootdown in disaggregated memory systems. The core, which invalidates the addresses, will update the PTE (1) and send a TLB shootdown request to the centralized memory manager (2). Centralized memory manager stalls all the other cores (3) and the stalled cores invalidate the copies of page table entry in their respective TLBs (4). After invalidating the TLB entries, the stalled cores send an acknowledgement request to the centralized memory manager and resume (5). When the centralized memory manager receives shootdown acknowledgements from all the cores, it resumes the shootdown of the initiating core (6). The entire TLB shootdown procedure consumes around 8us, on an average, in 8 core computing system [75].

4.2.2 Literature

A large body of prior art has investigated page migration in hybrid memory systems which proposes efficient way to migrate pages. Ramos et al. [76] proposed a multi-queue based approach to define the hotness and coldness of the pages. Wang [77] managed NVM at a super-page granularity by using a lightweight page migration. Wang also considered the utility of the migrating page. Yoon et al. [78] devised a policy that enables DRAM to cache pages which has high frequency of row buffer misses in NVM memory. CAMEO [79], PoM [80], Mempod [81] and BATMAN [82] discuses about the granularity and relaxations possible while swapping pages to maximize overall memory bandwidth. Other approaches [83, 84] involve both hardware and software. OS is utilized to identify hotness of the page. Page migration is explored in NUMA architectures [85]. These approaches depend on either compiler support or Linux kernel and leverage on counters for number of pages accesses. Lim et al. [86] proposed software-based prototype by extending the Xen hypervisor to emulate a disaggregated memory design wherein remote pages are swapped with local memory on-demand upon access, first touch policy. They also explored round-robin, clock and content based page placement policies to effectively manage the memory. Specifically content-based approach can be effectively utilized for page sharing.

Given the contention that results from memory sharing among nodes, proper management of the shared memory resource is a key design requirement. Unfortunately, the current literature lacks any detailed study that investigates the system-level aspects and memory management impact on disaggregated memory systems. Further, most of the previous schemes perform page swapping at a predefined time intervals and does not take time interval variation into consideration. Also, Lim et al. [86, 47] implemented disaggregated memory design on Xen hypervisor. Thus, we focus on devising innovating page migrating mechanisms that enable efficient memory management schemes in disaggregated memory systems to improve the overall performance. The major difference between prior work and our work is that in disaggregated memory systems, memory management decisions should occur at the *system* level and account for fabric network latency, global memory contentions, global memory latency and the necessary updates of system-level memory mappings. Such considerations pose a challenge on where and how to implement the memory management. For instance, should memory placement be handled by a global memory controller? How aggressively should we perform page migrations? How many pages should be migrated during each epoch, and, what page migration costs would still render page migration useful? While the objective is the same in any heterogeneous memory system work: *migrate hot pages to the faster memory*, the design aspects and usefulness of page migration strictly depend on the system architecture, memory architecture and memory technologies. Thus, designing and implementing memory management in disaggregated memory systems has its own unique challenges, conclusions and design guidelines. Moreover, most of the prior work either conclude their results based on trace-driven or analytical models [43, 44], or use real-system profiling where additional latency is added on each page fault [87], and, hence, are inapplicable to systems where global memory is directly accessible, i.e., not like a swap device. In contrast we use and provide a detailed cycle-level simulation model, which has been integrated with a best-of-class open-source simulation framework and is able to replicate the significant detail associated with the major system-level aspects that affect memory-management decisions.

4.3 Page Migration in FAM systems

In this section, we discuss our proposed memory management support for disaggregated NVM memory systems. Our scheme relies on page migration as a mechanism to enable more efficient page locality and data proximity to their most-accessing compute nodes. To design our memory management support, we start with identifying the answers for the following questions: (1) Which pages to migrate to local memory? (2) Which pages to select as victim pages, i.e., be evicted from local memory?

4.3.1 Detecting Hot and Victim Pages

Pages that needs to be migrated between the two levels of memory should be chosen carefully. Wrong selection of pages would degrade the performance of the application intensely. For instance, if a page in the local memory is accessed frequently and is migrated to the global memory, during the page migration process, the number of cycles to fetch the data of that page would be more. Apart from increasing the number of cycles to access frequently accessed data, a number of cycles would be wasted due to TLB shootdown. Hence efficient page selection algorithms to migrate pages are required to improve the overall performance of the system. For detecting hot pages, we leverage a counter-based scheme, however, we use clock-based replacement policy to select victim pages, as discussed below.

Page Insertion: Page insertion techniques are used to detect hot pages in the global memory and migrate them to the local memory. We leverage on counter-based scheme to select the pages to insert in the local memory since counter based scheme is simple and accurate. In this scheme every page access is accounted and during the process of page migration, the page counters are traversed to find out the most frequently accessed pages. Page accesses are stored in page access count table (PACT) as shown in Figure 4.3. This policy, as it seems, is simple, but there are two important overheads that should be considered. *Hardware Requirements:* As each page requires a separate counter, the number of counters needed would be more for systems that has significant amount of memory, specifically in disaggregated memory systems. This needs tremendous amount of additional hardware. A solution to overcome this is to maintain a cache of counters in the global memory controller and a counter is fetched from the memory during the cache miss. *Traversal Delay:* The page counters, either maintained as hardware counters or as a cache has to be traversed to find out the most and least frequently accessed pages. The number of entries in PACT will be significantly high for the systems with huge memory and a moderate page size (4KB). If the number of entries in PACT is huge, it takes a while to traverse the table to find out the most frequently accessed pages. Although page accounting and page selection process for migration can be performed at the background, while the data is fetched from the memory, eventually it is a costly operation to traverse PACT. We address these two concerns by fixing the PACT size and replacing the least frequently accessed page with in PACT to make space for the new page and storing the least frequently accessed page counter data in the memory. This eliminates huge hardware requirements and reduces the delay in traversing PACT.

In disaggregated memory systems, multiple nodes access the global memory. Hence global memory should have the ability to distinguish requests from different nodes. The memory controller achieves this by extracting the node number from the request packet. In Figure 4.3, *B* is termed as hot page detector. Hot page detection is performed in three steps: 1) During serving a request the page number is extracted from the base address and the page count is incremented in PACT if the page entry is found. If the page entry is not found in PACT, an entry is created and is then incremented. 2) If the page counter is higher than the page migration threshold limit the page is copied to the *pages to be migrated table per node* (PMTn). Global memory controller has to maintain PMT for every node since each node will have different pages to migrate.

Page Eviction: Page eviction technique is used to detect victim pages to be migrated to the global memory from the local memory. Counter based eviction policy is widely used as a page replacement policy to migrate pages between main memory and the secondary memory. Least recently used pages are selected and are moved to the secondary memory. We extend this scheme for selecting least recently used local pages (victim pages) to be migrated to the global memory. In clock based page selection policy each page is referenced, *A* in Figure 4.3. But unlike counter based method, this policy maintains per page reference bit in the page table rather than a counter which requires 32 or 64 bits per page. If the reference bit of the page is set then it is considered as a page which is accessed recently and vise verse. Initially all the page references are reset and for every page access the reference bit is set to indicate that it is accessed recently. A reference pointer is utilized to traverse the page table to select the victim pages by verifying the page reference bit. The reference pointer traverses the page reference table until it finds a page that is not accessed recently. While traversing the page table, the page reference which is set is reset by the reference pointer and the traversal continues. To lessen the delay in finding out the victim page the page table is traversed until specific entries (200 in our case) and once it is reached, the victim page is chosen as the page pointed out by the reference pointer by default. Selecting victim pages is triggered by victim page eviction handler, *D* from Figure 4.3, during page migration epoch.

4.3.2 Performing Page Migration

Centralized memory manager is required to maintain and allocate decoupled centralized memory. We used Opal [56] from SST[55] as a centralized memory manager. Opal is responsible for allocating memory to all the nodes without any conflicts. We extend Opal to perform page migration. For every *page_migration_epoch*, Opal communicates with the global memory controller and individual nodes to fetch pages that needs to migrate. Hot page insertion handler, *C* in Figure 4.3, fetches addresses of hot pages from the global memory controller. The global memory controller

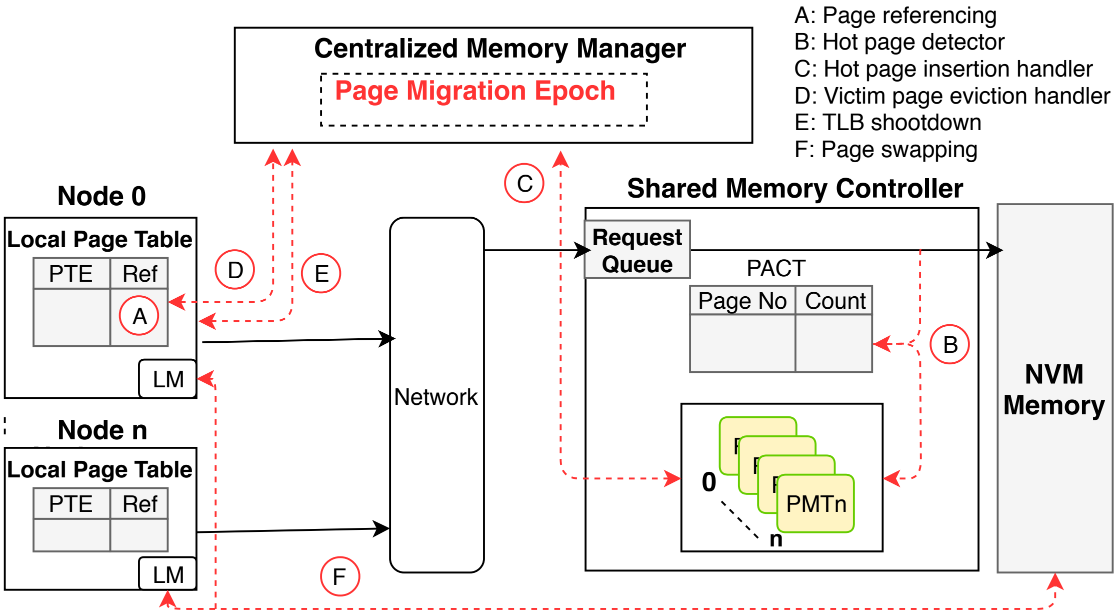

Figure 4.3: Page migration in disaggregated memory systems. LM: Local Memory, PMTn: Pages to Migrate Table per node, PACT: Page Access Count Table

which already segregated pages to be migrated during hot page detection, sends the respective page addresses (page numbers), if any, to Opal after sorting the PMTn table to migrate the most frequently accessed pages first and then clears PACT and PMT to collect page counts for the next epoch interval. Victim page eviction handler, *D* from Figure 4.3, fetches local memory pages to be moved to the global memory, with the help of clock based eviction method. Once both hot and victim pages are fetched by Opal, TLB invalidation and TLB shootdown events along with pages addresses to be remapped are sent to the respective nodes involved in page migration, *E* from Figure 4.3. Nodes which does not have any pages to migrate are not interrupted. The page contents are swapped by Direct Memory Access (DMA), *F* from Figure 4.3, during TLB shootdown.

4.3.3 Sequence of Events

The sequence of events are as follows: a) For every memory request pages are referenced at either the local node or at the global memory controller. If the memory request is to the local memory then the respective page is referenced in the page table of the local memory manager (A from Figure 4.3). If the memory request is to the global memory then the page access is counted at the global memory controller and hot pages are detected and stored in PMTn (*B* from Figure 4.3). b) Centralized memory manager, Opal, for every specific time interval, for instance 1M clock cycles, triggers the global memory and individual nodes to fetch hot and cold pages (*C* and *D* from Figure 4.3). c) Once Opal receives page addresses that has to be migrated, it triggers DMA to swap pages between global and local memory while TLB shootdown and TLB invalidation events are initiated to only those nodes which are involved in page migration (*E* and *F* from Figure 4.3).

Page migration does not always better the system. This depends on factors like frequency of page migration, number of pages migrating at a time, page migration threshold, TLB shootdown latency and page swapping delay.

It is also crucial to interpret at what frequency the page migration process should be performed. If the page migration is performed too often then most of the cycles would be wasted in migrating pages and TLB shootdowns. If the page migration is performed rarely then the advantage in migrating pages is lost since most frequently accessed pages would be migrated to local memory rarely during the lifetime of the application. Conventionally, various page migration schemes for different architectures talk about migrating one page during page migration process, while it is possible to migrate multiple pages at a time. The delay in migrating multiple pages is less compared to migrating single page. For instance, if we consider a $1us$ delay to swap contents of one page and if we assume a delay of $8us$ to invalidate TLB entries and update page table entry, the total delay to migrate one page is $9us$. If pages are migrated one at a time then every time a single page is migrated, the computing units have to wait for 9*us*. If more pages are migrated at a time then only the page swapping delay (1*us*) per page will be added, which is effective. Page migration threshold is an other factor which has an impact on pages to migrate. If the migration threshold is less then the pages with very less accesses to global memory would be eligible to migrate. Also if the migration threshold is pretty high then the pages with very high accesses to global memory would not be eligible to migrate, which results in less page migrations. During TLB shootdown and page swapping, computing units are stalled and are not allowed to proceed util all the pages are swapped and all the TLB levels are invalidated. Hence it must be understood that it is vital to study the impact of these factors on page migration in disaggregated memory systems.

4.3.4 Overheads

There are mainly three overheads associated with our design: 1) Hardware overhead: Global memory controller requires additional hardware for PACT and PMT. This overhead is minimal since PACT and PMT are fixed based on the number of pages to migrate. If we assume 100 pages to migrate at a time then each PMT (PMTn) should have a minimum of 100 entries and PACT should store a minimum of 800 entries, considering 8 nodes system. 2) Accounting overhead: Each page has to be accounted at the global memory whenever the page is accessed, which is in the critical path. This can be avoided from the critical path by performing such accounting in the background. 3) Page address transfer overhead: During the page migration, metadata like page addresses, are exchanged between centralized memory manager and global memory controller or local memory management units. This overhead is minimal since statistical data transfer would happen only during the page migration epoch and if the page migration epoch interval is high then the statistical data transfer would be minimum.

4.4 Evaluation

To study page migration aspects in disaggregated memory, we used a model of a disaggregated system developed in SST [55]. SST has been proven to be one of the most reliable simulators for large-scale systems due to the scalability and modular design of its components. SST includes multiple (swappable) simulation modules for various components. A module called *Opal* [56] has been developed in SST to simulate centralized memory manager for disaggregated memory model.

Table 4.1: System Simulation Parameters to Analyse Page Migration Support for FAM Architectures

Element	**Parameters**
CPU	2 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	256MB, 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	40ns

Table 4.2: Applications used Analyse Page Migration Support for FAM Architectures

Application	**Value**
Lulesh [61]	-s 120
SimpleMoC [62]	-t 2 -s
Pennant [63]	leblancbig.pnt
miniFE [64]	-nx 140 -ny 140 -nz 140
NAS:IS [65]	class C

We simulated disaggregated memory system with 8 nodes. All the nodes run simultaneously with each node hosting a benchmark. Simulation parameters of our simulation environment are shown in Table 4.1. According to the table, 2 cores are used for each node and each core can serve up to 2 instructions per cycle. The clock frequency of the cores is 2GHz, with each core configured to serve up to 100 million instructions of application execution during its HPC-relevant kernels. Three levels of cache are used, L1, L2, and L3, with sizes 32KB, 256KB and 16MB respectively and cache type is non-inclusive. Local memory is 256MB of DRAM memory on each node. Centralized memory is of an NVM type and configured to be 16GB, based on density compared to DRAM and number of nodes. Network latency is critical in disaggregated memory system. External network latency is 40*ns*, which has been modelled after public projections for a GenZ-enabled network.

Since our focus is on HPC applications we evaluated our design using 5 HPC-relevant mini-applications and benchmarks. Lulesh [61], a mini-app for unstructured hydrodynamics, Pennant [63] is an unstructured mesh physics mini-app designed for advanced architecture research, SimpleMoC [62] is a mini-app to demonstrate the performance characteristics and viability of the Method of Characteristics (MOC) in 3D neutron transport calculations in the context of full scale light water reactor simulation. The IS benchmark from the NASA Parallel Benchmark collection [65] is an integer sort kernel which performs efficient large-scale sorting operations. Finally, MiniFE [64] is a proxy application for unstructured implicit finite element codes. Applications along with their parameters are shown in Table 4.2. We decided upon these specific applications as these are known to provide memory accesses that are characteristic of larger codes. In most cases, their access patterns are memory-access intensive although the range of these accesses which is satisfied by caching, or accesses to memory, will vary by each kernel. Note that for our discussion of the following experiments, *N* indicates number of compute nodes.

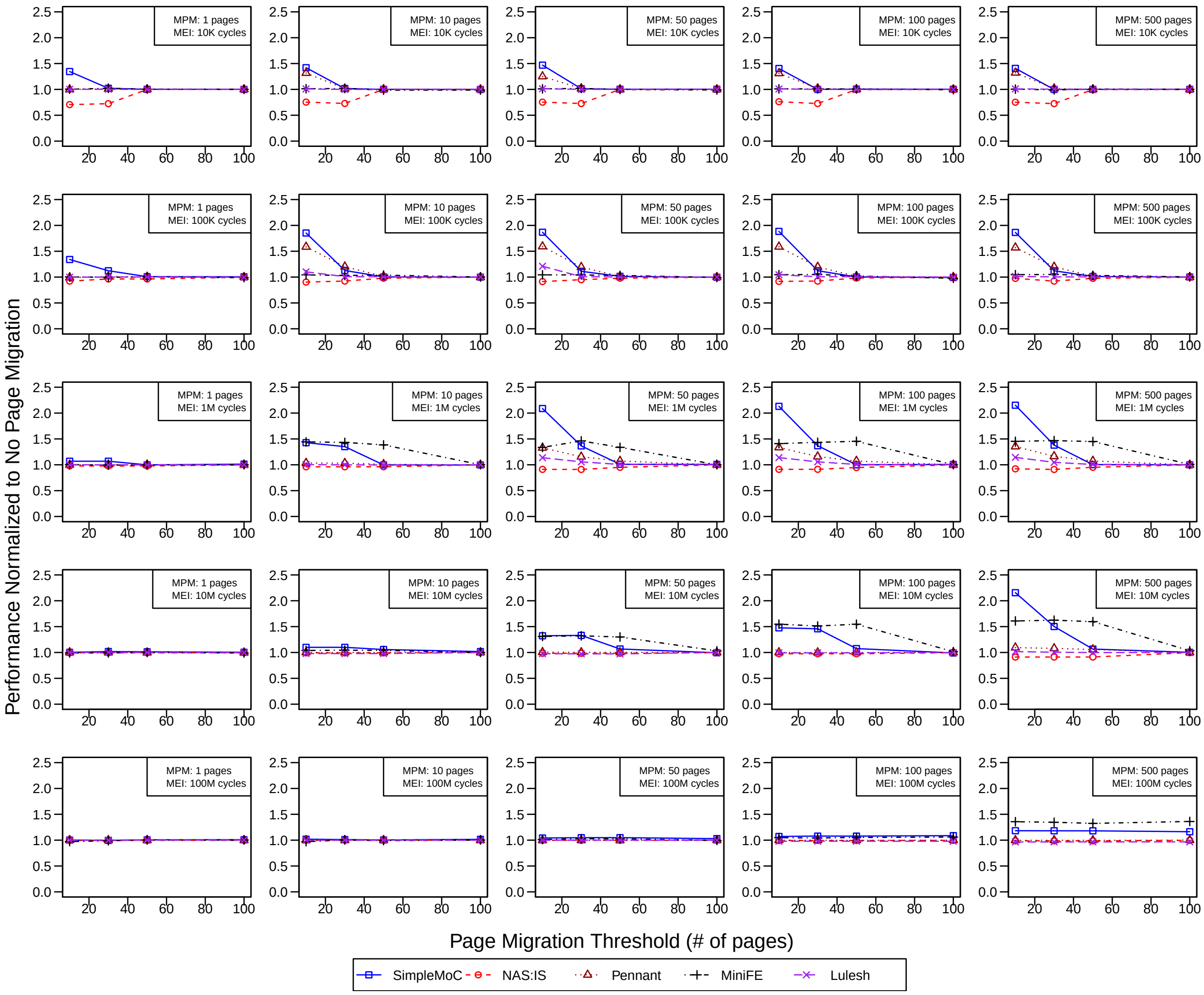

Figure 4.4: Performance improvement (normalized to no page migration) in disaggregated memory system when page migration parameters migration epoch length, number of pages to migrate and page migration threshold are varied. Shootdown latency is maintained at 8us and per page migration delay(cost) is 1us. MPM indicates Maximum pages to Migrate per epoch. MEI indicates Migration Epoch Interval

4.4.1 Effect of Page Migration Parameters

We modelled a combination of a threshold based method and clock based approach to migrate pages from between centralized and private memory. As mentioned in Section 4.3, performance gains of using page migration is dependent on migration frequency (page migration epoch), the number of pages to migrate at a time and page migration threshold to detect a page requiring migration. Hence we vary these parameters and study the effect of them. Figure 4.4 shows normalized performance (IPC) results with respect to disaggregated memory system without page migration. The x-axis of each sub-graph indicates PMT at global memory which decides if the page is a hot page or not. For instance, an x-axis of 50 in each graph indicates that when a shared page is accessed more than 50 times in the current epoch, then that specific page is marked as a page to be migrated to the private memory and might get migrated to the local memory during the migration interval. We varied PMT from 10 to 100 accesses. Graphs in each row show results for a specific migration epoch. For example, the migration interval of graphs in each column of row 1 is 10K cycles. While rows 2, 3, 4 and 5 indicate results for migration epoch of 100K, 1M, 10M, 100M cycles respectively. Each column represents the maximum pages that can be migrated. For instance, results in column 1 are configured to migrate a maximum of 1 page for every migration epoch and columns 2, 3, 4 and 5 migrate a maximum of 10, 50, 100, 500 pages per epoch respectively. The top most frequently accessed pages are chosen if the number of pages to be migrated exceeded maximum pages to be migrated per epoch.

We make the following observations by varying these parameters from Figure 4.4. If PMT is high, pages should be accessed frequently to count them as hot pages. On the other side if PMT is low, most of the pages would be counted as hot pages and there would be a pool of hot pages to choose from for migrating to local memory. Hence PMT decides the hotness of the pages. Therefore we divide our explanation into two parts: (1) High PMT (Pages are counted as hot pages if the frequency of page accesses is very high), and, (2) Low PMT (Pages are counted as hot pages if they are accessed less number of times).

High PMT: If PMT is high, a page should be accessed frequently to be counted as a hot page. In our results we observed that when the epoch size is small, 10K and 100K cycles (row 1 and 2), all the applications that we simulated, do not improve the performance when the page migration threshold is set to 50 accesses or beyond. This is because when the epoch length is small, the number of global memory pages getting accessed more than 50 times in that epoch is very narrow and hence none of the pages would reach the required PMT, resulting in no page migration. Hence when the epoch size is small, 50 accesses is defined as high PMT irrespective of the number of maximum pages to migrate. When the epoch size is larger – 1M and 10M cycles – MiniFE benefits from page migration even if the PMT is beyond 50 accesses (1.46x, 1.33x, 1.45x and 1.44x for 10, 50, 100 and 500 maximum pages to migrate respectively with a migration epoch of 1M cycles). And the performance is normalized to 1 when PMT is set at 100 accesses. Therefore for MiniFE, a high PMT is defined as 50 accesses if epoch size is less and 100 accesses if epoch size is more.

Low PMT: If PMT is low, most of the pages, if accessed regularly, are eligible to be counted as hot pages, hence many hot pages would be moved to local memory and improve the performance. It can be seen that apart from when only 1 page is allowed to migrate per epoch and when epoch interval is too high (100M cycles), we can observe performance improvement. If only one page is allowed to migrate per epoch, the cost of page migration, explained in 4.4.2, outweighs the benefits. Also if page migration epoch interval is too high, page migration frequency is very low which leads to no page migration. Hence there is no improvement in performance (row 5). There is improvement if the number of pages to migrate is 500, since even though the frequency is less, there are a higher number of pages able to be migrated. Although migrating 500 pages at a time is not practical, or at least is likely to present a significant migration cost, we show these results to see the variation in performance improvement. When the migration epoch size is less, 10K and 100K cycles, the benefits of page migration is nullified by page migration cost (row 1). Hence there is no improvement in performance for MiniFE. But for SimpleMoC and Pennant applications, the improvement is around 1.4x, 1.3x for 10, 50, 100 and 500 pages to migrate at maximum per epoch and is around 1.85x and 1.58x more for 10, 50, 100 and 500 pages to migrate at maximum per epoch with PMT of 10 and migration epoch of 10K and 100K cycles. For applications like NAS-IS the impact of page migration cost severely affects the performance. Performance degrades by 30% and 28% when PMT is at 10 and 30 accesses. As the epoch size increases from 10K to 100M the effect of page migration costs decreases as the frequency at which page migrations are performed is less. Hence the performance of the NAS-IS application is normalized to 1. As the epoch size is more 1M and 10M cycles the performance gain for SimpleMoC and Pennant applications diminishes. SimpleMoC achieves peak performance improvement of 2.08x, when migration epoch is 1M cycles and a maximum of 50 pages to migrate with a PMT of 10. MiniFE also benefits from page migration if the migration epoch is maintained at 1M and 10M cycles (1.4x more with PMT of 10, 30 and 50). The peak performance improvement can be noted when epoch interval is 10M cycles and a maximum of 500 pages to migrate with a PMT of either 10, 30 or 50 accesses (1.6x) but migrating 500 pages at once is not advisable. Due to space constraints, moving forward, we will only show the best possible cases for all the applications- PMT of 10 (low), 50 pages to migrate and the migration epoch interval of 10K, 100K and 1M cycles.

4.4.2 Page Migrations Costs

From the above observations it should be understood that intensive page migration leads to severe page migration costs. Page migration costs can be classified into three categories: (a) *Invalidating TLB units - TLB shootdown latency:* TLB shootdown latency can be reduced by using schemes like self-invalidating TLB's [44]. (b) *Swapping page content delay:* While the pages are undergoing swapping, computing units might operate on yet to be swapped page. Since a batch of pages are

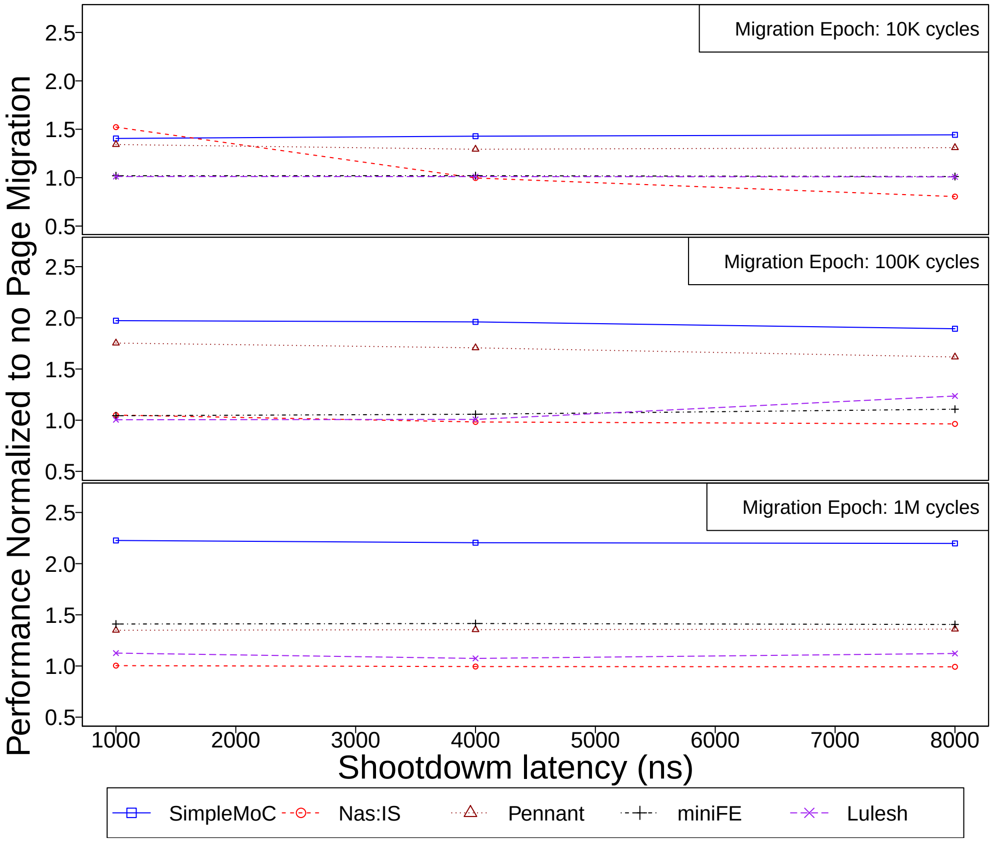

Figure 4.5: Performance improvement (normalized to no page migration) in disaggregated memory system with best possible cases. Migration epoch interval is 10K, 100K and 1M cycles with PMT of 10 accesses, 8us TLB shootdown latency, a maximum of 50 pages to migrate per epoch and per page migration delay(cost) of 1us per page.

undergoing swapping computing units which operate on these pages should halt. We call this page swapping delay. Usually pages are swapped with the help of DMA engines. To account for page swapping operation, we add an additional 1us latency in our experiments. Accordingly, page swapping cost increases by 1us for every page that gets migrated from central memory to local memory. If there are 50 pages to migrate then page swapping cost is 50us. This can be reduced by intuitively recording pending pages that will undergo swapping and performing page swapping atomically. That is, MMU units of each node decides if the address translation should proceed or not by checking if the page associated with the address is marked as a pending pages to swap. If the page is not marked, MMU would proceed with the address translation. If the page is marked and if the status of the page indicates that the page is undergoing swapping, the MMU waits till the swapping is done. If the page is not undergoing swapping but in the pending pages to be swapped list, MMU would proceed with the addresses translation without waiting. With this approach and with the help of atomic page swapping, page swapping delay can be completely nullified. (c) *Cost to find pages to migrate:* This is minimal since most of the work has been done while accounting for the page accesses in the background.

Leveraging on reducing page migration cost schemes, we intuitively evaluated page migration in disaggregated memory systems by varying TLB shootdown latency. Figure 4.5 shows the results for the best case configuration according to Figure 4.4, column with maximum number of page migrations at a time is 50 (column 3) and a PMT of 10 accesses, with varying TLB shootdown latency (8us to 1us) as shown on the x-axis. We optimistically choose a batch of 50 page to migrate at a time and nullified page swapping cost, since we believe that during TLB shootdown, DMA engine would have swapped 50 pages between global and local memory. With low TLB shootdown cost (1us), applications like NAS:IS whose performance was degrading with TLB shootdown cost of 8us, 0.8x, could improve its performance by 1.52x (with migration epoch of 10K). As the epoch size increases, the improvement due to page migration is reduced. 1.05x and 1.0x for NAS:IS application when TLB shootdwon delay is 1us with 100K and 1M migration epoch intervals. For Lulesh application the improvement is 1.75x, 1.7x and 1.61x for 1us, 4us and 8us TLB shootdown delay respectively. For other applications the improvement is marginal.

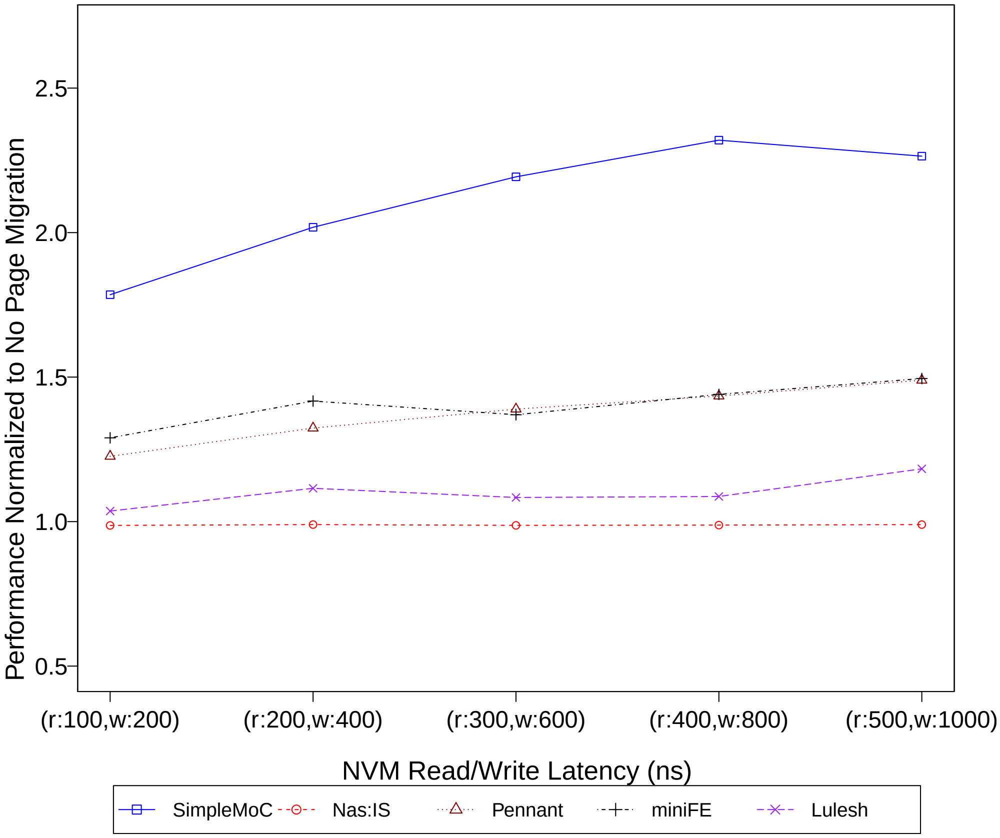

Figure 4.6: Performance improvement in disaggregated memory system with respect to conventional memory system with different NVM read/write latency. PMT is 10 accesses with a maximum of 50 pages to migrate, TLB shootdown latency of 8us and per page migration delay(cost) is 1us per page. Page migration is performed for every 1M cycles.

4.4.3 Sensitivity to NVM's Read/Write Latency

Centralized memory in disaggregated memory systems must meet several requirements. One of them is the ability to allocate memory to all nodes accessing it. NVM memory technology is a perfect candidate to fullfil such requirement as the density of NVM is higher than DRAM. On the other hand, NVMs are notorious for the high write latency compared to DRAM. And although that NVM's read latency is much better than its write latency counterpart, it is still slower than the read latency of a typical DDR generation. We believe that read/write latency are crucial in the migration of pages to/from centralized memory. To this end, we studied page migration benefits by varying NVM read/write latency. We categorize NVM into 5 categories - very fast (read and write latency is 100 and 200ns), fast (read and write latency is 200 and 400ns), moderate (read and write latency is 300 and 600ns), slow (read and write latency is 400 and 800ns) and very slow (read and write latency is 500 and 1000ns).

We can intuitively expect that if the global memory is "fast", then baseline scheme would perform just well without any page migration. On the other hand, if the global memory is "slow", we expect page migration to optimize performance significantly versus the baseline of no page migration, as we reduce the number of "slow" memory accesses. To showcase this, and since the improvement due to page migration is evident for all applications when pages are migrated every 1M cycles, we use this as our migration epoch.

In Figure 4.6, 'r' indicates read latency and 'w' indicates write latency. For example (r:100,w:200) indicates NVM read latency of 100ns and write latency of 200ns. As the type of the NVM varies from very fast to very slow, the benefits of page migration is more clear. For instance performance gains due to page migration for SimpleMoC with very fast global memory is 1.78x and with very slow global memory the performance gain is 2.3x. For MiniFE and Pennant the improvement is around 1.25x to 1.48x when the global memory is varied from very fast to very slow. The improvement due to page migration for Lulesh application with very fast NVM as global memory is hardly 3%, however, when the global memory is very slow the performance gain reaches up to 18%

4.5 Conclusion

To improve the performance of disaggregated memory systems, we have proposed a novel memory management scheme. As disaggregated memory systems support both local and shared memory, we identify hot pages in global memory and provide migration capabilities to local memory using a combination of threshold based and clock based policies. We provide insights into the impact of migrating pages at regular intervals, showing that the benefit of page migration is dependent on factors like page migration epoch, the maximum of number of pages to migrate at each epoch, whether a page migration threshold can be used to differentiate hot and cold pages as well as the costs associated with TLB shootdowns and page swapping delays. We evaluated 5 HPC-relevant applications and benchmarks to study the effect of these factors on page migration. We showed that NVM is a feasible memory type to construct disaggregated memory model, and hence we studied the effect of NVM read/write latency on page migration in disaggregated memory systems. We show the best case improvement of up to 2.3x when page migration is applied on a disaggregated memory system with slow NVM as main memory.

CHAPTER 5: ARCHITECTURE-AWARE VIRTUAL MEMORY SUPPORT FOR FABRIC-ATTACHED MEMORY SYSTEMS

Since memory-centric architectures leverage FAMs as physically shared memory pools, multiple compute nodes, potentially running applications from different users, can access pages in the same FAM memory modules. This access model is different from conventional HPC architectures where each compute node has its own memory modules and applications' memory accesses are limited to its own nodes, unless explicitly requested from other nodes through software interface. Therefore, a new question arises: who is responsible for access control of FAMs? In this chapter, we discuss the existing memory management approaches for FAM systems and explain the advantages and disadvantages of each scheme. Further, we propose a secure and efficient memory management scheme for FAM systems.

5.1 FAM Management Approaches

Based on the previous work, we identified two ways to manage pages in shared FAM pools - exposing FAM and indirect access to FAM.

5.1.1 Exposed FAM (E-FAM)

The first approach is to expose each node to the real physical addresses (FAM addresses) and modify the OS kernel running on each node to communicate with the external memory broker to allocate FAM pages, Figure 5.1(a). OSes need to be patched to communicate with the global memory manager node (e.g., through MPI interface) to coordinate memory management with other nodes [47, 48]. Additionally, in this approach, without strict access control mechanisms, malicious

OSes, applications, and PEs (e.g., accelerators, SoCs, etc.) can potentially compromise the entire system by accessing the data of other users in the shared FAMs. Note that in this system architecture, there could be compute nodes containing PEs from different vendors. Even if not malicious, these PEs could contain bugs in their internal virtual memory implementation, which can compromise the whole system. However, E-FAM approach requires only single-stage address translation compared to two-stage address tranation for indirect access to FAM approach. This leads to lesser memory accesses for address translations, section 5.2.

5.1.2 Indirect Access to FAM (I-FAM)

Externally vetting accesses to shared FAM modules, at the system-level, and not rely solely on internal access control within PEs is the second approach. In this approach FAM pages are transparently allocated to the nodes on-demand, i.e., each compute node has the illusion that it has a contiguous large physical space [47]. Such an approach is similar in spirit to how hypervisors give virtual machines (VMs) the illusion that each VM has a contiguous guest physical memory, which eventually gets translated into the real system physical address through the hypervisor. Adding a translation layer at the system-level allows running unmodified OSes on nodes and enforcing access control, but incurs significant performance overheads due to two-stage address translations, section 5.2. Lim et al. [47] explored a two-stage address translation system for disaggregated memories. We call such a scheme *Indirect FAM*, since FAM is accessed indirectly. In I-FAM, a simple system translation unit (STU), similar in spirit to *Gen-Z memory management unit (ZMMU)* [88], can be implemented in a router connected directly to the node, in the memory blade or as an independent network element, as shown in Figure 5.1(b). STU is responsible for caching system-level translations, i.e., node address to FAM address, and access permissions. Moreover, STU is capable of sending address translation service requests (similar to PCIe's [89]) or request physical pages from the system-level memory broker (in case of unmapped addresses). Note that STU is

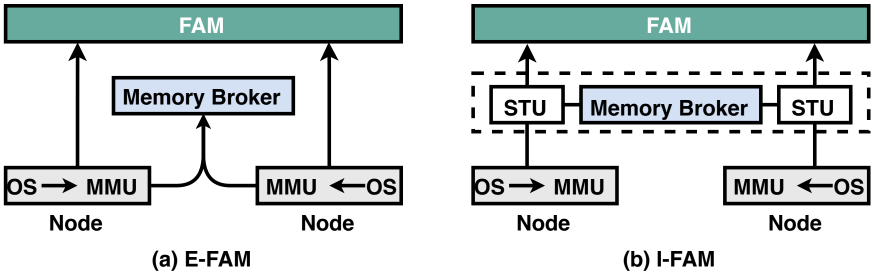

Figure 5.1: Two ways of managing memory in FAM systems.

similar in spirit to the ZMMU [88].

5.2 Hierarchical Page Tables

Hierarchical (multi-tier) page tables are commonly used for address translations in modern servers due to their performance, dynamic growth, and scalability. In such settings, a virtual address is provided as an input to the translation process, then the offsets (derived from virtual address) are used to index each level to obtain the address of the next level. Finally, the last level, typically called PTE level, has the actual translation entry, i.e., the corresponding physical address and the access permissions. However, to reduce translation overheads, hardware-support for memory management, typically implemented as the Memory Management Unit (MMU) and maintained by the OS, is provided. The MMU is responsible for caching the translations, i.e., PTEs, in TLBs. Moreover, MMU can also cache the contents of different levels of the page table in what is called page table walking (PTW) caches [49]. Finally, MMU is responsible for walking the page table in case of TLB miss to complete the translation process.

As shown in Figure 5.2(a), for every TLB miss, the page table is walked. In x86-64 systems, a 4-level page is typically used, and the levels are called PGD, PUD, PMD and, PTE, respectively. Therefore, each memory access needs additional four memory requests to walk the page table. The

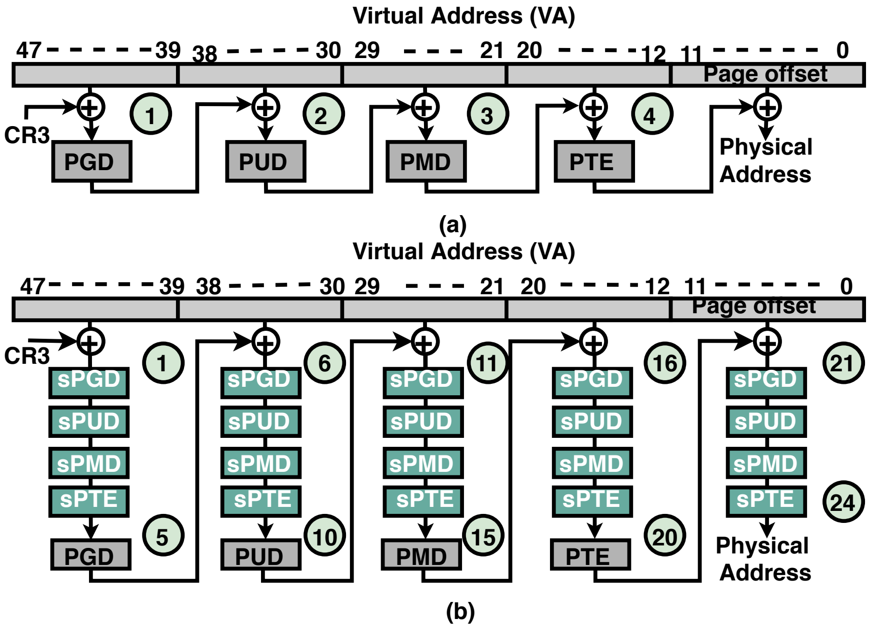

Figure 5.2: Page table walking in (a) x86 and (b) virtualized systems.

root of the page table of the process currently executing on the core is loaded in Control Register 3 (CR3) in the core. In each virtual address, the bits beyond the page offset (typically 12 bits) are divided into multiple sections (9 bits each) where each section is used to index a specific level in the page table. Finally, the last-level page (PTE) has the actual page mapping. Hence, for every TLB miss, the entire page table is walked which will incur an additional four memory accesses.

In virtualized systems, hypervisors like Xen[90] are responsible for maintaining multiple guest systems and managing their memory. For such systems, nested paging is one approach in which two page tables are maintained: one to convert virtual address to guest address and the other is a nested page table to convert guest address to system physical address. Each level of the guest page table has to walk the nested page table which requires four more memory accesses per guest page table level, Figure 5.2(b). Hence, 24 memory accesses are required to fetch the translation. This leads to huge overhead and hence, Bhargava et al. [49] proposed PTW caches, nested TLBs, and nested PTW caches to reduce the number of memory accesses to translate an address. PTW cache unit caches the intermediate level address translations and helps in reducing the average number of PTW steps.

5.2.1 Exposed FAM vs Indirect Access to FAM

While E-FAM provides translation overheads as low as native systems, it requires modifying OS [48] and enormously enlarges the attack surface; any malicious node/OS can map its address space into any location in global memory, hence leaks data from other nodes. Although, I-FAM provides security without any modifications to the OSes, it is not performance friendly. Figure 5.3 shows slowdown in I-FAM compared to insecure E-FAM wherein no indirection is needed. We observe a performance degradation of 20.6x for *sssp* benchmark (details about the methodology and the benchmarks are discussed in Section 5.4). The slowdown is attributed to the increased address translation requests observed at FAM due to indirection at the system-level, Figure 5.4. For instance, the percentage of address translation requests for *canl* benchmark is 44.36% in E-FAM, however, this increases to 84.13% in I-FAM. Also, we note that benchmarks that are not sensitive to address translations become highly sensitive to address translations in I-FAM. Address translation requests increase from 1.81% to 53.69% for *cactus* benchmark. Clearly, I-FAM brings in significant performance overheads.

5.2.2 Threat Model

In our threat model, we assume that compute nodes themselves can have bugs that can be exploited by malicious applications or OSes. Such threats are common and evidenced by the recent vulnerabilities in Intel's processors (e.g., Meltdown [91] and Spectre[92]). Our threat model assumes that a malicious application or OS runs on a specific node that tries to illegitimately access memory

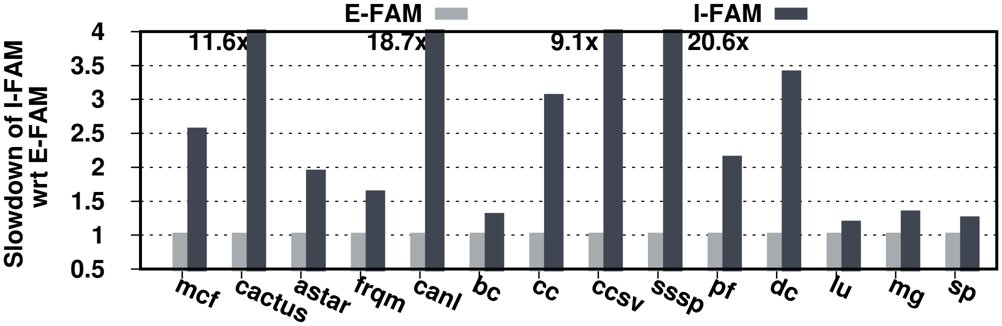

Figure 5.3: Normalized performance with respect to E-FAM.

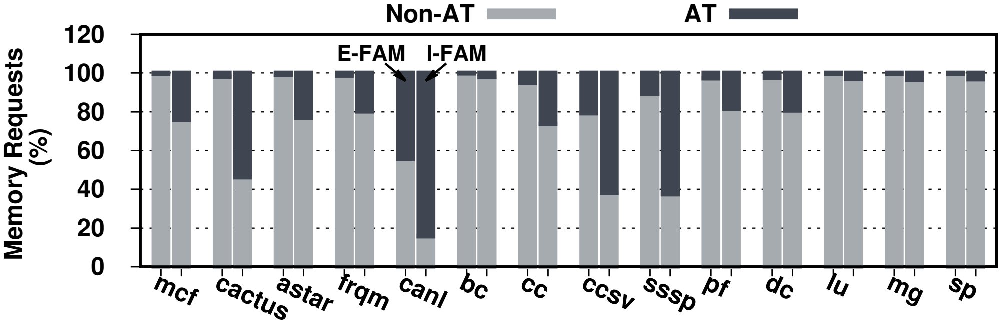

Figure 5.4: Breakdown of percentage of address translation (AT) and non address translation (Non-AT) requests observed at FAM in I-FAM and E-FAM systems.

pages of other nodes and users in the shared FAMs. Note that we assume a compute node, at any point in time, is owned by a single user (i.e., a user allocated the node to run an application). By exploiting a bug in virtual memory implementation within a compute node or a vulnerability in OS, the attackers can directly map its own virtual space to any physical page in the shared FAM and hence, be able to access it freely. Therefore, to minimize the attack surface, an additional level of access control needs to vet accesses that come from compute nodes to ensure that they are for pages belonging to the node. Thus, any pages in FAM that are considered exclusive to a node, must be protected from any access by other nodes as long as such pages are allocated. Attacks such as timing side-channel, covert-channel and physical attacks are beyond the scope of this paper and can be addressed with many available solutions based on the system nature. Similar to most HPC systems, we disallow co-locating resource allocations on the same node, which minimizes the risk of information leakage (within a node). In summary, we mainly focus on enforcing access control on shared FAMs, to limit the impact of vulnerabilities within compute nodes on other compute nodes' data. Such protection is analogous to security guarantees provided by virtual memory for applications running on a native system, but at the node level. Our threat model trusts the memory and fabric, i.e., memory provides nodes with the requested data and does not try to give them data from other locations. Similarly, the fabric will not change the address in a request after it has been vetted by access control.

5.3 Decoupled Access Control and Address Translation (DeACT) Scheme

To minimize the performance overheads of transparent access control and management support for shared FAM pools, we propose *decoupled access control and address translation*. DeACT leverages the architecture layout of memory-centric architectures and the ability to decouple access control from translation. Specifically, DeACT allows *unverified* caching of translations in the small local memories within compute nodes but enforces access control at the system level. By decoupling access control and translation and leveraging part of the local memories in compute nodes as unverified caches, DeACT exploits the high spatial locality of access control metadata (ACM) for each node. By doing this, DeACT brings in significant performance improvements and reduces the number of translation requests significantly, while strictly enforcing access control. The goal of DeACT is to design a FAM architecture with better performance, without sacrificing security and with minimum or no changes to the OS. A comparison of baseline FAM management approaches and our proposed DeACT FAM approach is shown in Table 5.1.

Table 5.1: FAM Architectures Comparison.

Architecture	**Performance**	**Avoid OS Changes**	**Security**
E-FAM	✓	✗	✗
I-FAM	✗	✓	✓
DeACT	✓	✓	✓

5.3.1 Overview

When designing support for virtual memory, we aim at abstracting away the details of the global memory from nodes' OSes, however, while enforcing isolation and minimizing translation overheads. To do so, we adopt a two-layer approach where each node's OS manages an imaginary flat node physical memory. The node physical memory range can be thought of as a range of two different NUMA zones, one zone (low addresses) corresponds to the local DRAM and the other zone (high addresses) corresponds to the FAM. With such a design, each node's OS manages its memory allocations oblivious to the actual status of FAM. While such an approach abstracts away the complexity of managing a shared resource (memory), it adds significant performance overheads due to two levels of indirection. Therefore, we need novel mechanisms to improve the performance of such a design without compromising security.

One major observation we make is that access control can be decoupled from the translation process. In particular, the translation from node address to FAM address can be sped up significantly by caching the translations at node-level memory. Later, if the translation of a specific node address exists locally in the node, the global memory request is forwarded to the FAM, with the obtained/cached FAM addresses. In other words, the node can provide the final FAM addresses it needs to access. As the reader can expect, the access control is offloaded to the off-the node components, e.g., STU units. Since the access permissions need to be checked for the specific

FAM address provided by the node, we dedicate specific parts in the FAM to store ACM of FAM. Such parts are known for STUs and the addresses of the ACM of any FAM page can be derived merely from the FAM address. For instance, assume we want to keep a 16-bit ACM for each 4KB page and assume the metadata starts at address $MTAdd$ in FAM. To read ACM of FAM address X, we read the 64-byte block at address $MTAdd + \frac{X}{4096 \times 32}$ (division by 32 is needed to get the base address of 64-byte block for 16-bit ACM). For simplicity, the metadata of each 4KB page is the node ID of the node that owns that page and read/write permissions. Read and write fields consume two bits and the rest of the bits (14) are allocated for the node ID. Nodes can choose to execute pages read from the FAM. However, while executing instructions, the access permissions of the executable pages are verified by TLBs, within the node. FAM pages could also get shared between the nodes. Thus, we use all the node ID bits of the page metadata set to 1 to indicate a shared page. Hence, we can have up to 16383 nodes supported in the system.

Since pages can be shared by a subset of nodes, just indicating a page is shared is insufficient. Therefore, we use a bitmap-like scheme to indicate which nodes are allowed to access a specific page. However, since having a bitmap for each 4KB can introduce significant overheads, we limit shared pages to 1GB physical pages. For each 1GB physical page in global memory, we have a corresponding 16K bits bitmap (2KB) in the metadata region. Since such overhead is negligible (less than 0.0001%), and to enable easier indexing of metadata, we dedicate a bitmap for each 1GB physical region regardless of being used as a shared page or not. Therefore, when ACM is accessed, if the node ID bits of the metadata indicates a shared page, we immediately fetch the corresponding parts of the bitmap to check if the node has access permissions. In contrast, if the node ID bits do not indicate a shared page, we simply compare its value (owner node ID) with the ID of the requesting node, to verify the legitimacy of the access. Note that when a shared page is allocated (or becomes shared), all of its node ID bits in the metadata fields correspond to its 4KB chunks (sub-pages) are set to shared, i.e., `0xfffd`. When the page is shared, the last two bits of the

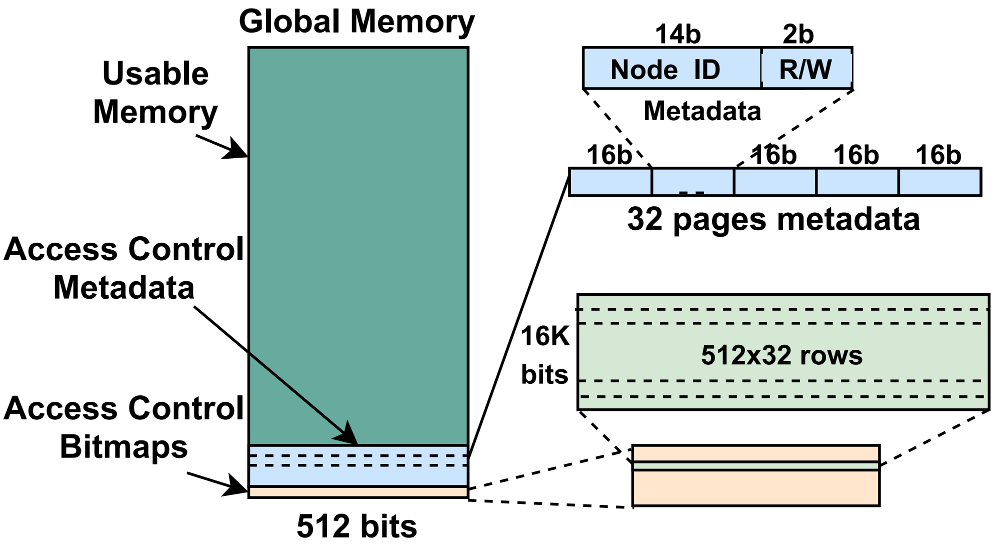

Figure 5.5: Page access control metadata and bitmaps in FAM.

metadata field indicate read and write permissions assigned to the node. This enables enforcing mixed access permissions for nodes sharing a page. For instance, a specific subset of nodes is allowed to read and write to the shared page and the rest of them can only read the shared page.

One obvious optimization to reduce the overheads of obtaining such ACM is to cache them. However, since such metadata must be enforced by FAM managers, not the node, such metadata should be only cached outside the nodes and inaccessible by the nodes or their own OSes. Therefore, we opt for caching such metadata in STUs. Such STUs can potentially have a small lookup table, similar to TLBs. As mentioned earlier, such STUs can be added to the global memory blade or simply at the first router/switch that connects a node to the system fabric. It is also important to note that such metadata has very high spatial locality, a single 64B block covers 32 4KB pages, i.e., 128KB region for a 16-bit ACM. Therefore, even a very small TLB-like cache can save a significant number of reads to access control metadata. While beyond the scope of this paper, in encrypted memories, if each node uses a unique memory encryption key, we could allow read requests without checking access control; writes can tamper with data but reads are useless if the node has a different unique key, and thus, no need for enforcing access control for reads.

As we now understand how our decoupled access control works, we will discuss how we accelerate the translation process. To speed up the translation process, we (a) propose node-level unverified caching of system-level translations: We notice that a very small portion of local memory can be used to cache system-level translations, which will be later sent for verification at system-level, (b) efficiently cache ACM in STU.

5.3.2 System Overview

Figure 5.6 shows a schematic overview of our proposed design. Decoupling the metadata from page mapping enables the system-level translations to be cached in the local memory. Hence, we maintain a FAM translation cache in the DRAM. We add a *FAM translator* ① in the memory controller to map node addresses to FAM addresses by accessing the FAM translation cache ②. Although node addresses are mapped to FAM addresses by the FAM translator unit it is still a partial translation since accesses have to be verified. To complete the mapping, the FAM accesses are verified by the STU ③. Hence, unlike I-FAM, DeACT requires two steps to translate a node address and verify the access.

For a translation miss, the FAM page table has to be walked. In our design we let the walking to be done by the STU since we observe that the overhead of including FAM PTW inside the node is costlier than the benefits. Firstly, due to security reasons, we aim at a clean separation of address translations and ACM, if the fabric translations are cached inside the node. Hence, if the intermediate translations are also cached within the node, the ACM for intermediate page tables should also be decoupled. However, the two-step process required to complete the mapping, delays address translations significantly, considering four memory accesses during PTW. That is, at every

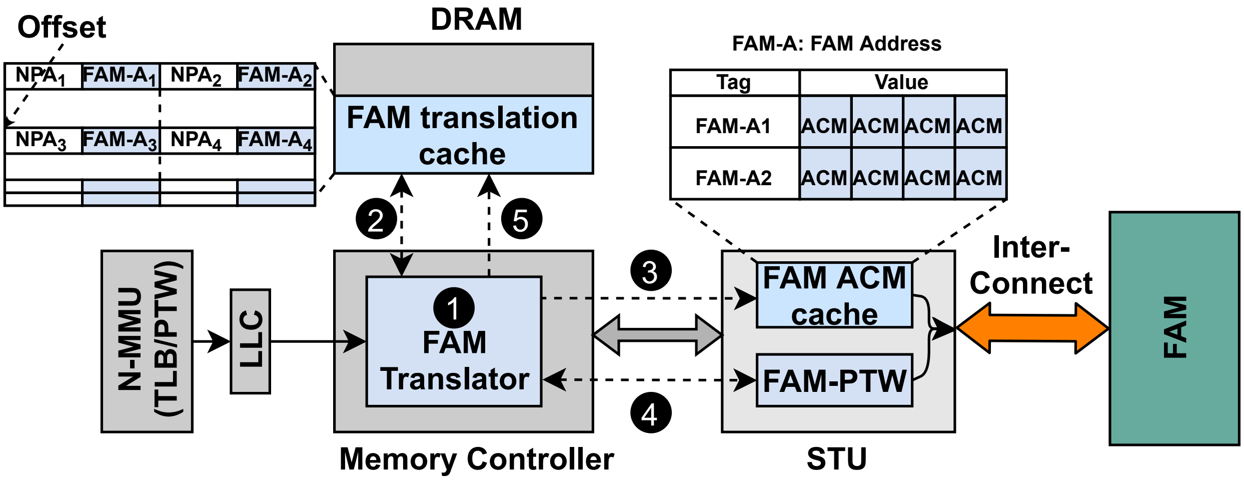

Figure 5.6: DeACT FAM schematic.

intermediate level, the ACM should also be fetched from the memory incurring additional memory accesses. Secondly, since FAM translation cache size in local memory is significantly higher than the STU cache size, we observe a hit rate of more than 90% in FAM translation cache in the local memory. Hence, walking the FAM page table within the node would unnecessarily increase the complexity without many benefits. Also, it would increase the complexity of the memory controller. Thus, we apply DeACT only to the last level of the page table (PTE). Therefore, during a FAM translation miss, the FAM translator forwards the request to the STU, which walks the FAM page table and fetches the entry on behalf of the FAM translator ④. After receiving the missed translation, the FAM translator maps the pending requests and then updates the FAM translation cache in the local memory⑤.

5.3.3 FAM Translator

The idea of the FAM translator in the local memory controller is to translate node addresses to FAM addresses without verifying memory accesses. Functionalities of FAM translator are: (a) fetching the translation from the FAM translation cache (b) matching the tag (c) handling translation hits and misses (d) handling off-the node responses and (e) updating FAM translation cache in the local memory.

Accessing DRAM for Translation: To fetch the translation from the local memory, ⓐ in Figure 5.7, the FAM translator calculates the local memory address by adding starting address of FAM translation cache to the offset, Figure 5.6. Offset is dependent on the type of the FAM translation cache in the local memory. For simplicity, we use a four-way associative cache. This is because memory access granularity is 64-bytes and each mapping entry requires 104 bits; 52 bits for tag (node page address) and 52 bits for value (FAM page address), for a page size of 4KB. Single memory access fetches four entries. Thus, offset is obtained by performing a modulus operation on node page number with the number of FAM translation cache sets.

Tag Matching: After fetching the translations from the local memory, FAM translator matches address tags using comparators, ⓑ in Figure 5.7. We add four comparators and a multiplexer to perform tag matching concurrently. If none of the tags match, the output of the multiplexer is set to 0. This takes just one cycle to match the tag but the number of comparators required is 4x more compared to using just one comparator when four tags are matched serially in four cycles. However, these additional comparators add up minutely to the overall hardware cost and area.

Handling Translation Hits: When any of the tags, fetched from the FAM translation cache, match with the required node page address the multiplexer outputs the respective FAM address. FAM translator replaces the node address with the FAM address and forwards the request. However,

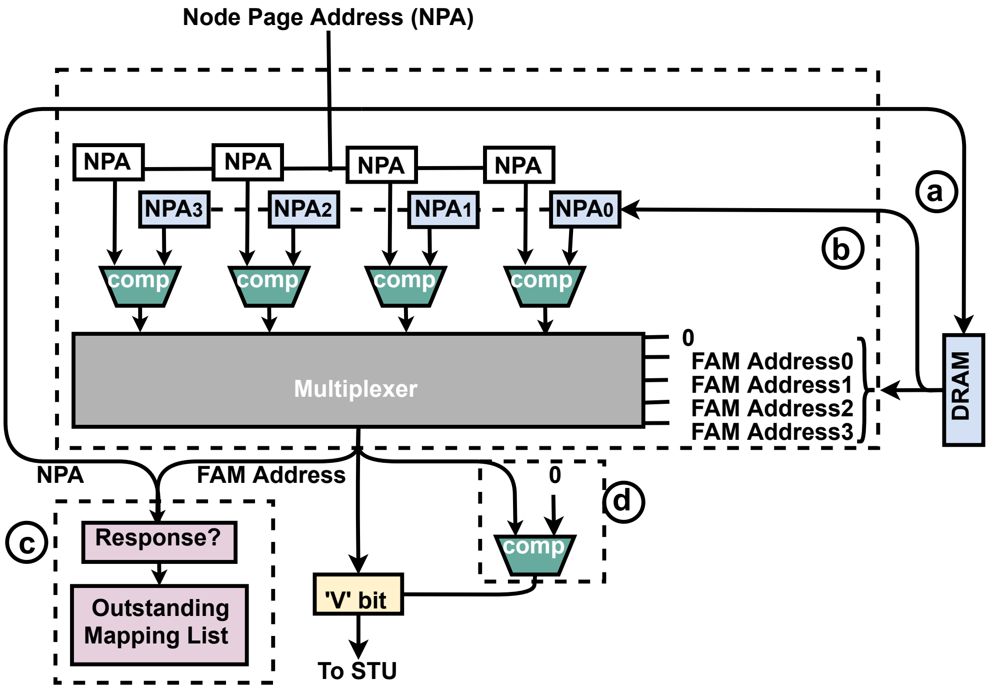

Figure 5.7: FAM translator.

before forwarding the request to STU, the FAM translator identifies if the request is expecting any response back from the FAM. If so the FAM address to node address mapping is stored in the *outstanding mapping list*, ⓒ in Figure 5.7. This is because FAM responses contain data tagged with FAM addresses and nodes only deal with node addresses. *Outstanding mapping list* is used to convert the FAM address to node address during FAM response. Since the number of outstanding requests is limited (128 requests in our evaluation setup, Section 5.4) the number of entries in *outstanding mapping list* is also limited. In I-FAM this list is maintained in STU. But since the FAM translations are performed within the node and STU does not understand node addresses in DeACT, this list is maintained within the node.

Handling Translation Misses: A translation is identified as a miss if the output of the multiplexer is zero, ⓓ in Figure 5.7. During a miss, FAM translator forwards the request to the STU to walk the page table. FAM PTW unit of STU retrieves the node address from the missed request and walks the page table. After the page table is walked, the STU translates the node address to the FAM address and then verifies the access to forward the missed request to the FAM. Also, the STU sends the page mapping to the FAM translator for updating the FAM translation cache and to register the mapping in the *outstanding mapping list* if needed.

STU receives two types of requests from a node, mapped and not mapped requests. Mapped requests are those whose node address is translated to FAM address by FAM translator. For such requests, STU verifies FAM access permissions. On the other hand, STU walks the page table for not mapped requests using the node address from the request address field. To make STU distinguish between the two types of requests we add a verification ('V') flag to the request packet. This flag is set by the FAM translator unit if the mapping is successful and is reset for a missed translation. Using the 'V' flag STU either forwards the request to the verification unit or to the PTW unit.

Handling off-the node memory responses: FAM translator segregates responses into two types (a) memory response and (b) mapping response. Memory responses are forwarded to the last level cache by fetching the node address from the *outstanding mapping list*. Mapping responses are received from the STU PTW unit. FAM translator updates the FAM translation cache during mapping response.

Updating FAM Translation Cache: Since the granularity of memory access is 64 bytes each access to FAM translator operates on four FAM mappings. To update the FAM translation cache, the FAM translator has to write to one of the four mappings fetched. Hence, during a translation cache update, the FAM translator reads 64 bytes from the local memory, updates one of the entries, and writes back 64 bytes. For simplicity, we randomly selected one of the four entries to replace. It is possible to implement different cache replacement policies but such policies require additional

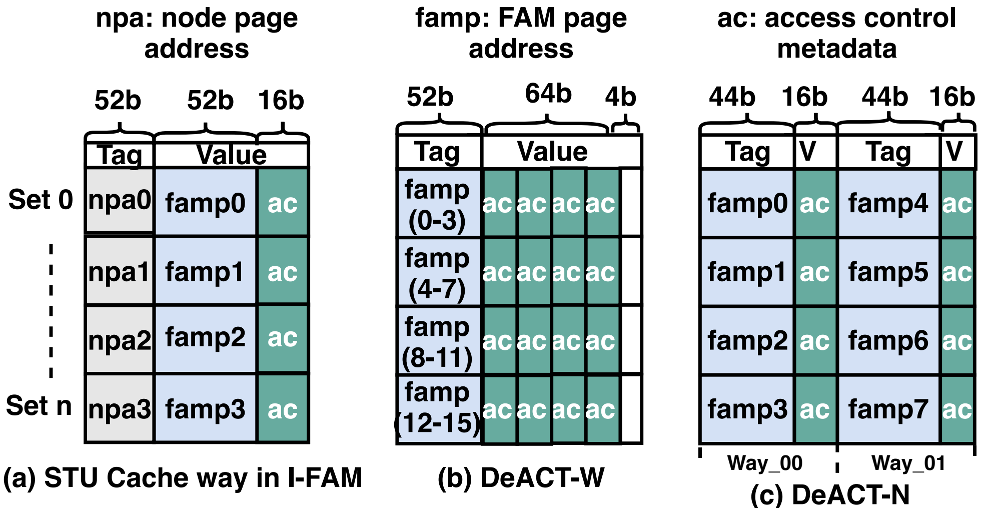

Figure 5.8: ACM organization in STU cache way in (a) I-FAM and (b,c) DeACT.

DRAM space; to store mapping status, and additional writes to the DRAM; to update mapping replacement status for every FAM access.

5.3.4 FAM Access Verification

Memory accesses verification is performed by the STU in DeACT. STU verifies the memory access by a) checking if the page being accessed is assigned to the node, using node ID in the metadata and b) checking the read and write permissions from the metadata. However, the metadata is not provided by the node since we maintain the page metadata in the memory off-the node to provide security, Section 5.3.1. Hence, for STU to verify the FAM access it needs to fetch the page metadata from the memory, Figure 5.5.

Since STU is off-the node, it can be used to cache ACM. STU in I-FAM caches both FAM page mapping and ACM together, Figure 5.8(a) (52 bits for the tag (node page address) and 52 bits for FAM page address and 16 bits for ACM). However, STU in DeACT caches only ACM. Although

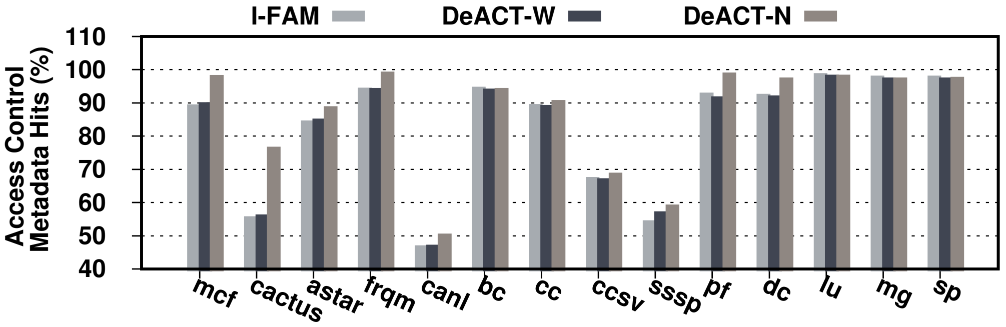

Figure 5.9: Access control metadata hit rate.

DeACT reduces the frequency at which the page table is walked by leveraging local memory to store FAM mappings, it introduces additional memory access for the ACM. Hence, to reduce the number of accesses to FAM for ACM, we explore organizing ACM in the available space, after decoupling the page mapping from the STU cache in DeACT.

Way-level contiguous organization (DeACT-W): This is a simple organization wherein the space available in each cache way, after removing the address mapping, is used to cache ACM of contiguous pages, Figure 5.8(b). Since ACM is 16 bits and the space available is 52 bits (FAM page address), four contiguous pages ACM is stored in one cache way. For instance, ACM for pages from 0 to 3 are stored in one cache way and 4 to 7 are stored in a different cache way. Hence, caching of ACM increases by four times.

Non-contiguous organization (DeACT-N): With DeACT-W we observe ACM hit rate of almost 90% for most of the benchmarks, Figure 5.9. However, ACM hit rate for benchmarks like *canl*, *sssp* and *cactus*, which are sensitive to address translations, is less than 60%. This is because STU in DeACT-W achieves a higher hit rate when spatial locality, while accessing memory, is higher. However, since FAM is shared by multiple nodes, memory allocation is random and hence, has poor spatial locality while accessing memory. Thus, instead of organizing STU ACM cache to cache contiguous pages ACM, we organize STU ACM cache to cache ACM of non-contiguous pages i.e., the free space is used to store tag and ACM pair of another page which is either contiguous or non-contiguous, Figure 5.8(c).

Since each tag and ACM pair needs 68 bits; 52 bits for the tag and 16 bits for the ACM, the available free space, 52 bits, is not sufficient to store an additional pair. Thus, to fit ACM for two different pages within the same way of a set (within the available space) we confine the number of tag bit to 44. With 44 tag bits STU can cover up to 8 tera pages metadata[1] and hence, each node can access 32 petabytes of memory unlike 16384 petabytes with DeACT-W. However, 32 petabytes is also significantly higher for a node. Thus, each cache way is subdivided into two sub-ways (way_00 and way_01 for way 0) and each sub-way has a tag and ACM. This increases the total number of ways for a set and matching the tags of sub-ways is similar to matching the tags of different ways in a cache. Organizing ACM in STU cache in this manner doubles the caching of ACM and unlike DeACT-W, DeACT-N stores non-contiguous pages ACM.

The non-contiguous organization of ACM in the STU cache increases the hit rate from 90% to almost 99% for most of the applications. Also, the hit rate for address translation sensitive benchmark, for instance, *cactus*, increases from less than 55% to almost 76%. The improvement compared to DeACT-W is due to random accesses to FAM (see Section 5.4).

5.4 Methodology and Results

To evaluate our design we used a decoupled memory model implemented in SST [55]. SST is an event-based cycle-level simulator that has been proven to be one of the most reliable simulators for large-scale systems due to the scalability and modular design of its components. SST includes multiple simulation modules for various components. To evaluate FAM architectures a FAM man-

[1] Note that these numbers are based on the tag and data bits assumed in the STU cache of I-FAM as shown in Figure 5.8(a)

Table 5.2: System Configuration

Node	
CPU	4 Out-of-Order cores, 2GHz, 2 issues/cycles, 32 max. outstanding requests
TLB	2 levels, L1 size: 32 entries, L2 size: 256 entries
L1	Private, 64B blocks, 32KB, LRU
L2	Private, 64B blocks, 256KB, LRU
L3	Shared, 64B blocks, 1MB, LRU
Local memory	DRAM, Size: 1GB
STU	
Cache	Size: 1024 entries, associativity: 8
Fabric Network	
Latency	500ns
Fabric Attached Memory (NVM)	
Capacity	16GB
Latnecy	Read 60ns, Write 150ns
Banks	32
Outstanding requests	128

ager (memory broker), Opal [56], was developed in SST. We modified SST memory management unit, Samba [57] and Opal [56] modules to model our design. We modeled an STU component in SST to translate node addresses to fabric addresses and to verify FAM accesses. As our approach focuses on accelerating the address translations, we validate our approach by calculating the performance of the system in terms of instructions per cycle.

Table 5.2 shows system simulation parameters. We simulated 4 cores and each core can serve up to two instructions per cycle with a frequency of 2GHz. Each core is configured to execute a minimum of 100 million instructions of an application execution during its HPC-relevant kernels. L1, L2, and L3, caches are inclusive with sizes 32KB, 256KB, and 1MB respectively. Local memory, is 1GB DRAM[48]. For FAM, as projected by many system vendors (e.g., as in The Machine of HP Labs [13]), we use emerging NVM due to its high-density, ultra-low idle power mixture of storage and memory characteristics[93, 87, 94, 44, 95]. Our assumed FAM is 16GB NVM[2]. We simulated a fabric network to connect to NVM memory with a network latency of 500ns, modeled after recent research and public projections for fabric interconnects [48, 96, 97]. Two levels of TLBs, each of which is simulated with 32 and 256 entries within the node. Since STU is an external hardware per node, we have restrictions over adding additional hardware. Hence, to avoid significant hardware overhead, we implemented STU to cache 1024 page table entries with 128 sets and associativity of 8, similar to Haswell Xeon L2 TLB design[98]. However, we also evaluated DeACT by varying STU cache size. For optimization proposed by Bhargava et al. [49] we used 32 PTW cache entries. The proposed FAM translation cache size in DRAM is 1MB.

Since our focus is on HPC applications we evaluated benchmarks from different benchmark suits, as shown in Table 5.3. Our selection of benchmarks from these benchmark suits are based on (a) the benchmark should have a minimum of 5 misses per kilo instructions (MPKI) (b) should be compatible with the simulation setup (c) the performance degradation with I-FAM should be more than 15% compared to E-FAM since we observe application which does not get impacted much by introducing indirection degrades it performance with DeACT, explained in Section 5.4. Considering these criteria we have evaluated 29 benchmarks and zeroed in on 14 benchmarks that are meeting the requirements. Selected benchmarks with their respective MPKI are shown in Table 5.3. We used short forms to represent applications. The short forms are next to the application names in Table 5.3. Connected components graph analytic benchmark has 2 variants *cc*; which uses Afforest sub-graph sampling algorithm [99], and *ccsv*; which uses Shiloach-Vishkin algorithm [100].

The goal of DeACT is to provide security from other tenants in FAM systems without significantly

[2] In reality local memory is in GBs and global memory is in TBs or PBs. However, given slow simulation speeds, we scale down the memory sizes and among the total application's memory (average of 309MB during the simulation period), 20% is allocated from the local memory and 80% is allocated from the FAM.

Table 5.3: Applications

Benchmark Suite	Application	MPKI
SPEC CPU 2006 [101]	Mcf	73
	Cactus	60
	Astar	9
PARSEC [102, 103]	Freqmine (frqm)	16
	Canneal (canl)	57
Intel GAP [104]	Betweenness Centrality (bc)	113
	Connected Components (cc, ccsv)	56, 130
	Single-Source Shortest Paths (sssp)	144
Mantevo [105]	Path Finder (pf)	41
NAS [65]	DC	49
	LU	111
	MG	99
	SP	141

impacting the performance. Hence, we compare DeACT with two baselines, E-FAM and I-FAM. E-FAM is not secure but has better performance. I-FAM is secure but performs poorly (remember that I-FAM is similar to the optimization proposed by [49]).

5.4.1 FAM Address Translation Hit Rate

Figure 5.10 depicts address translation hit rate while accessing FAM in I-FAM and DeACT. The hit rate corresponds to the number of mapping entries that are cached in I-FAM and DeACT. DeACT has a significantly higher hit rate (more than 90%) because the FAM translation cache in local memory can cache a significantly higher number of mapping entries than limited entries that can be cached in I-FAM using STU cache. For instance, the hit rate for *canl* benchmark is as low as 46.44% in I-FAM. However, with DeACT the hit rate is improved to almost 95.88%. Hence, only 4.12% of the FAM accesses require page table to be walked.

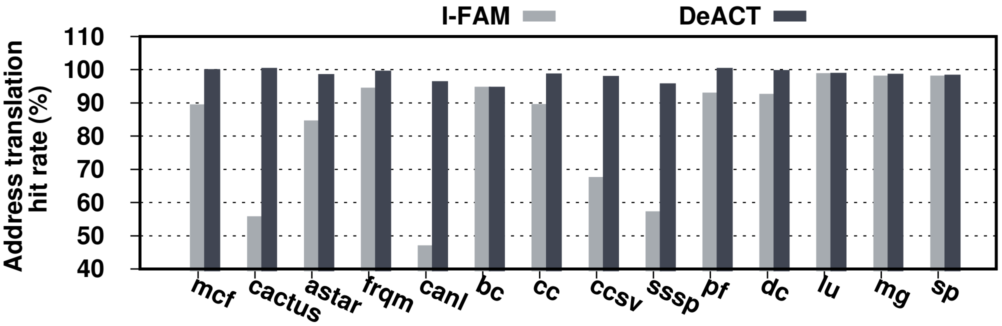

Figure 5.10: Address translation hit rate in I-FAM and DeACT

5.4.2 Address Translation Requests at FAM

Although the frequency at which PTW is reduced with DeACT, it introduces additional memory access for ACM. ACM is cached at STU cache and address translation is cached in the local memory. Hence, in DeACT, address translation and ACM has different hit rates. As shown in Figure 5.9, ACM hit rate in DeACT-W is not improved compared to I-FAM due to poor spatial locality. Hence, the reduced number of address translation requests observed at FAM in DeACT-W, Figure 5.11, is only due to reduced frequency of walking the page table and it also includes additional memory access for ACM. However, ACM hit rate is improved in DeACT-N due to the non-contiguous caching of ACM. For mcf benchmark, address translation requests sent by the node to the FAM are reduced from 23.97% to 11.82% with DeACT-W, and this further reduces to 1.77% with DeACT-N.

5.4.3 Impact of DeACT on Performance

In this section, we show how DeACT performs compared to E-FAM and I-FAM. E-FAM performs better than I-FAM and DeACT, hence, we show our results with respect to E-FAM in Figure 5.12. As mentioned in Section 5.2.1, I-FAM slows down the system performance significantly. Our ex-

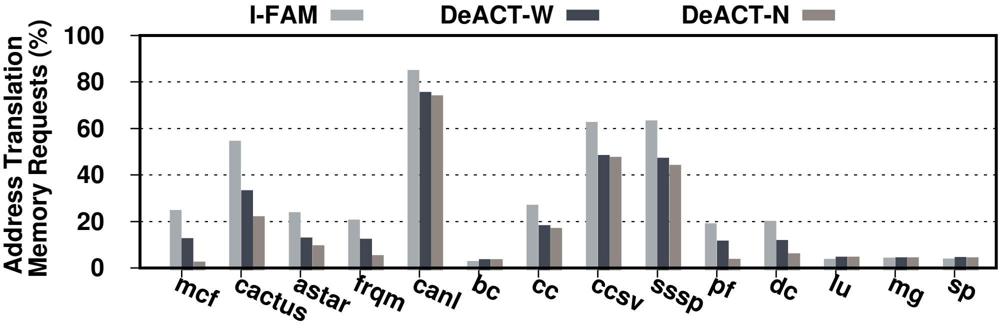

Figure 5.11: Percentage of address translation requests at FAM.

periments demonstrate that DeACT can potentially bridge the gap between E-FAM and I-FAM. For instance, *mcf* slows down by 0.39x in I-FAM compared to E-FAM. DeACT-W performance is 0.7x wrt E-FAM, improving the performance by 1.79x compared to I-FAM. Further with DeACT-N the performance is improved by 2.55x and is just 0.92x times slower than E-FAM. This improvement is attributed to the increased FAM address translation hits, using local DRAM and increased ACM hits in STU, leading to decreased accesses to FAM for page table requests by the node, as shown in Figure 5.11. The inequality between performance improvement and reduction in the observed percentage of address translation requests at FAM is because the local memory is accessed for every FAM access for the translation. In Figure 5.12, DeACT-A indicates the performance improvement achieved by only caching address translations in the local memory and not modifying STU to store ACM of multiple pages per entry as with DeACT-W and DeACT-N. We observe the performance improvement of DeACT-A and DeACT-W is almost the same since FAM pages are allocated randomly. DeACT-W performs better when the spatial locality is higher while accessing memory. However, with random FAM allocation, the benchmarks have poor spatial locality while accessing memory. The additional improvement achieved by DeACT-N compared to DeACT-A is due to caching ACM of two non-contiguous pages within an entry in the STU.

For *canl, ccsv and sssp* benchmarks, we observe a significant percentage of FAM address transla-

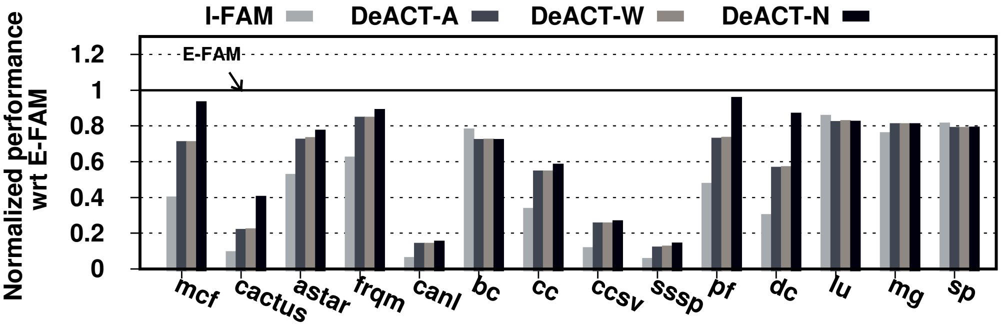

Figure 5.12: Normalized performance with respect to E-FAM.

tion misses in I-FAM, and hence, we observe an increase in the percentage of address translation requests to FAM. The performance for such benchmarks even with DeACT-N is slower compared to E-FAM, 0.14x for *canl*. However, compared to I-FAM, DeACT-N achieves a speedup by 2.7x for these benchmarks.

DeACT-N achieves a maximum performance improvement of 4.6x for *cactus* benchmark. However, DeACT either does not improve or degrades the performance for *bc*, *lu*, *mg* and *sp* benchmarks. Because these benchmarks are very less sensitive to indirection in I-FAM, Figure 5.11, as they have better address translation hit rate, Figure 5.10. However, in DeACT the DRAM has to be accessed for address translations, which is costlier than accessing the STU cache. Also, the benchmarks have to go through two serial steps for address translation and access verification, unlike a single step for the same in I-FAM. Hence, DeACT is better suitable for benchmarks which have a significant impact on performance with I-FAM. In total, we observe an average performance drop of 69.7% with I-FAM and with our proposed mechanism the performance degradation is 35.3% compared to E-FAM. Hence, DeACT improves I-FAM by 80%.

5.4.4 Sensitivity Analysis

The impact of performance in FAM systems is dependent on various factors. In this section, we show how DeACT behaves under various system configurations. The default system parameters are as shown in Table 5.2. Note that for sensitivity results we show the geometric mean of the evaluated SPEC CPU 2006, PARSEC, and GAP benchmarks separately. Also, among NPB benchmarks, we observed *dc* is the only benchmark that has a significant performance impact in I-FAM even under various circumstances. Hence, going forward we show sensitivity results only for *dc* benchmark among NPB benchmarks. Also, since DeACT-N improves the performance more than DeACT-W, we focus on DeACT-N scheme.

5.4.4.1 STU Cache Size and Associativity

One of the main factors which impact the performance of I-FAM is the size of the STU cache. STU is hardware maintained outside the node to enforce system access control and page mapping. The number of entries STU can cache is limited, since we are proposing STU per node and is implemented in the routers connected to the nodes. Adding more entries indicates adding more hardware which increases the hardware budget and complicates routers. In our experiments, STU caches 1024 entries. However, we study DeACT by varying STU cache size from 256 entries to 4096 entries. Figure 5.13 shows performance speedup compared to I-FAM by varying STU cache size. As STU cache size decrease the speedup with DeACT is significantly high, 4.68x with 256 entries for *dc* benchmark. However, as the cache size increase, the performance improvement is confined which is obvious. The speedup reduces from 3.45x to 1.75x when STU cache size is varied from 256 entries to 4096 entries for PARSEC benchmarks. Higher STU cache size has a higher hit rate and hence, less address translation requests to FAM. However, higher cache size leads to more hardware overhead.

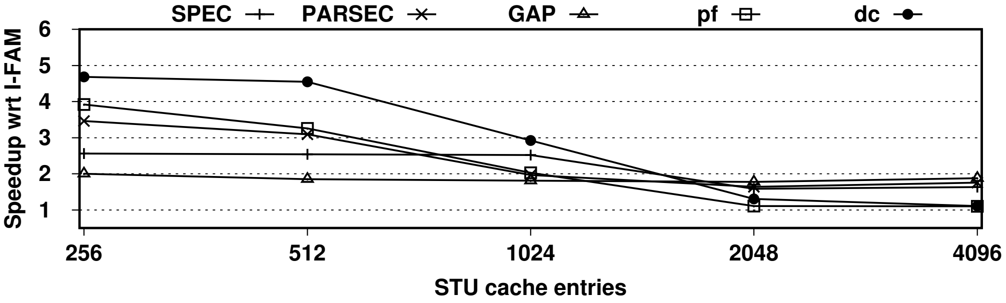

Figure 5.13: Performance improvement wrt STU cache size.

Although we do not show here, we also evaluated DeACT by varying STU cache associativity. We observed that as the associativity increases, the performance improvement with DeACT decreases and gets saturated. When associativity is four the performance improvement is 3.26x for *dc* benchmark and is 2.66x when associativity is 32. The speedup is 2.5x when associativity is greater than 32 for the same benchmark. Similarly for PARSEC benchmarks the speedup is 2.18x, 1.83x, and 1.81x when associativity is 4, 32, and greater than 32.

5.4.4.2 Access Control Metadata Size

ACM size is a key design aspect of DeACT as the number of nodes supported by FAM systems is dependent on metadata size, refer to Section 5.3.1. With 16-bit metadata size FAM systems can host a total of 16383 nodes and with 8-bit metadata 8191 nodes are supported. When ACM is 8 bits STU in DeACT-W can cache metadata of eight consecutive pages, with 16-bit metadata STU can cache metadata of four consecutive pages, and with 32-bit metadata STU can cache metadata of two consecutive pages, increasing the amount of metadata cached by 8x, 4x and 2x respectively compared to I-FAM. However, we observe that the performance improvement is almost the same for these three scenarios, Figure 5.14. This is because, as asserted, although caching of ACM increases in DeACT-W, it caches only ACM of contiguous pages and since the allocation of FAM is random excess caching of ACM is not leveraged.

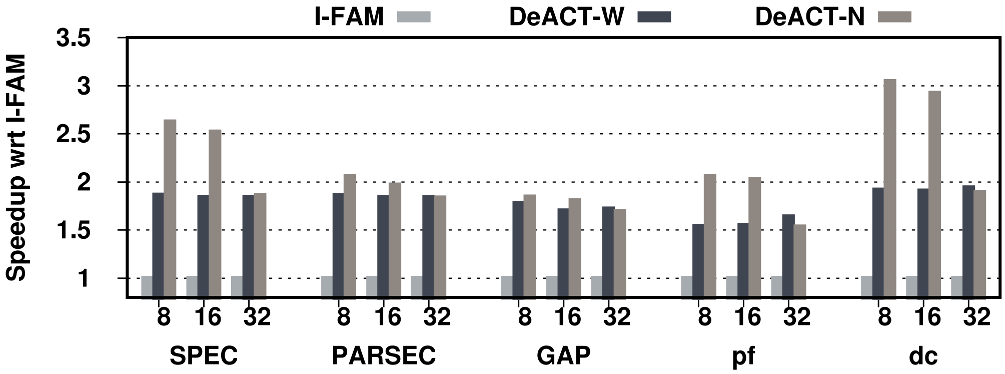

Figure 5.14: Metadata size effect on performance.

When ACM size is 8 bits, tag and metadata pair in DeACT-N requires 52 bits (44 bits for tag and 8 bits for ACM, see Section 5.3.4). Hence, in a single way, STU cache can store two tag and ACM pairs, similar to when ACM is 16 bits. However, the amount of memory required to store ACM of all the pages is reduced to half. As an experimental model, we further reduce the size of the tag to allocate three pairs of tag and ACM per STU cache way, when ACM is 8 bits. When ACM is 32 bits STU can cache only one tag and ACM pair in DeACT-N. We observe that as caching of tag and ACM pairs in the STU cache way increase, from one to three, the performance improvement with DeACT-N also improves. For instance. the system performance improves by 2.62x, 2.52x, and 1.85x when three, two, and one pairs of tag and ACM are cached in each STU cache way, for SPEC CPU 2006 benchmark. It is interesting to note that when only one pair of tag and ACM is stored in the STU cache way the performance improvement is less than or equal to DeACT-W. This is because when only one pair of tag and ACM is cached in each STU cache way the performance improvement is only due to increased address translation hits in FAM translation cache and ACM hit rate is same as I-FAM in DeACT-N.

5.4.4.3 Fabric Latency

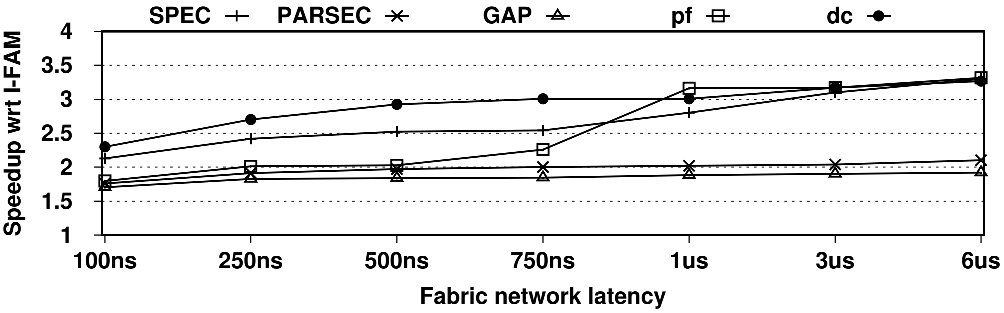

Figure 5.15: The impact of fabric latency on performance.

We propose DeACT for FAM architectures specifically and hence, one of the crucial parameter to consider for such architectures is fabric network latency. In our approach we considered 500ns as fabric latency. However, fabric networks are being explored intensively by various fabric providers [**?**, 15, 16, 106]. Previous approaches considered various fabric network latencies [96, 48, 107, 108]. Thus, we evaluated DeACT under the influence of various fabric latencies, Figure 5.15. An obvious observation is that when fabric network latency is less, the performance improvement with DeACT is also less and when the network latency is high, the performance improvement with DeACT is more. This is because when the fabric network latency is less, performance degradation in I-FAM itself is less, compared to E-FAM. This goes inversely when fabric latency is high. We see that even when fabric network latency is less, 100ns, DeACT achieves an improvement of 1.79x wrt to I-FAM. In contrast, when the network latency is 6us, DeACT speeds up I-FAM by 3.3x for *pf* benchmark.

5.4.4.4 Number of Nodes

The interconnecting fabric connects multiple PEs to the decoupled FAM modules. A single FAM module is expected to be part of a single memory pool. FAM architectures are constructed with

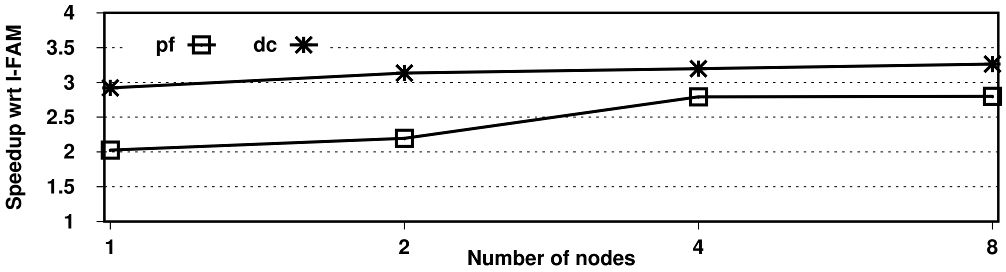

Figure 5.16: Impact of increasing the number of nodes on performance.

multiple such memory pools and PEs. The performance of FAM architectures depends on the number of PEs and the number of memory pools. We maintained memory pools directly proportional to the number of nodes and each node has four PEs. For instance, an eight node system consists of 32 PEs and eight FAM modules. Each of the PEs accesses any of the memory pools. Memory pools and the PEs are connected through a common fabric network. The delay in accessing FAM depends on the number of nodes sharing the fabric interface and memory. Although scalability is beyond the scope of this paper we evaluated our approach when multiple nodes (up to 8) share the fabric. As the number of PEs sharing the fabric increase, we observe that the slowdown due to I-FAM is higher. This is due to more cycles are consumed to fetch FAM page table entries since, the fabric network and memory are shared by the nodes. As a result, the performance improvement with DeACT is more since, DeACT avoids accessing FAM for page table entries for most of the time. When the fabric network and memory are allocated to only one node the performance improvement with DeACT is 2.92x for *dc* benchmark and it increases to 3.26x when fabric network and memory are shared between 8 nodes.

5.5 Related Work

Recently, disaggregating memory from PEs has been explored as an alternative memory architecture to overcome various operational and scalability challenges of in-node memory architectures [48, 107, 109, 47, 86, 108]. Works such as [52, **?**, 16, 110] discuss and explore fast interconnect to enable decoupling memory. However, there has been limited work discussing virtual memory and security for such systems. Lim et al. [47] discussed two stage address translations for FAM systems, but their approach is limited to using remote memory merely as a swap space. Lim et al. [47] also proposed fine-grained remote memory accesses, which is similar to E-FAM and is not secure as discussed. Shan et al. [48] proposed decoupled OS for FAM systems and the address translations are performed by the FAM modules. However, for such a scheme to work the caches must be virtually indexed and virtually tagged which is not adopted and is not a practical design. Also, it requires significant changes to the OS. Aguilera et al. [107] invalidated virtual memory paging for such huge memory designs. Aguilera et al. also proposed fixed virtual address regions for the nodes [109]. However, this requires modifications to applications' binaries. In this paper, we discuss virtual memory support for FAM architectures with two stage address translations (I-FAM) and propose DeACT scheme to accelerate address translations.

Decoupling access control and address mappings have been explored previously. Alam et al. [111] discussed decoupled address control from address mapping allowing applications to perform address translations by itself. Olson et al. [112] proposed an approach to sandbox accelerators by providing them with flat address space. DeACT uniquely leverages the architecture layout of FAM architectures when decoupling access control from translation; it allows fast caching of translations in local nodes' main memories and maintains access control in the trusted area (i.e., at system-level). Additionally, DeACT supports data sharing across nodes and leverages system-level translation units at the fabric.

Although virtual machine guests are different from nodes in FAM systems, in both cases virtual addresses are translated at two stages to access memory. A significant amount of work has been done to improve the performance of virtualized conventional machines. Bhargava et al. [49] accelerate 2D PTW by studying reuse of page entry references and extend PTW caches to temporarily cache nested dimension. Ahn et al. [113] revisited hardware-assisted page walks by speculative shadow paging mechanism, called speculative inverted shadow paging, which is backed by non-speculative flat nested page tables. The speculative mechanism provides a direct translation with a single memory reference for common cases and eliminates the page table synchronization overheads. Agile paging is proposed by Gandhi et al. [114]. Agile paging allows a virtualized page walk to start with the shadow paging for stable upper levels of the page table and allows switching in the same page walk to nested paging for lower levels of the page table which receive frequent updates. This way agile paging makes use of both shadow paging and nested paging. While these approaches improve the system performance, these are proposed for virtual machines and our approach is orthogonal to these schemes.

5.6 Conclusion

In this work, we study memory management aspects in FAM systems and comprehend that memory access latency and security are two crucial concerns with the existing management schemes for such architectures. Hence, we discuss virtual memory management in disaggregated memory systems and propose solutions to speed up address translations and provide security for such systems. While approaches like [113, 114] reduce the number of memory accesses to fetch address mapping in virtualized systems, they target native virtualized systems. Due to the hierarchical nature of the memory, disaggregated memory systems have their own challenges supporting virtual memory. We show that exposing global memory to the nodes needs OS alterations and compromise security from neighbor nodes. The virtual memory approach for disaggregated memory (indirect memory access) does not ask for OS modifications and provides security from neighbor nodes, but performs poorly. Although virtual memory support is discussed for disaggregated memory systems in [47, 48, 109, 107], in such approaches remote memory is merely used as swap space and required application and OS changes. We proposed a decoupled address translation and ACM approach to improving the performance of I-FAM. We show that an improved spatial locality of system-level translations by decoupling the system-level address translations from system-level ACM and caching the decoupled FAM translations in the local memory. We also explore ACM caching in STU cache to improve the performance. Overall, we achieved a performance improvement of up to 4.59x (1.8x on average).

LIST OF REFERENCES

[1] J. T. Pawlowski, “Hybrid memory cube: breakthrough dram performance with a fundamentally re-architected dram subsystem,” in *Hot Chips*, vol. 23, 2011.

[2] S. Liu, B. Leung, A. Neckar, S. O. Memik, G. Memik, and N. Hardavellas, “Hardware/software techniques for dram thermal management,” 2011.

[3] H. David, C. Fallin, E. Gorbatov, U. R. Hanebutte, and O. Mutlu, “Memory power management via dynamic voltage/frequency scaling,” in *Proceedings of the 8th ACM international conference on Autonomic computing*, pp. 31–40, ACM, 2011.

[4] Q. Deng, D. Meisner, L. Ramos, T. F. Wenisch, and R. Bianchini, “Memscale: active low-power modes for main memory,” in *ACM SIGPLAN Notices*, vol. 46, pp. 225–238, ACM, 2011.

[5] C.-H. Lin, C.-L. Yang, and K.-J. King, “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.

[6] S. Liu, K. Pattabiraman, T. Moscibroda, and B. G. Zorn, “Flikker: saving dram refresh-power through critical data partitioning,” *ACM SIGPLAN Notices*, vol. 47, no. 4, pp. 213–224, 2012.

[7] J. Liu, B. Jaiyen, R. Veras, and O. Mutlu, “Raidr: Retention-aware intelligent dram refresh,” in *ACM SIGARCH Computer Architecture News*, vol. 40, pp. 1–12, IEEE Computer Society, 2012.

[8] D. Wu, B. He, X. Tang, J. Xu, and M. Guo, “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.

[9] "Adaptive resource optimizer for optimal high performance compute resource utilization," pp. 1–5, Synopsys Inc, silicon to software, Mountain View, 2015.

[10] I. Foster, Y. Zhao, I. Raicu, and S. Lu, "Cloud computing and grid computing 360-degree compared," *arXiv preprint arXiv:0901.0131*, 2008.

[11] M. Chowdhury, M. Zaharia, J. Ma, M. I. Jordan, and I. Stoica, "Managing data transfers in computer clusters with orchestra," *ACM SIGCOMM Computer Communication Review*, vol. 41, no. 4, pp. 98–109, 2011.

[12] A. Khrabrov and E. De Lara, "Accelerating complex data transfer for cluster computing.," in *HotCloud*, 2016.

[13] D. Comperchio and J. Stevens, "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.

[14] G.-Z. Consortium *et al.*, "Gen-z overview," tech. rep., Tech. Rep., 2016.[Online]. Available: http://genzconsortium. org/wp-content . . . , 2016.

[15] D. Sharma, "Compute express link," *CXL Consortium White Paper.[Online]. Available: https://docs. wixstatic. com/ugd/0c1418 d9878707bbb7427786b70c3c91d5fbd1. pdf*, 2019.

[16] C. Consortium *et al.*, "Cache coherent interconnect for accelerators (ccix)." `http://www.ccixconsortium.com`, 2017.

[17] J. Taylor, "Facebook's data center infrastructure: Open compute, disaggregated rack, and beyond," in *Optical Fiber Communication Conference*, p. W1D.5, Optical Society of America, 2015.

[18] J. Kyathsandra and E. Dahlen, “Intel rack scale architecture overview,” *Proc. INTEROP, Las Vegas, NV, 2–6 May 2013*, 2013.

[19] A. Chen, “Emerging nonvolatile memory (nvm) technologies,” in *Solid State Device Research Conference (ESSDERC), 2015 45th European*, pp. 109–113, IEEE, 2015.

[20] H. Li and Y. Chen, “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] J. Handy, “Understanding the intel/micron 3d xpoint memory,” in *Proc. SDC*, 2015.

[22] A. R. Reinberg and R. C. Zahorik, “X-point memory cell,” Aug. 17 2004. US Patent 6,777,705.

[23] P. Cappelletti, “Non volatile memory evolution and revolution,” in *Electron Devices Meeting (IEDM), 2015 IEEE International*, pp. 10–1, IEEE, 2015.

[24] A. Rudoff, “The impact of the nvm programming model,” in *Storage Developer Conference*, 2013.

[25] Z. Li, R. Zhou, and T. Li, “Exploring high-performance and energy proportional interface for phase change memory systems,” in *2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*, pp. 210–221, IEEE, 2013.

[26] B. C. Lee, P. Zhou, J. Yang, Y. Zhang, B. Zhao, E. Ipek, O. Mutlu, and D. Burger, “Phase-change technology and the future of main memory,” *IEEE micro*, vol. 30, no. 1, pp. 143–143, 2010.

[27] B. C. Lee, E. Ipek, O. Mutlu, and D. Burger, “Architecting phase change memory as a scalable dram alternative,” in *ACM SIGARCH Computer Architecture News*, vol. 37, pp. 2–13, ACM, 2009.

[28] "Linux Direct Access of Files (DAX)." `https://www.kernel.org/doc/Documentation/filesystems/dax.txt`.

[29] D. T. Chakri Padala and V. Yadav, "Time for memory disaggregation? ericsson research blog," may 2017.

[30] G.-Z. Consortium *et al.*, "Gen-z–a new approach to data access," 2017.

[31] K. Michael and W. Mike, "Gen-z dram and persistent memory theory of operation." `https://genzconsortium.org/white-papers/`, 2019.

[32] C. Consortium *et al.*, "Cache coherent interconnect for accelerators (ccix)." `http://www.ccixconsortium.com`, 2017.

[33] G.-Z. Consortium, "The gen-z tenets." `https://www.youtube.com/watch?v=raed9Xxvj8Y&feature=youtu.be`, 2017.

[34] B. W. Barrett, S. Smith, J. Dinan, K. Seager, and R. E. Grant, "Sandia openshmem," tech. rep., Sandia National Lab.(SNL-NM), Albuquerque, NM (United States), 2016.

[35] J. R. Hammond, S. Ghosh, and B. M. Chapman, "Implementing openshmem using mpi-3 one-sided communication," in *Workshop on OpenSHMEM and Related Technologies*, pp. 44–58, Springer, 2014.

[36] K. Feind, "Shared memory access (shmem) routines," *Cray Research*, vol. 53, 1995.

[37] P. Grun, S. Hefty, S. Sur, D. Goodell, R. D. Russell, H. Pritchard, and J. M. Squyres, "A brief introduction to the openfabrics interfaces-a new network api for maximizing high performance application efficiency," in *2015 IEEE 23rd Annual Symposium on High-Performance Interconnects*, pp. 34–39, IEEE, 2015.

[38] "Openfabrics interfaces working group: Libfabric programmer's manual.." `https://ofiwg.github.io/libfabric/`.

[39] P. Shamis, M. G. Venkata, M. G. Lopez, M. B. Baker, O. Hernandez, Y. Itigin, M. Dubman, G. Shainer, R. L. Graham, L. Liss, *et al.*, "Ucx: an open source framework for hpc network apis and beyond," in *2015 IEEE 23rd Annual Symposium on High-Performance Interconnects*, pp. 40–43, IEEE, 2015.

[40] H. Greenberg, M. Lang, L. Ionkov, and S. Blanchard, "Redfish—resilient dynamic distributed scalable system services for exascale,"

[41] F. Bellard, "Qemu, a fast and portable dynamic translator.," in *USENIX annual technical conference, FREENIX Track*, vol. 41, p. 46, Califor-nia, USA, 2005.

[42] `https://github.com/FabricAttachedMemory/Emulation`.

[43] M. R. Meswani, S. Blagodurov, D. Roberts, J. Slice, M. Ignatowski, and G. H. Loh, "Heterogeneous memory architectures: A hw/sw approach for mixing die-stacked and off-package memories," in *High Performance Computer Architecture (HPCA), 2015 IEEE 21st International Symposium on*, pp. 126–136, IEEE, 2015.

[44] A. Awad, A. Basu, S. Blagodurov, Y. Solihin, and G. H. Loh, "Avoiding tlb shootdowns through self-invalidating tlb entries," in *Parallel Architectures and Compilation Techniques (PACT), 2017 26th International Conference on*, pp. 273–287, IEEE, 2017.

[45] P. J. Teller, "Translation-lookaside buffer consistency," *Computer*, vol. 23, no. 6, pp. 26–36, 1990.

[46] N. Amit, "Optimizing the tlb shootdown algorithm with page access tracking," in *Proc. USENIX Ann. Conf*, pp. 27–39, 2017.

[47] 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 *ACM SIGARCH Computer Architecture News*, vol. 37, pp. 267–278, ACM, 2009.

[48] Y. Shan, Y. Huang, Y. Chen, and Y. Zhang, "Legoos: A disseminated, distributed {OS} for hardware resource disaggregation," in *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*, pp. 69–87, 2018.

[49] R. Bhargava, B. Serebrin, F. Spadini, and S. Manne, "Accelerating two-dimensional page walks for virtualized systems," in *ACM SIGOPS Operating Systems Review*, vol. 42, pp. 26–35, ACM, 2008.

[50] D. Buragohain, A. Ghogare, T. Patel, M. Vutukuru, and P. Kulkarni, "Dime: A performance emulator for disaggregated memory architectures," in *Proceedings of the 8th Asia-Pacific Workshop on Systems*, p. 15, ACM, 2017.

[51] K. Asanović, "Firebox: A hardware building block for 2020 warehouse-scale computers," 2014.

[52] J. Gu, Y. Lee, Y. Zhang, M. Chowdhury, and K. G. Shin, "Efficient memory disaggregation with infiniswap.," in *NSDI*, pp. 649–667, 2017.

[53] R. A. Uhlig and T. N. Mudge, "Trace-driven memory simulation: A survey," *ACM Computing Surveys (CSUR)*, vol. 29, no. 2, pp. 128–170, 1997.

[54] S. W. Sherman and J. Browne, "Trace driven modeling: Review and overview," in *Proceedings of the 1st symposium on Simulation of computer systems*, pp. 200–207, IEEE Press, 1973.

[55] A. F. Rodrigues, K. S. Hemmert, B. W. Barrett, C. Kersey, R. Oldfield, M. Weston, R. Risen, J. Cook, P. Rosenfeld, E. CooperBalls, *et al.*, "The structural simulation toolkit," *ACM SIGMETRICS Performance Evaluation Review*, vol. 38, no. 4, pp. 37–42, 2011.

[56] V. Kommareddy, C. Hughes, S. D. Hammond, and A. Awad, "Opal: A centralized memory manager for investigating disaggregated memory systems.," Tech. Rep. SAND2018-9199, Sandia National Lab.(SNL-NM), Albuquerque, NM (United States), 2018.

[57] A. Awad, G. R. Voskuilen, S. D. Hammond, and R. J. Hoekstra, "Samba: A detailed memory management unit (mmu) for the sst simulation framework.," Tech. Rep. SAND2017-0002, Sandia National Lab.(SNL-NM), Albuquerque, NM (United States), 2017.

[58] A. Awad, G. R. Voskuilen, A. F. Rodrigues, S. D. Hammond, R. J. Hoekstra, and C. Hughes, "Messier: A detailed nvm-based dimm model for the sst simulation framework," No. SAND2017-1830, Sandia National Laboratories (SNL-NM), 2017.

[59] S. Borkar, "Networks for multi-core chips–a controversial view," in *Workshop on on-and off-chip interconnection networks for multicore systems (OCIN), Stanford*, 2006.

[60] 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," *The Role of Reactor Physics toward a Sustainable Future (PHYSOR)*, 2014.

[61] I. Karlin, J. Keasler, and J. Neely, "Lulesh 2.0 updates and changes," tech. rep., Lawrence Livermore National Laboratory (LLNL), Livermore, CA, 2013.

[62] G. Gunow, J. Tramm, B. Forget, K. Smith, and T. He, "Simplemoc-a performance abstraction for 3d moc," 2015.

[63] 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.

[64] M. A. Heroux, D. W. Doerfler, P. S. Crozier, J. M. Willenbring, H. C. Edwards, A. Williams, M. Rajan, E. R. Keiter, H. K. Thornquist, and R. W. Numrich, “Improving performance via mini-applications,” *Sandia National Laboratories, Tech. Rep. SAND2009-5574*, vol. 3, 2009.

[65] D. H. Bailey, “Nas parallel benchmarks,” in *Encyclopedia of Parallel Computing*, pp. 1254–1259, Springer, 2011.

[66] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, *et al.*, “The nas parallel benchmarks,” *The International Journal of Supercomputing Applications*, vol. 5, no. 3, pp. 63–73, 1991.

[67] P. Zhou, Y. Du, Y. Zhang, and J. Yang, “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.

[68] L. Subramanian, V. Seshadri, Y. Kim, B. Jaiyen, and O. Mutlu, “Predictable performance and fairness through accurate slowdown estimation in shared main memory systems,” *arXiv preprint arXiv:1805.05926*, 2018.

[69] H. Usui, L. Subramanian, K. Chang, and O. Mutlu, “Squash: Simple qos-aware high-performance memory scheduler for heterogeneous systems with hardware accelerators,” *arXiv preprint arXiv:1505.07502*, 2015.

[70] M. K. Jeong, M. Erez, C. Sudanthi, and N. Paver, "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.

[71] J. Zhao, O. Mutlu, and Y. Xie, "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.

[72] H. Strass, "An introduction to nvme." Seagate Technology LLC, `https://www.seagate.com/files/www-content/product-content/ssd-fam/nvme-ssd/nytro-xf1440-ssd/_shared/docs/an-introduction-to-nvme-tp690-1-1605us.pdf`, 2016.

[73] A. Arpaci-Dusseau, "Translation lookaside buffers (tlbs)," 2000.

[74] C. Villavieja, V. Karakostas, L. Vilanova, Y. Etsion, A. Ramirez, A. Mendelson, N. Navarro, A. Cristal, and O. S. Unsal, "Didi: Mitigating the performance impact of tlb shootdowns using a shared tlb directory," in *Parallel Architectures and Compilation Techniques (PACT), 2011 International Conference on*, pp. 340–349, IEEE, 2011.

[75] B. F. Romanescu, A. R. Lebeck, D. J. Sorin, and A. Bracy, "Unified instruction/translation/data (unitd) coherence: One protocol to rule them all," in *Proceedings-International Symposium on High-Performance Computer Architecture*, 2010.

[76] L. E. Ramos, E. Gorbatov, and R. Bianchini, "Page placement in hybrid memory systems," in *Proceedings of the international conference on Supercomputing*, pp. 85–95, ACM, 2011.

[77] X. Wang, "Supporting superpages and lightweight page migration in hybrid memory systems," *arXiv preprint arXiv:1806.00776*, 2018.

[78] H. Yoon, J. Meza, R. Ausavarungnirun, R. A. Harding, and O. Mutlu, "Row buffer locality aware caching policies for hybrid memories," in *Computer Design (ICCD), 2012 IEEE 30th International Conference on*, pp. 337–344, IEEE, 2012.

[79] C. Chou, A. Jaleel, and M. K. Qureshi, "Cameo: A two-level memory organization with capacity of main memory and flexibility of hardware-managed cache," in *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 1–12, IEEE Computer Society, 2014.

[80] J. Sim, A. R. Alameldeen, Z. Chishti, C. Wilkerson, and H. Kim, "Transparent hardware management of stacked dram as part of memory," in *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 13–24, IEEE Computer Society, 2014.

[81] A. Prodromou, M. Meswani, N. Jayasena, G. Loh, and D. M. Tullsen, "Mempod: A clustered architecture for efficient and scalable migration in flat address space multi-level memories," in *High Performance Computer Architecture (HPCA), 2017 IEEE International Symposium on*, pp. 433–444, IEEE, 2017.

[82] C. Chou, A. Jaleel, and M. Qureshi, "Batman: techniques for maximizing system bandwidth of memory systems with stacked-dram," in *Proceedings of the International Symposium on Memory Systems*, pp. 268–280, ACM, 2017.

[83] S. Kannan, A. Gavrilovska, V. Gupta, and K. Schwan, "Heteroos: Os design for heterogeneous memory management in datacenter," in *ACM SIGARCH Computer Architecture News*, vol. 45, pp. 521–534, ACM, 2017.

[84] F. X. Lin and X. Liu, "Memif: Towards programming heterogeneous memory asynchronously," *ACM SIGARCH Computer Architecture News*, vol. 44, no. 2, pp. 369–383, 2016.

[85] G. Piccoli, H. N. Santos, R. E. Rodrigues, C. Pousa, E. Borin, and F. M. Quintão Pereira, "Compiler support for selective page migration in numa architectures," in *Proceedings of the 23rd international conference on Parallel architectures and compilation*, pp. 369–380, ACM, 2014.

[86] K. Lim, Y. Turner, J. R. Santos, A. AuYoung, J. Chang, P. Ranganathan, and T. F. Wenisch, "System-level implications of disaggregated memory," in *High Performance Computer Architecture (HPCA), 2012 IEEE 18th International Symposium on*, pp. 1–12, IEEE, 2012.

[87] A. Awad, S. Blagodurov, and Y. Solihin, "Write-aware management of nvm-based memory extensions," in *Proceedings of the 2016 International Conference on Supercomputing*, p. 9, ACM, 2016.

[88] G. Consortium, "Gen-z zmmu and memory interleave," *Online]. https://genzconsortium.org/wp-content/uploads/2018/05/Gen-Z-MMU-and-Memory-Interleave.pdf*, July 2017.

[89] P. Express, "Address translation services revision 1.1," *Online]. https://composter.com.ua/documents/ats*$_r1.1_26Jan09.pdf, January2009$.

[90] S. Xi, J. Wilson, C. Lu, and C. Gill, "Rt-xen: Towards real-time hypervisor scheduling in xen," in *Proceedings of the ninth ACM international conference on Embedded software*, pp. 39–48, ACM, 2011.

[91] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg, "Meltdown," *arXiv preprint arXiv:1801.01207*, 2018.

[92] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, *et al.*, "Spectre attacks: Exploiting speculative execution," in *2019 IEEE Symposium on Security and Privacy (SP)*, pp. 1–19, IEEE, 2019.

[93] S. Chen, L. Liu, W. Zhang, and L. Peng, "Architectural support for nvram persistence in gpus," *IEEE Transactions on Parallel and Distributed Systems*, vol. 31, no. 5, pp. 1107–1120, 2019.

[94] A. Awad, S. Hammond, C. Hughes, A. Rodrigues, S. Hemmert, and R. Hoekstra, "Performance analysis for using non-volatile memory dimms: opportunities and challenges," in *Proceedings of the International Symposium on Memory Systems*, pp. 411–420, 2017.

[95] V. R. Kommareddy, C. Hughes, S. Hammond, and A. Awad, "Investigating fairness in disaggregated non-volatile memories," in *2019 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, pp. 104–110, IEEE, 2019.

[96] P. X. Gao, A. Narayan, S. Karandikar, J. Carreira, S. Han, R. Agarwal, S. Ratnasamy, and S. Shenker, "Network requirements for resource disaggregation.," in *OSDI*, vol. 16, pp. 249–264, 2016.

[97] V. Shrivastav, A. Valadarsky, H. Ballani, P. Costa, K. S. Lee, H. Wang, R. Agarwal, and H. Weatherspoon, "Shoal: A network architecture for disaggregated racks," in *16th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 19)*, pp. 255–270, 2019.

[98] B. Gras, K. Razavi, H. Bos, and C. Giuffrida, "Translation leak-aside buffer: Defeating cache side-channel protections with {TLB} attacks," in *27th {USENIX} Security Symposium ({USENIX} Security 18)*, pp. 955–972, 2018.

[99] M. Sutton, T. Ben-Nun, and A. Barak, "Optimizing parallel graph connectivity computation via subgraph sampling," in *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pp. 12–21, IEEE, 2018.

[100] Y. Shiloach and U. Vishkin, “An o (log n) parallel connectivity algorithm,” tech. rep., Computer Science Department, Technion, 1980.

[101] J. L. Henning, “SPEC CPU2006 benchmark descriptions,” *ACM SIGARCH Computer Architecture News*, vol. 34, pp. 1–17, sep 2006.

[102] C. Bienia, S. Kumar, J. P. Singh, and K. Li, “The parsec benchmark suite: Characterization and architectural implications,” in *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, pp. 72–81, ACM, 2008.

[103] C. Bienia and K. Li, “Parsec 2.0: A new benchmark suite for chip-multiprocessors,” in *Proceedings of the 5th Annual Workshop on Modeling, Benchmarking and Simulation*, vol. 2011, 2009.

[104] S. Beamer, K. Asanović, and D. Patterson, “The gap benchmark suite,” *arXiv preprint arXiv:1508.03619*, 2015.

[105] M. Heroux and R. Barrett, “Mantevo project,” 2016.

[106] O. Consortium *et al.*, “Opencapi specifications,” 2018.

[107] M. K. Aguilera, N. Amit, I. Calciu, X. Deguillard, J. Gandhi, P. Subrahmanyam, L. Suresh, K. Tati, R. Venkatasubramanian, and M. Wei, “Remote memory in the age of fast networks,” in *Proceedings of the 2017 Symposium on Cloud Computing*, pp. 121–127, 2017.

[108] 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*, pp. 417–427, ACM, 2019.

[109] M. K. Aguilera, N. Amit, I. Calciu, X. Deguillard, J. Gandhi, S. Novakovic, A. Ramanathan, P. Subrahmanyam, L. Suresh, K. Tati, *et al.*, “Remote regions: a simple abstraction for

remote memory," in *2018 {USENIX} Annual Technical Conference ({USENIX} {ATC} 18)*, pp. 775–787, 2018.

[110] M. S. Birrittella, M. Debbage, R. Huggahalli, J. Kunz, T. Lovett, T. Rimmer, K. D. Underwood, and R. C. Zak, "Intel® omni-path architecture: Enabling scalable, high performance fabrics," in *2015 IEEE 23rd Annual Symposium on High-Performance Interconnects*, pp. 1–9, IEEE, 2015.

[111] H. Alam, T. Zhang, M. Erez, and Y. Etsion, "Do-it-yourself virtual memory translation," in *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, pp. 457–468, IEEE, 2017.

[112] L. E. Olson, J. Power, M. D. Hill, and D. A. Wood, "Border control: Sandboxing accelerators," in *2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 470–481, IEEE, 2015.

[113] J. Ahn, S. Jin, and J. Huh, "Revisiting hardware-assisted page walks for virtualized systems," in *2012 39th Annual International Symposium on Computer Architecture (ISCA)*, pp. 476–487, IEEE, 2012.

[114] J. Gandhi, M. D. Hill, and M. M. Swift, "Agile paging: exceeding the best of nested and shadow paging," in *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pp. 707–718, IEEE, 2016.