MEMPLEX: A Memory System with Replication and Migration of Data for Multi-Chiplet NUMA Architectures

Neethu Bal Mallya
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
Gothenburg, Sweden
neethub@chalmers.se

Bhavishya Goel
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
Gothenburg, Sweden
goelb@chalmers.se

Ioannis Sourdis
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
Gothenburg, Sweden
sourdis@chalmers.se

Abstract

As the semiconductor industry struggles with the diminishing returns of Moore's law and explores innovative solutions for integrating more resources on a chip, multi-chiplet chips offer a cost-efficient alternative to large monolithic chips due to their higher yield. However, chiplet-based systems inherently exhibit Non-Uniform Memory Access (NUMA) characteristics and, therefore, suffer from slow remote accesses. Although data placement in multi-chiplet NUMA systems can be optimized in software, currently, there are no hardware mechanisms to dynamically improve data placement in DRAM distributed across chiplets. Our experiments show that this leads to wasting a significant fraction of system performance compared to a hypothetical system with ideal data placement. Our work addresses this problem by introducing MEMPLEX, a novel memory system for multi-chiplet NUMA architectures, which offers data replication and migration in the memory nodes of a multi-chiplet system. MEMPLEX allocates a small fraction of each memory node to construct a DRAM cache and offers their remaining capacity to a shared flat address space with hardware migration. In a nutshell, MEMPLEX DRAM cache attracts data of the working set to the local memory node and decides whether to migrate them upon eviction based on their usage in the cache. Thereby, MEMPLEX improves data locality, regains a large fraction of the above performance overhead, and offers substantial energy savings.

CCS Concepts

• **Computer systems organization** → **Multicore architectures**.

ICS '25, Salt Lake City, UT, USA

ACM ISBN 979-8-4007-1537-2/25/06
https://doi.org/10.1145/3721145.3725776

Keywords

Chiplets, Non-Uniform Memory Access, Caching, Migration

ACM Reference Format:
Neethu Bal Mallya, Bhavishya Goel, and Ioannis Sourdis. 2025. MEMPLEX: A Memory System with Replication and Migration of Data for Multi-Chiplet NUMA Architectures. In *2025 International Conference on Supercomputing (ICS '25), June 08–11, 2025, Salt Lake City, UT, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3721145.3725776

1 Introduction

Fitting more resources onto a chip has always been a key aspect of enhancing the performance of processor chips. This became more critical in the multicore era after Dennard scaling could no longer deliver higher frequencies due to power limitations. However, with Moore's law running out of steam, technology scaling is increasingly challenging, and integrating more resources on a single monolithic chip has become too expensive. Building larger chips out of multiple smaller chiplets offers higher yield and is thus a lower cost alternative [10, 19]; however, compared to their monolithic counterparts, chiplet-based chips have significant performance overheads [31].

Multi-chip integration technologies were initially employed to build High Bandwidth Memory (HBM) and position it closer to a processing die, such as a GPU [4] or a vector engine [36]. It was soon expanded to disintegrate processors to multiple chiplets, as seen in AMD's EPYC and RYZEN architectures, providing access to multiple memory nodes with non-uniform access latencies that vary by tens of nanoseconds [33]. Currently, multi-chiplet chips, such as AMD MI300 [1] and Intel Sapphire Rapids [35], integrate multiple CPU and/or GPU chiplets along with HBM nodes, forming part of a complex and less uniform memory system.

Non-uniform Memory Access (NUMA) machines entail the performance pitfall of long latency remote accesses, but also offer opportunities for performance optimizations, if

data locality is maximized. In the 1990s, Cache Only Memory Architectures (COMA) [8] and Cache Coherent NUMA (CC-NUMA) [55] memory systems improved data locality and, consequently, the performance of NUMA multi-socket machines by replicating (caching) and/or migrating data close to the processor chip. More recently, hybrid memory systems composed of nodes with heterogeneous characteristics, such as smaller HBM and larger but lower bandwidth external DRAM, have employed similar techniques to reduce memory access times, including, DRAM caching [5, 14–17, 24, 28, 29, 41, 50], data migration [20, 21, 32, 40, 42, 45, 51], or a combination of both [52].

The focus of this work is on NUMA architectures composed of multiple CPU chiplets and HBM nodes, such as the AMD MI300 [1] or Intel Sapphire Rapids [35]. Currently, such systems use HBM as part of a flat address space and rely on Operating System (OS) support or user optimizations to place data closer to the processing chiplet, or they use the entire HBM as a DRAM cache of an external DDR memory, thereby wasting valuable main memory capacity [35]. This paper demonstrates that even when memory allocation is NUMA-aware, placing data in the closest available memory node relative to the processing node, system performance is still significantly reduced compared to an ideal system that always finds data in its local memory node. Our aim is to alleviate this performance loss by reducing remote memory accesses.

To this end, we propose MEMPLEX, a novel memory system that offers replication and migration of data across the memory nodes of a multi-chiplet chip in order to enhance data locality. As a result, the number of accesses to the closest memory node for each processing chiplet is increased, while accesses to the remote memory nodes are minimized. This reduces the average memory access time, thereby improving system performance. The proposed memory system uses a small fraction of each HBM node as a DRAM cache and decides whether to migrate data upon eviction from that cache based on the usage of the evicted blocks. Concisely, this paper makes the following contributions:

- Investigates the performance bottlenecks in a multi-chiplet NUMA system revealing a performance loss of 26% and 31% in 4- and 16-chiplet configurations, respectively, compared to an ideal system.
- Introduces MEMPLEX, the first multi-chiplet NUMA architecture that combines replication and migration of data across multiple memory nodes, offering:
  - Most of the HBM capacity as a shared flat address space, unlike designs that use it entirely as a DRAM cache.
  - Superior performance to existing software solutions offering NUMA-aware data placement and designs using HBM exclusively as a cache.
- Evaluates MEMPLEX on multi-programmed mixes of workloads from different benchmark suites (detailed in Section 4.3), and shows that, compared to a multi-chiplet system with NUMA-aware data placement and no support for DRAM caching or migration, MEMPLEX:
  - Eliminates 80% of the remote memory traffic, resulting in a 44% reduction in dynamic memory energy consumption in a 4-chiplet system.
  - Achieves up to 7% speedup (5% on average) when $\frac{1}{16}$ of each HBM is dedicated for caching in a 4-chiplet system, with performance gains increasing up to 15% (10% on average) in 16-chiplet systems.

The remainder of this paper is organized as follows: Section 2 discusses related work, Section 3 presents the MEMPLEX architecture, Section 4 outlines our experimental setup, Section 5 presents our evaluation results, and Section 6 concludes with a summary of our findings.

2 Related Work

This section reviews prior work on data replication and migration in memory systems, focusing on: (i) traditional multi-socket NUMA systems, (ii) hybrid memory systems combining HBM and off-chip DRAM, and (iii) software support for NUMA architectures.

2.1 Non-Uniform Memory Access in Shared Memory Systems

Distributed shared memory (DSM) systems inherently deal with the problem of incurring higher delays when retrieving data mapped to a remote memory node compared to its local memory, resulting in non-uniform memory access. Optimizing latency for remote data access in DSMs has been an extensive topic of research in computer architecture for many decades. Cache-Coherent NUMA (ccNUMA) machines address this challenge by allowing the remote data to be cached in the local node's cache hierarchy [23, 25, 30, 55]. References to the remote data that miss the local node's cache hierarchy are sent to the *home node* of the referenced page. The home node is responsible for the initial allocation of the page and is in charge of maintaining the consistency and coherence of that page across the system.

Due to the relatively smaller size of the remote cache, the ccNUMA systems exhibit high sensitivity to data placement. This sensitivity can be mitigated to some extent by strategies such as caching remote data on DRAM [6, 13] or leveraging OS support for dynamic page migration to local memory [53]. Cache-only memory architecture (COMA) machines address this problem by allowing the remote pages to be freely migrated to the local memory, improving the chances of the referenced data being available locally. Since there is no concept of a home node in traditional COMA systems, block localization in case of a miss in the local memory

can be challenging and time-consuming. FLAT-COMA [46] resolves this problem by assigning a fixed home node for each page. In this scheme, the pages are free to migrate to remote nodes, but the location of the directory remains fixed. COMA systems implement block replacement and relocation mechanisms in hardware, resulting in increased hardware complexity. Simple-COMA (S-COMA) [11] systems simplify the hardware implementation by offloading some of this complexity to the operating system. When a remote page is first referenced, it results in a page fault. The operating system allocates a page frame in the local memory for the remote page and fetches the remote block into this newly allocated page frame. Subsequent references to the same block get mapped directly to the local memory. Since the physical addresses in the local memory are handled independently by the local MMU, identical blocks residing in different nodes can have different physical addresses. Consequently, nodes need a global identifier for migrated pages for inter-node communication. Hence, each node maintains a translation table responsible for converting local addresses to global addresses and vice versa.

Reactive NUMA (R-NUMA) [9] aims to combine the performance benefits of ccNUMA and S-COMA. This scheme initially allocates the remote block in the remote cache to achieve a low initial overhead cost of ccNUMA. The system keeps track of block refetching due to conflict and capacity misses to remote cache and initiates S-COMA page allocation process when refetch count exceeds a certain threshold.

2.2 Hybrid Memory Systems

DRAM-based hybrid memory systems combine two types of memory to balance performance and capacity. The first type is High Bandwidth Memory (HBM), which offers high data transfer rates but has limited capacity due to heat dissipation issues, increased cost, and stacking efficiency. To complement the HBM, conventional lower-bandwidth external DRAM is used to expand the system's overall memory capacity. Ideally, the goal is to design a memory system that seamlessly integrates the high bandwidth of HBM with the larger capacity of off-chip external DRAM, providing an efficient balance of both performance and storage.

Currently, there are two primary approaches for organizing hybrid memory systems. The first approach is to use HBM and off-chip DRAM as part of the same main memory, with a migration mechanism that brings the "hottest" data to the high-bandwidth 3D-stacked DRAM [20, 21, 32, 40, 42, 45, 51]. Some of these designs rely on OS to select and migrate data [32] and, although simpler, have a slow response to working set changes. Other data migration solutions are implemented in hardware, offering a faster response, but need to handle address remapping and keep it transparent to the OS [20, 21, 40, 42, 45, 51]. The second approach uses HBM as a DRAM cache of the external DRAM [5, 14–17, 24, 28, 29, 41, 50], with the primary challenge being the overheads associated with metadata management (tags).

A hybrid of these two approaches has also been proposed in the Hybrid^2 design, which reserves a fraction of the HBM for caching and offers its remaining capacity to the main memory [52]. Hybrid^2 targets systems with a single processor chip and 2-level hybrid memory, i.e. HBM and external DDR. In contrast, MEMPLEX extends the concept of combining replication and migration to chiplet-based systems featuring multiple processing chiplets, multiple HBM nodes, and external DDR. Unlike Hybrid^2, MEMPLEX faces more complex challenges, such as (i) the remap information needs to be fragmented and scattered to the various NUMA nodes, (ii) data allocation and migration decisions are intricate due to the varying distances between memory nodes, and (iii) introducing multiple DRAM caches (one per HBM node) calls for compatibility with directory-based cache coherence protocols [6]. MEMPLEX addresses these challenges and provides an innovative solution that offers data replication and migration in the memory system of a multi-chiplet processor, improving the performance and energy efficiency of chiplet-based systems and demonstrating scalability to larger systems with more chiplets.

Existing commercial multi-chiplet processors with HBM, such as Intel Sapphire Rapids [35], do not combine replication and migration. Their HBM nodes are either used entirely as DRAM cache wasting capacity, or as part of a flat address space with no hardware support for migration. In the latter case, software techniques can be employed to alleviate NUMA overheads, as explained next.

2.3 Software support in NUMA machines

Modern operating systems widely support NUMA architectures through various mechanisms. Operating systems like Linux [44], Windows [2], and FreeBSD [3] implement NUMA-aware scheduling algorithms to place processes and threads closer to the memory nodes, minimizing access latency. Additionally, operating systems allow user applications to discover NUMA topology, request memory from specific nodes, and set process affinity through NUMA APIs. In addition, Linux also facilitates manual page migration [22] from the remote NUMA node to the one currently running the process. The automatic NUMA balancing mechanism in Linux [12] enables periodic unmapping of process memory, NUMA hinting faults, migration-on-fault, and automatic placement of tasks closer to the memory. These operating system features can make applications NUMA-aware and improve performance on NUMA machines mostly by selecting a static mapping that places data close to the consuming threads. However, even with dynamic migration support, this approach often requires programmer intervention and

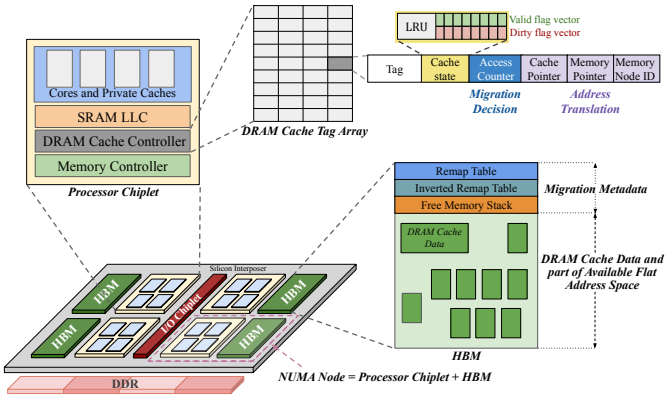


Figure 1: MEMPLEX System Overview

has a slower response to working set changes compared to hardware migration solutions [52]. In contrast, MEMPLEX aims to transparently improve performance on NUMA systems without putting an additional burden on application programmers.

3 MEMPLEX Design

MEMPLEX is a memory system for chiplet-based architectures, comprising multiple processor chiplets and HBMs integrated on a silicon interposer as well as external DDR memory accessed via an IO chiplet, as illustrated in Figure 1. Without loss of generality, the system is organized in NUMA nodes composed of a processor chiplet and an HBM. Each processor chiplet has a high bandwidth connection to its nearby, local HBM, henceforth denoted as Local Memory (LM), and can gradually reach larger parts of the shared memory at the cost of lower bandwidth, by connecting first to the remote HBMs of other nodes on the chip and second to the off-chip DDR, collectively referred to as Remote Memory (RM). On a system with such trade-offs between memory bandwidth and memory capacity, MEMPLEX improves data locality by employing a DRAM cache and a migration scheme on the shared flat address space. It allocates a small portion of each HBM in the system as the data array of a sectored DRAM cache, private to the node, and utilizes its remaining capacity, together with the external DDR, to form a shared flat address space offering hardware support for data migration across the shared HBMs and the external DDR.

3.1 MEMPLEX System Overview

The MEMPLEX system combines data replication and migration across the HBMs of a multi-chiplet chip as well as the external DDR. A fraction of each HBM is allocated to store the data array of a sectored DRAM cache, which attracts data frequently used by the cores on the local chiplet. The rest of its capacity is part of the flat address space. The DRAM Cache Tag Array (DCTA) is maintained in SRAM locally at a reasonable cost. Migration decisions are made per sector upon its eviction from the DRAM cache.

In MEMPLEX, the data management operates at distinct granularities. The data blocks in the DRAM cache are fetched at the cache line granularity (64 Bytes). The DRAM cache tags are maintained at the sector granularity, which for simplicity is equal to an OS page (4 KBytes). After a cache miss in the SRAM LLC, DCTA is the first point of reference for determining whether the requested cache line is available within the DRAM cache. The requested cache line may reside either in the LM or in RM. In the event of a tag array miss, a new entry for the missing sector is allocated in the DCTA regardless of where the requested cache line resides. However, a new data array entry for the sector is allocated in LM only if the requested cache line resides in the RM. Otherwise, if the sector is already located in the main memory part of LM, the added DCTA entry would point to the existing location of the sector in LM and mark all cache lines as dirty to ensure a writeback after eviction. Thereby, replication of data that already reside in LM is avoided, while DCTA acts as a cache of the address remap information.

An HBM is logically, rather than physically, partitioned between the DRAM cache and the flat address space, and the partitioning is facilitated by pointers maintained in the DCTA. This allows for a seamless link of sectors already present in the LM to the DRAM cache tags. Moreover, it enables cached sectors from RM to be migrated into LM without relocating the already fetched cache lines.

The sectored DRAM cache allows the tags to be kept entirely on the processor chiplet without significant SRAM cost due to its small size. This induces minimal latency to the critical memory access path as all the memory requests go through the DCTA. The tag array also contains additional information to facilitate the data migration within the shared memory. Besides the tag and cache state, each entry in the on-chip tag array stores the remapped address of the sector, serving as a cache of the migration metadata, which effectively reduces the overhead of address remapping. Section 3.2 elaborates on the DCTA structure.

When a sector is evicted from the DRAM cache, the migration mechanism decides whether to migrate it to the LM or evict it back to its current location in RM. The migration decision is based on the cost of migration in terms of the memory traffic and the number of accesses to the sector while in the DRAM cache. By deferring the migration decision until a DRAM cache eviction, the management of migration-related metadata is moved off the critical path, thereby minimizing its impact on performance. Additionally, the RM traffic generated by migrations is dynamically adjusted according to the workload behavior.

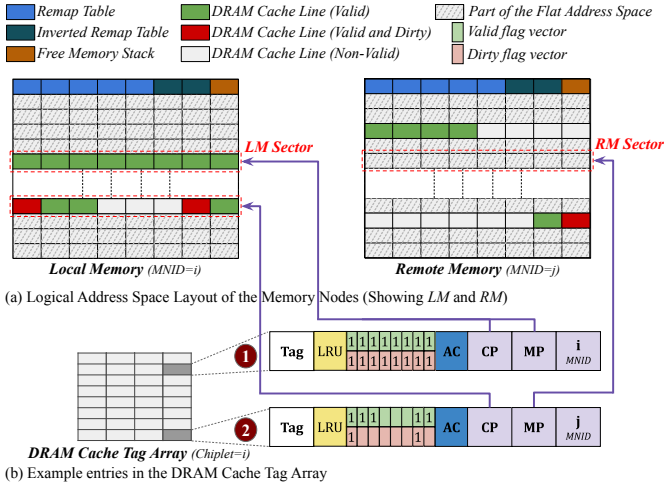


Figure 2: (a) Logical Address Space Layout of the Memory System and (b) Example entries in the DCTA

3.2 DRAM Cache Controller

Each processor chiplet in the system features a DRAM Cache Controller (DCC) responsible for high-level block management tasks. This includes handling requests from the processor, accessing the on-chip tags, fetching cache blocks on misses, evicting blocks, and generating writeback traffic to the main memory for dirty blocks. Additionally, the DCC manages sector migrations between the local and remote memory. This involves translating the addresses of remapped sectors, selecting sectors for migration to LM, and making migration decisions based on data usage (while in DC) and migration overhead considerations. Section 3.7 discusses how DCC manages migrations. All memory requests go through the DCC, which communicates with the memory controller to access the HBMs and external DDR.

The DCC manages the DRAM Cache Tag Array (DCTA), which stores all tags for the DRAM cache in SRAM on the processor chiplet. DCTA is set-associative, with each set containing entries for multiple sectors. Each entry in DCTA, as depicted in Figure 1, comprises the sector tag, state bits for each cache line in the sector (including valid and dirty bits), an access counter, two pointers, and a node identifier. The Access Counter (AC) monitors sector accesses and is used upon DRAM cache eviction to decide whether to migrate the sector to LM or evict it to RM. AC is incremented only for non-migrated LM sectors to prevent potential starvation within the cache set, ensuring LM sectors with frequent accesses are not evicted from the DCTA. Additionally, we ignore the sectors whose counters have reached the maximum value to prevent starvation from RM sectors that remain in the cache for prolonged periods. Pointers facilitate address translation of processor physical addresses to sector locations in the memory system. The Cache Pointer (CP) decouples the set and way from the physical location of data in the LM. This indirection allows our design for sector migration to LM without the need to copy data from one LM location to another. The Memory Pointer (MP) points to sector physical locations in RM and helps avoid remap table lookups. MP is same as CP for sectors that belong to the LM or for sectors that have entirely migrated to the LM. The Memory Node Identifier (MNID) is used to identify the node where the cached sector is located in the main memory. For the HBM address space, the MNID is the node ID where the HBM is located. On the other hand, the external DDR is divided into a number of regions equal to the number of nodes in the system, so each region is assigned to a different node, as illustrated in Figure 1. When the MNID matches the self-ID of a node, it indicates that the respective node has the sector in its flat address space, either as a result of migration or as the original *Home Node (HN)* location determined by the memory allocation.

3.3 Memory Layout & Metadata

Figure 2(a) illustrates the logical address space layout of the memory system, depicting the HBM local to a processor chiplet (LM) and another one that is remote (RM). The external DDR memory is not illustrated as it only contains data as part of the address space. On the contrary, each HBM includes a reserved portion containing migration metadata structures used in the MEMPLEX design. The non-reserved portion of the HBM is logically partitioned between DRAM cache data and the available flat address space across the shared memory. This means that the sector corresponding to a DCTA entry can be located anywhere across the flat address space of its respective LM (shown by the lined area in Figure 2(a)). DCC uses the pointers maintained in the DCTA to track the location of sectors within its respective LM. Sectors in LM may either fully reside in its DRAM cache (with a corresponding DCTA entry) or not at all. Sectors in RM may be partially or fully cached in the DRAM cache, also with corresponding DCTA entries.

Figure 2(b) demonstrates examples of DCTA entries. The first entry (1 in Figure 2(b)) corresponds to a sector entirely migrated to its respective LM, as indicated by CP specifying its location. In this scenario, MP is the same as CP, MNID is the node's self ID, and as a convention, all valid and dirty bits are set. The second entry (2 in Figure 2(b)) represents a sector partially cached in the DRAM cache, indicating it has not been migrated to the LM. Some cache lines of the sector have been fetched to the LM, as indicated by the valid flag vector of the DCTA entry. The dirty flag vector specifies the cache lines of the sector that were written while in the DRAM cache. The CP and MP pointers indicate the sector location in the LM and RM corresponding to MNID, respectively.

Migration Metadata Structures. Our design allows all-to-all address remapping for pages across the flat address space available in the HBMs and external DDR. To achieve this, we maintain the following structures in each memory node:

- **Remap Table:** Each node in the system maintains a remap table which stores mappings from the processor physical address to the actual memory location of the sector in the memory system. A remap table stores entries of sectors (pages), which natively belong to its node and are migrated elsewhere as well as sectors migrated to its local HBM from other parts of the memory. Unlike the centralized remap table in the Hybrid^2 design, where all remap information is stored in one location, our approach distributes remap information across NUMA nodes. This fragmentation introduces greater complexity in tracking data. To mitigate this complexity, the remap table structure is optimized by implementing it as a hash table that maintains an entry for each of the native LM sectors that have migrated to RMs and for the RM sectors that have migrated to the LM. This means that if a native LM sector gets a miss in the remap table, the sector is in its default, native location. The structure is indexed by the processor physical address and points to the memory location of the sector. On a sector migration, the remap table is updated to reflect the new address. It is worth noting that the DCTA serves as a cache for remap table entries of sectors currently (partially or fully) in the DRAM cache, facilitated by the pointers illustrated in Figure 2(b). This distributed yet optimized design enables effective memory location tracking across NUMA nodes.
- **Inverted Remap Table:** This table contains processor physical addresses corresponding to all locations within the respective memory node. The table also includes a bit map representing the sharers of the address if the sector is cached. This table is employed during the migration of blocks out of the memory node. Further details on its usage are provided in Section 3.5.
- **Free Memory Stack:** Each node maintains a stack of a minimum number of its own free locations, which currently hold no valid data and are available for use. A predefined number of entries from this stack are given exclusively to each other memory node for migrating data. Thus, in addition to its own free locations, each memory node maintains a stack of free locations reserved for use on all other memory nodes. Furthermore, when a node exhausts its available free locations, it requests additional entries to replenish its stack. The stack size is bound to the number of sectors that can fit within the DRAM cache. The stack pointer and a set number of top entries of the stack per node are stored on-chip within the DCC to minimize LM access.

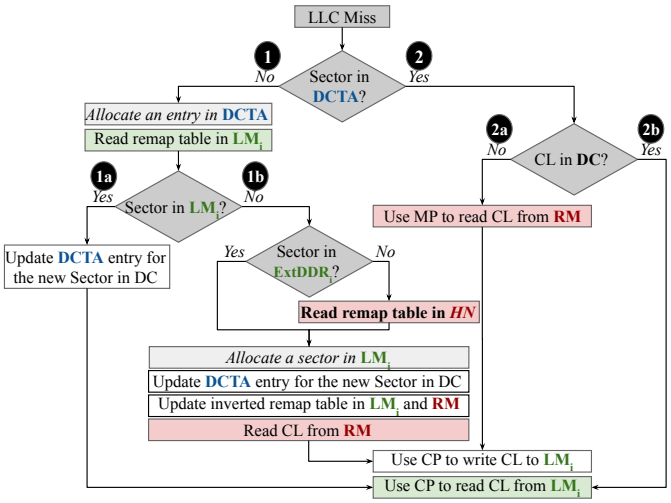


Figure 3: Memory Access Path

MEMPLEX overheads are primarily related to (1) added logic in the memory controller (for supporting migration and DRAM caching), which is similar to the overheads imposed by existing hardware migration and DRAM caching approaches, (2) SRAM cost for storing DCTA, and (3) DRAM space for metadata. The space allocated for all the above metadata is small (even when considering the full remap table) and in our implementations constitutes only 0.5% of a memory node capacity.

3.4 Memory Access Path

When a memory request arrives in the DCC of a requesting node due to an LLC miss, the DCTA is indexed with the (physical) address to determine if the requested sector and the specific cache line is available within the DRAM cache. This operation can result in one of four possible outcomes, as illustrated in Figure 3.

❶ ***DCTA Miss:*** In this scenario, the DCTA does not contain an entry corresponding to the requested sector. The requested sector may reside either in the LM or in any of the RM locations. Regardless of where the requested sector resides (LM or RM), an entry is allocated in the DCTA for that sector. Section 3.6 elaborates on the allocation of a new entry in the DCTA and the eviction process in the DRAM cache if necessary during this allocation. The address remap table in the LM is accessed using the sector's physical address to determine the sector's location in the memory system. If the remap table in LM does not contain the updated location of the requested physical address, the system defaults to accessing the remap table in the *Home Node (HN)*, decoded from the higher-order bits of the address. This is typical for the first access to a sector allocated in any of the RMs.

Subsequent requests from the same chiplet would be served from its DCTA.

1a ***Requested Sector in LM:*** If the sector is located in the LM, then all cache lines associated with that sector are already present in the LM. Consequently, the entry in the DCTA is updated accordingly. The CP and MP are set to point to the LM location of the sector. Additionally, the MNID is set to the self-ID, and all cache lines are flagged as valid and dirty.

1b ***Requested Sector in RM:*** If the sector is not located in the LM, but is in the region of external DDR that belongs to the requesting node, then the location of the sector is already known from step 1a. Otherwise, the remap table in the RM node, i.e., the node of the physical address is accessed to get the updated location of the requested physical address. Next, space is allocated in the LM for caching the new sector in the DC, and the requested cache line needs to be fetched from RM to the newly allocated location in LM. Section 3.5 elaborates on the allocation process followed by a memory node. Subsequently, the DCTA is updated with the new sector. The CP is set to point to the newly allocated LM location of the sector. The MP is set to the RM location of the sector, and the MNID is assigned the ID of the RM node. The valid flag is set only for the fetched cache line, while the dirty flag depends on the request type. Additionally, the inverted remap table in the requesting node is updated with the physical address of the sector, even though this sector has not yet been migrated to LM. This is done to ensure correctness during LM allocation, as explained in Section 3.5.

2 ***DCTA Hit:*** In this scenario, the DCTA contains an entry matching the requested sector. However, even though there is an entry for the sector in the DCTA, the requested cache line might be located in the LM or not.

2a ***Requested Cache Line not in DRAM Cache:*** In this scenario, an entry for the sector exists in the DCTA, but the specific cache line is not valid. This indicates that the sector is located in the RM, and only certain cache lines of the sector have been fetched to the DRAM cache. Subsequently, the MP pointer is utilized to retrieve the requested cache line from the RM, while the CP pointer is employed to write the cache line to the appropriate location in the LM.

2b ***Requested Cache Line in DRAM Cache:*** In this scenario, the requested cache line is located in the DRAM Cache. The sector can be located either in the LM or the RM. In either scenario, the requested cache line is accessible in the LM through the CP pointer of the DCTA entry.

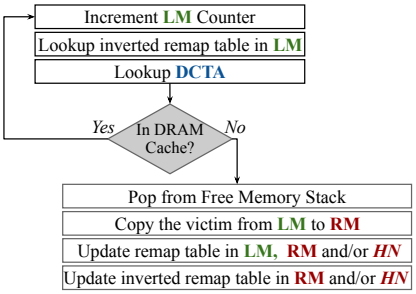


Figure 4: Allocating a sector in Local Memory

3.5 Allocating a Sector in Local Memory

When a DCTA miss occurs and is indicated that the requested sector resides in any of the RM (1b in Figure 3), a new sector must be allocated in LM. To make space for this new sector, another sector must be migrated away to any RMs. When the cache is initially empty at boot, we employ a simple counter to allocate space for the cache within LM. Figure 4 illustrates the sector allocation process in the LM. During this process, the DCC (i) identifies the victim sector in the LM, (ii) locates a free sector in the nearest RM for allocation from the Free Memory Stack, (iii) copies the data from the victim sector in the LM to the free sector in RM, and (iv) after the data is copied, the mapping structures are updated to reflect the new location of the sectors in the LM and RM.

3.5.1 Finding victim sector in LM. A FIFO policy is employed to identify a victim sector in LM. A Local FIFO counter, wrapping around all the available LM locations, is incremented each time a new location in LM is needed. However, the sector corresponding to the counter may currently be assigned to the DRAM cache (indicated by CP in the DCTA entry) or cached in any of the RMs. To handle this, the inverted remap table is indexed with the counter to obtain the sector's physical address. Then, look up the DCTA using the physical address of the sector. If the sector is in the DCTA, we proceed to the next one until finding an available sector. This ensures correctness, as a sector in the DRAM cache must not be migrated to RM. Furthermore, this approach yields a better replacement decision than FIFO alone, as sectors frequently accessed are more likely to reside in the DRAM cache and avoid migration to RM. To minimize the latency of this step, which is in the critical path of an access, each DCC maintains a buffer of a few (e.g., two) spare, unused DC data entries ready to be used as victim sectors.

3.5.2 Finding free sector in RM. To locate a free sector in RM, we utilize the free entries of RM stored in the Free Memory Stack of the node. When a sector is migrated from RM to LM, its original RM location is pushed onto the Free Memory Stack of the RM, making it available to be overwritten.

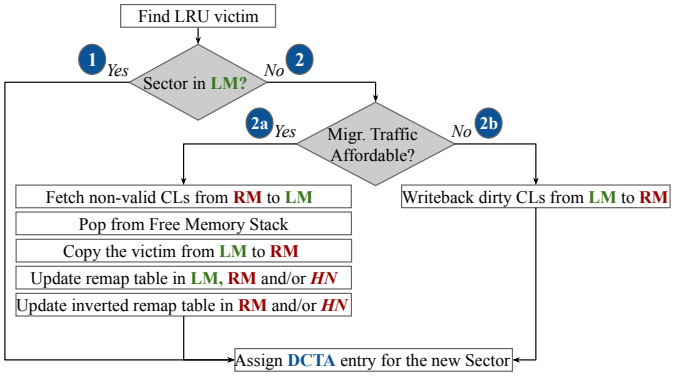


Figure 5: Allocating an entry in DCTA

3.6 DRAM Cache Evictions

Figure 5 depicts the DRAM cache eviction logic, where the DCC employs the LRU algorithm to determine which sector to evict from the DRAM cache. The DRAM cache can contain (i) Sectors already in the LM, (ii) Sectors that have migrated to the LM, or (iii) Sectors located in the RM, with some or all cache lines already fetched to the LM.

3.6.1 Evicting sectors already in LM. For sectors in cases (i) and (ii) involving data already in the LM or migrated to it (1 in Figure 5), no data movement is necessary. The remap table has been updated with the evicted sector's location during migration to LM, and the inverted remap table has been updated with the physical address of the evicted sector when first fetched in the DRAM cache. Thus, the corresponding DCTA entry can be reassigned.

3.6.2 Evicting sectors located in RM. For sectors in case (iii) located in any of the RMs, the DCC determines whether to migrate the sector to the LM or evict it back to the RM. Migration to LM (2a in Figure 5) involves fetching all non-valid cache lines of the sector from the RM and updating migration structures. In contrast, eviction (2b in Figure 5) involves writing back all dirty cache lines of the sector to the RM, and no remapping data structures need to be updated. The algorithm for deciding between migration and eviction is detailed in the following section (Section 3.7).

3.7 Migration Decision and Traffic Regulation

This section discusses the mechanism employed to regulate the migration traffic overheads and the process of deciding between migration and eviction.

3.7.1 Migration Traffic Overheads. When evicting a sector from the DRAM cache that has not been migrated to the LM, DCC has two choices: Either (i) evict the sector back to the RM, requiring the writeback of all the dirty cache lines (2b in Figure 5), or (ii) migrate the sector to the LM by fetching the non-valid cache lines of the sector from the RM (2a in Figure 5). The choice between the two is made based on the migration overhead, which is calculated in terms of the number of RM accesses caused by a migration decision and, in essence, indicate memory traffic cost. The number of RM accesses depends on the number of valid and dirty cache lines within the sector in the DRAM cache.

In the case of eviction, the RM accesses (E_{RM}) correspond to the number of dirty cache lines (N_{dirty}) that must be written back to RM. However, in the case of migration, the RM accesses (M_{RM}) are determined by two factors: firstly, the number of cache lines that need to be fetched from RM, calculated by subtracting the number of valid cache lines (N_{valid}) already present in the sector from the total number of cache lines per sector (N_{all}); and secondly, the cost of swapping out the evicted sector from LM to accommodate the new one, which necessitates N_{all} writebacks to RM.

$$E_{\text{RM}} = N_{\text{dirty}} \tag{1}$$

$$M_{\text{RM}} = (N_{\text{all}} - N_{\text{valid}}) + N_{\text{all}} \tag{2}$$

Thus, the overhead incurred in migrating a sector in terms of RM accesses (O_{m}) is given by the equation:

$$O_{\text{m}} = E_{\text{RM}} - M_{\text{RM}} + 1 = 2 \times N_{\text{all}} - N_{\text{valid}} - N_{\text{dirty}} + 1 \tag{3}$$

where the constant "1" is added as a minimum overhead. The O_{m} can range from 1, indicating all cache lines of a sector are valid and dirty, to $2 \times N_{\text{all}}$, which occurs when only one cache line of a sector is valid and clean upon eviction from the DRAM cache. Nevertheless, this latency overhead does not impact the critical path of memory access, as migration decisions are made only during evictions.

3.7.2 Balancing Migration and Processors Traffic. DCC maintains a remote access counter to monitor RM accesses, distinguishing between migration and processor requests. The counter is incremented for every DRAM cache miss that must be fetched from the RM. When a sector is migrated, the counter is decremented by its migration overhead (O_{m}). Besides monitoring the RM accesses, DCC also checks the number of sector accesses (The field AC in DCTA). If enabled, this check ensures that the value of AC of a sector is greater than AC of other sectors in the same cache set. The conjecture is that if this check is successful, the sector is likely to be reused again and thus worth keeping in LM.

When deciding on a sector for migration, its O_{m} is compared to the remote access counter. If O_{m} is smaller and the

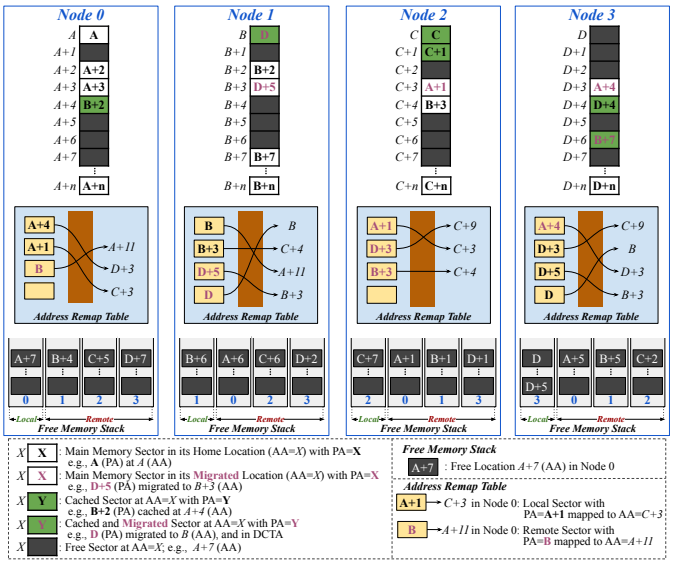


Figure 6: Example snapshot of HBM contents and metadata for a 4-node MEMPLEX system. The mapping of sectors with physical addresses (PA - bold), to actual addresses (AA - *italics*) of the machine is illustrated, as well as the contents of the remap table and free memory stack for each node.

above check is met, the sector is considered for migration. Essentially, the remote access counter acts as an upper bound on the number of RM accesses for migration and is periodically reset (every 100K cycles) to adapt to workload phase changes. The check on AC regulate eligible sectors for migration, striking a balance between data "hotness" (usage) and migration cost to optimize system performance. These checks occur during eviction and hence do not affect the critical path of a memory access.

3.8 An Example Illustration

Figure 6 provides an illustrative example of how MEMPLEX handles migration metadata. In this example, there are four nodes in the system, each displaying certain memory entries, a portion of the address remap table, and the free memory stack. For simplicity, regions of external DDR are omitted. The physical address of the sector is denoted by **A** (in bold), and the actual address (location within the memory node) is denoted by *A* (in italics). The entire address range includes **A**, **A+1**, ..., **A+n**, **B**, ..., **B+n**, **C**, ..., **C+n**, **D**, ..., **D+n**, divided across 4 memory nodes. Node 0 is the Home Node for physical addresses **A** to **A+n**, containing locations *A* to *A+n*, Node 1 for **B** to **B+n**, and so on for the remaining nodes. If the sector migrates from its Home Node, it will have an Owner Node where the sector currently resides.

Various scenarios of sector placement and migration are shown: (i) Sectors placed in their native location, e.g., sector with physical address PA=**A** placed in its actual address AA=*A*; (ii) Cached sectors, e.g., sector with PA=**B+2** from Node 1 cached in AA=*A+4* of Node 0 while maintaining its data in AA=*B+2* of Node 1. The corresponding entry for this cached sector would be in the DCTA (not shown) of Node 0; (iii) Migrated sectors, e.g., sector with PA=**D** from Node 3 migrated to location AA=*B* in Node 1. The address remap tables in Node 1 and Node 3 reflect this migration, and the entry is also in the DCTA (not shown) of Node 1. Since location **D** has been migrated out of Node 3, it is now available and added to the Free Memory Stack.

An interesting scenario involves sectors with PA=**C** and PA=**C+1**, which are placed in their native location in Node 2 and are cached in the DRAM cache of the same node as they are used by the respective processor chiplet. Another notable scenario occurs when Node 0 misses in its DRAM cache for a line in sector **D+5**. Since the sector is not found in Node 0's remap table, it must consult the remap table of sector **D+5**'s home node, i.e., Node 3. The Node 3 remap table indicates that the sector has migrated to Node 1, at AA=*B+3*, where the requested line is found after adding the line offset.

3.9 Cache Coherence

In MEMPLEX, each processor chiplet is paired with a memory node, which allocates a portion of its memory as a private DRAM cache. This DRAM cache can store pages from both local and remote memory nodes. Since a page may reside in multiple DRAM caches simultaneously, a cache coherence protocol is necessary to ensure coherence across the different memory nodes. While the specific cache coherence mechanism for DRAM caches is not detailed in this proposal—leaving room for future optimizations—MEMPLEX is designed to be compatible with a directory-based protocol, such as the one described in CANDY [6].

The home node of a cache line can either be statically assigned or dynamically determined by consulting the remap table, which locates the current placement of a sector. The coherence directory is stored in DRAM within each memory node to mitigate the SRAM storage overhead. Additionally, an SRAM cache of the directory is maintained on the processor chiplet to store recently accessed entries. This hybrid approach helps reduce both the substantial storage overhead and the access latency typically associated with coherence directories in the memory nodes. However, the migration policy for shared sectors warrants reconsideration, as sector usage in a single DRAM cache may not provide enough data for optimal decision-making and could conflict with the migration policies of other nodes. Leveraging the sector's directory to collect information from all sharers could facilitate more efficient migration decisions.

The cache coherence optimizations and their evaluation with multi-threaded workloads are left for future work. Nevertheless, even multi-programmed workloads show a significant reduction in remote traffic, noticeable performance gain, and substantial energy gains, as detailed in Section 5.

4 Experimental Setup

4.1 System Configuration

Our microarchitectural simulation offers detailed modeling of the memory and interconnects, as outlined in Section 4.2, making it computationally intensive for large systems. To keep the simulation times of our experiments within affordable bounds (tens of hours per simulation point), the modeled systems are scaled down to a quarter of a real one. A full-scale AMD Zen4C chiplet consists of 16 cores; therefore, our performance analysis focuses on chiplets scaled to one-quarter of the AMD Zen4C or Intel Sapphire Rapids chiplets. Consequently, these scaled-down chiplets contain only a quarter of the number of cores and connect to a quarter of the HBM channels, as detailed in Table 1. Additionally, the L2 and L3 caches are undersized to put more pressure on the memory system and increase LLC misses per kilo instructions (MPKI), which is otherwise difficult to achieve when simulating systems for only a few billion instructions.

Based on the scaled down chiplet size (16.5 mm^2)[1] the microbump budget is calculated to be proportional to the number of cores it includes. In addition, the following parameters were used for calculating the microbump budget: (i) a microbump pitch of 45 µm, (ii) reserving 40% of the microbumps for power. Then, the number of microbumps available for data were allocated for (i) connecting to the HBM channels, (ii) one bidirectional link to the IO chiplet, (iii) multiple bidirectional links to the other CPU chiplets. Then the width of the links to IO and CPU chiplets, as well as the total number of links to other CPU chiplets were adjusted to fit the microbump budget. Finally, the latency of the inter-chiplet links was measured to be 2 or 3 (NoC) cycles according to the chiplet's dimensions and the latency of the links on a passive silicon interposer similar to [7, 47].

To motivate the use of chiplet-based architectures, we measured the costs of 4, 8, and 16 16-core chiplets, i.e., full scale, with the above configuration compared to their equivalent hypothetical monolithic chips using the chiplet actuary model by Feng and Ma [10]. We considered (i) processor chiplets of size 66 mm^2 manufactured at 5 nm, (ii) a 400 mm^2 IO chiplet at 14 nm, and (iii) passive interposer in 65 nm technology. The analysis showed that the recurring engineering cost of chiplet-based systems were 52-55% of their monolithic counterparts[31].

[1]Calculated based on Zen4 after scaling down L2 and L3 sizes proportional to the capacity indicated in Table 1.

Table 1: System Configuration[1]

System	
Chiplets	4 chiplets[1]
Cores and Caches	
Cores	4 cores[1] / chiplet, out-of-order, 3.2 GHz
TLB	I-TLB: 512-entry, 4-way, 1 cycle latency D-TLB: 512-entry, 4-way, 1 cycle latency
L1 Cache	L1-I: Private, 32KB, 4-way, 2 cycle access latency L1-D: Private, 32KB, 4-way, 2 cycle access latency
L2 Cache	Private, 256KB, 8-way, 4 cycle access latency
L3 Cache	Shared, 1MB/core, 16-way, 12 cycle access latency[2]
Main Memory	
HBM2	1 GB/chiplet, 2 GHz, 4 channels, 128 bits per channel, tCAS-tRCD-tRP: 14-14-14 ns, RD/WR+I/O Energy = 6.4 pJ/bit
DDR4	4 GB, 3.2 GHz, 1 channel, 64 bits per channel, tCAS-tRCD-tRP: 22-22-22 ns, RD/WR+I/O Energy = 33 pJ/bit
Network	
Intra-chiplet	2 GHz, 3-stage router (VA/SA, ST, LT), 2x3 Mesh, 4 VCs per port, credit-based flow control, 256 bit link for data, 154 bit link for control (coherence) traffic, 5 flit buffers, XY Routing
Inter-chiplet	2 GHz, 3-stage router (VA/SA, ST, LT), 2x2 Mesh, passive interposer, 2 to 3 cycle link latency[3], 7 to 9 flit buffers [3]

[1] This configuration is the default setting. The parameter adjustments are detailed in the respective evaluation sections of the sensitivity studies.
[2] L3 access latency is 8 cycles for 2MB, 12 for 4MB, and 15 for 8MB.
[3] Depending on the maximum inter-chiplet link length [47].

4.2 Simulation Setup

MEMPLEX is evaluated using BZSim [48], which has been extended to model the memory system and interconnects of chiplet-based chips. BZSim is based on the ZSim simulator [43] integrated with BookSim2 [18] for cycle-accurate intra- and inter-chiplet network modeling, enhanced with a technique to detect and skip simulation of low contention traffic to speedup simulation times [48]. BZSim offers microarchitectural simulations with detailed (cycle-accurate) interconnect modeling at an order of magnitude faster simulation speeds compared to GEM5, enabling multi-billion instruction experiments within reasonable times [48]. DRAMSim3 [26] was used for cycle-accurate DRAM modeling and CACTI [54] for estimating cache access times.

The system treats all HBM and external DDR memory as part of a unified flat address space. The virtual memory system was implemented based on HSCC [27]. The cores are configurable with translation lookaside buffers (TLBs) for both instructions and data, as well as with page table walkers (PTWs). Additionally, the memory management modules include a NUMA-aware (distance-aware) allocation policy. This policy allocates pages to the HBM in the chiplet where they are first accessed. If space is not available in the nearest HBM, they are allocated in the next neighboring HBM or in the external DDR.

Table 2: Workload Characteristics

Benchmark	Label	Input	LLC MPKI	Footprint (GB)	Assigned to Mixes mix-id$^{\#of\ instances}$
LLC MPKI 20-40					
pageRank[2]	PRL2	LDBC (100k)	37.41	0.84	1^3,2^3,3^2,4^3,5^3,6^3,7^1
mcf[1]	MCF	Default	34.01	0.45	2^2,5^3,7^2
graphColoring[2]	GCL2	LDBC (100k)	30.70	0.45	1^1,2^1,4^1,6^1,7^2
graphColoring[2]	GCL3	LDBC (10k)	21.26	0.09	1^2,2^1,3^2,5^2,7^1
Random Access Workload[3]	RAND	N=30, M=1000, chunk=1024	20.83	0.70	1^1,2^3,3^1,4^2,5^2,6^1,7^1
LLC MPKI 10-20					
connectedComp[2]	CCL3	LDBC (10k)	19.33	0.09	1^3,2^2,3^1,4^3,5^2,6^2,7^1
lbm[1]	LBM	Default	18.19	0.40	3^1,6^2,7^1
BFS[2]	BFSCR	CA RoadNet	17.25	0.64	1^1,2^1,3^1,4^2,6^2,7^1
fotonik3d[1]	FOTO	Default	17.07	0.59	1^1,4^1
pageRank[2]	PRL3	LDBC (10k)	13.96	0.09	2^1,4^1
xalancbmk[1]	XAL	Default	13.62	0.16	1^1,2^1,3^2,4^1,5^2,6^3,7^1
blender[1]	BLEN	Default	12.78	0.08	2^1,4^1
shortestPath[2]	SPCR	CA RoadNet	12.30	0.64	1^1,3^2,6^1,7^1
XSBench[4]	XSB	XXL	11.11	0.37	7^1
graphColoring[2]	GCCR	CA RoadNet	10.69	0.63	1^1,3^1,5^1,7^1
LLC MPKI 0-10					
parest[1]	PAR	Default	8.54	0.05	3^1
roms[1]	ROMS	Default	7.58	0.25	7^1
triangleCount[2]	TCL2	LDBC (100k)	6.24	0.55	5^1,7^1
graphColoring[2]	GCL1	LDBC (1000k)	5.92	0.29	1^1,4^1,6^1
pageRank[2]	PRKR	Knowledge Repo	4.56	0.30	3^1
BFS[2]	BFSL1	LDBC (1000k)	2.71	0.98	3^1

[1] SPEC CPU 2017 [37], [2] GraphBIG [34], [3] GUPS [39], [4] XSBench [49]

4.3 Workloads

Multi-programmed workloads are used in our experiments from the SPEC CPU2017 benchmark suite [37] (the seven with highest MPKI), GraphBIG [34], Random access workload from the GUPS suite [39] and XSBench [49]. For the SPEC CPU2017 and GraphBIG benchmarks, we use Simpoints [38] to select a representative slice of one billion instructions. We have chosen 21 different workloads, detailed in Table 2, and created random multi-programmed mixes mapping one benchmark to each core. Each mix of applications has a minimum total memory footprint of 7 GB and a geometric mean LLC MPKI of at least 11. To scale these mixes for systems with 32 or 64 cores, we replicate the 16-application mix twice for the 32-core system and four times for the 64-core system. All experiments run with an average of 125 million instructions per core warm-up period, where memory allocation is enabled, followed by an average of 250 million instructions per core of detailed simulation.

4.4 Evaluated Systems

In the evaluation, we consider four distinct systems that offer unique approaches to managing memory resources in a NUMA multi-chiplet architecture.

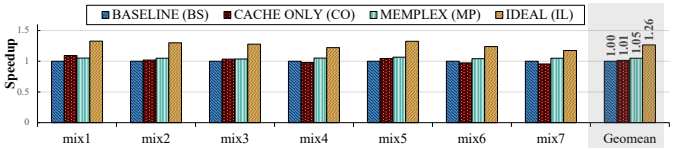
(1) **Baseline (BS)**: A multi-chiplet system with private LLC, NUMA-aware data placement and no support for DRAM caching or migration. The default configuration (depicted in Table 1) includes 4 chiplets, each with 4 cores and private LLC, integrated on a passive interposer with 256 bit NoC data-links, 4 HBM channels per chiplet, 1 link to IO chiplet, and 1 channel to external DDR.
(2) **DRAM Cache-Only (CO)**: A multi-chiplet system in which each chiplet uses its entire local HBM as a private DRAM cache. The workload mixes are calibrated to fit within a main memory that combines the capacities of HBMs and external DDR. To ensure a fair comparison, the external DDR size in the **CO** is increased to accommodate the workload mix.
(3) **MEMPLEX (MP)**: A multi-chiplet system with NUMA-aware data placement, where a fraction of the HBM is used as a private DRAM cache for each chiplet, while the remaining portion serves as part of the main memory, with data migration support.
(4) **Ideal (IL)**: A multi-chiplet system that operates under the ideal scenario in which an LLC miss is always resolved by the nearest HBM channel, assuming infinite capacity of the local HBM. This setup allows memory allocation solely within the local HBM, ensuring that all memory requests remain local to the chiplet and eliminating the latency associated with accessing remote HBM or external DDR.

5 Evaluation

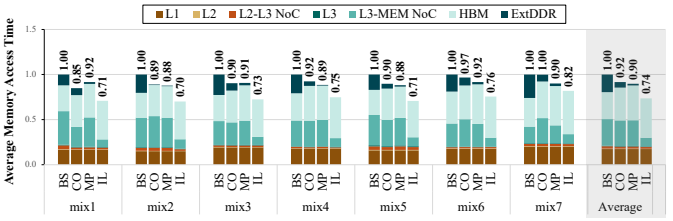
5.1 Performance

We evaluate the system performance using Instructions Per Cycle (IPC) as the primary metric. Additionally, Average Memory Access Time (AMAT) is measured and broken down to: (i) the access time for each cache level, (ii) the Network-on-Chip (NoC) latency between each level (L2-L3 and L3-MEM), and (iii) the DRAM access time. Furthermore, the percentage of accesses to local and remote HBM nodes, as well as to external DDR are reported.

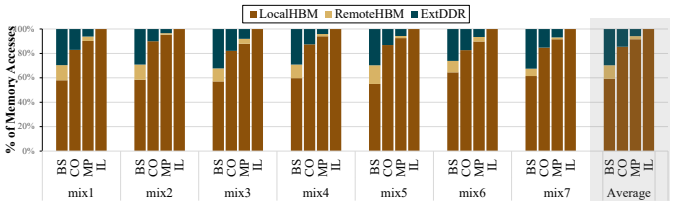
Figure 7 illustrates, for each workload mix, the performance speedup (in terms of IPC), the average memory access time, and the distribution of DRAM accesses. These metrics are presented for the default configuration of a 4-chiplet system (**BS**), a DRAM cache-only design (**CO**), the MEMPLEX design with a 1:16 DRAM cache to Main Memory ratio (**MP**), and an ideal system (**IL**). The MEMPLEX system improves baseline performance by 3-7%, with an average improvement of 5%, as shown in Figure 7(a). The DRAM cache-only design exhibits a mixed trend across the analyzed workload mixes, with an average performance improvement of 1% over the baseline. This variation is attributed to the differing number of requests, and thus remote traffic to the external DDR for fetching cache lines into the DRAM cache, which is also reflected in the AMAT numbers. In comparison, the ideal system—26% faster than the baseline—achieves this by always directing LLC misses to the nearest HBM. However,



(a) IPC (Normalized to Baseline)

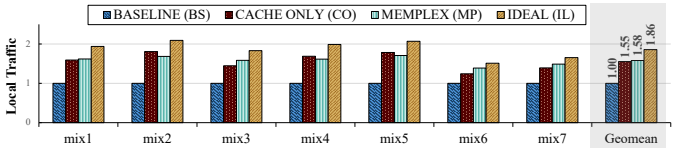


(b) Average Memory Access Time (AMAT) (Normalized to Baseline)

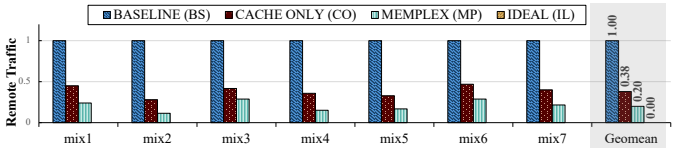


(c) Distribution of Memory Accesses

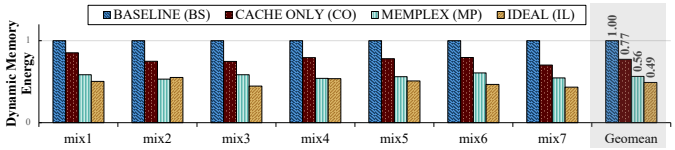
Figure 7: Performance, AMAT and distribution of memory access comparison between Baseline, DRAM cache-only, MEMPLEX and Ideal designs.



(a) Local Traffic (Normalized to Baseline)



(b) Remote Traffic (Normalized to Baseline)



(c) Dynamic Memory Energy (Normalized to Baseline)

Figure 8: Local and Remote Memory Traffic, and Dynamic Memory Energy Consumption normalized to the Baseline.

MEMPLEX, while 17% slower than the ideal system, still represents a significant improvement over the baseline, effectively reducing the impact of remote requests that are inherent in multi-chiplet NUMA architectures. This positions MEMPLEX as a more efficient alternative, narrowing the performance gap toward ideal scenarios.

The speedup achieved by MEMPLEX over the baseline is in line with the decrease in AMAT which is on average 10%, as shown in Figure 7(b). A closer look reveals that the main source of performance overhead in the chiplet-based system is the significant portion of data accesses placed remotely. As illustrated in Figure 7(c), the baseline system experiences an average of 39% of remote data accesses, comprising 10% to remote HBM and 29% to external DDR. Both types of remote accesses involve slow inter-chiplet communication due to limited bandwidth and higher latency. When the entire HBM is dedicated to caching, the DRAM cache-only design reduces remote accesses to just 14%, all of which are directed to external DDR. Meanwhile, MEMPLEX effectively addresses this challenge by bringing 90% of the data within the local HBM, leaving only 10% requiring access to remote memory, thereby significantly reducing the performance impact of remote memory accesses.

5.2 Memory Traffic

Figure 8(a) and Figure 8(b) show the local and remote memory traffic normalized to the the baseline multi-chiplet system for all workload mixes. The remote memory traffic includes both remote HBM and external DDR accesses. The DRAM cache-only design increases local memory traffic by 55% compared to the baseline. However, this benefit is offset by a 62% decrease in remote traffic to the external DDR, highlighting a trade-off that limits its overall performance. In contrast, MEMPLEX delivers a more balanced and effective solution. It generates more local memory traffic compared to the baseline system, with 58% more requests being served from the local HBM. This increase in local traffic is bolstered by an impressive 80% reduction in remote memory traffic, which outperforms the 62% reduction achieved by DRAM cache-only design.

Thus, MEMPLEX effectively combines DRAM caching and data migration to achieve a notable reduction in remote traffic, translating to improved performance and energy savings, as described next. However, the performance gains are not fully maximized, as the MEMPLEX design still involves accesses to the external DDR. Furthermore, unlike the DRAM cache-only design, which dedicates the entire HBM for caching, MEMPLEX sacrifices only $\frac{1}{16}$ of the HBM capacity, demonstrating a far greater resource efficiency.

5.3 Energy Consumption

Figure 8(c) presents the dynamic memory system energy consumption normalized to the baseline multi-chiplet system for all workload mixes. The DRAM cache-only design exhibits a 23% reduction in dynamic energy consumption compared to the baseline system. This improvement stems primarily from caching, which significantly reduces remote accesses to external DDR—a major source of energy consumption in the baseline architecture. In comparison, MEMPLEX delivers a remarkable 44% reduction in dynamic memory energy consumption relative to the baseline system. This significant decrease is primarily attributed to lower accesses to remote HBM and external DDR, effectively minimizing high-energy operations and leveraging efficient local memory access. Processor energy and static memory energy (refresh energy) are not reported as these are largely proportional to runtime.

5.4 Sensitivity analysis on System Size

An important focus of sensitivity analysis is the effect of system size. Chiplet-based designs are expected to scale more cost-effectively; however, it is still uncertain how their performance overheads change as the system size increases while keeping the number of cores per chiplet constant. To investigate this, in addition to the above evaluated 4-chiplet MEMPLEX system (16-core system), the performance of a 16-chiplet system, which corresponds to 64 cores, is assessed as shown in Figure 9(a). As observed above, in a 4-chiplet system, the ideal design would offer 26% higher performance than the baseline and MEMPLEX delivers on average 5% of that speedup and up to 7% for a single mix. As expected, on a 16-chiplet system, the NUMA overheads increase, and so do the MEMPLEX gains. In this case, the ideal design would perform 31% better than the baseline, and MEMPLEX speedup is up to 15% for a single mix and 10% on average.

5.5 Sensitivity analysis on DRAM Cache Size

MEMPLEX can be configured with various DRAM cache sizes, and this design choice significantly influences both the performance and the size of the DCTA. Figure 9(b) presents the results of a sensitivity analysis exploring different DRAM Cache to Main Memory ratios (1:8, 1:16, 1:32) in comparison to the baseline system without MEMPLEX. In terms of area overhead, the DCTA requires 512 kB, 1 MB, or 2 MB for these respective ratios, assuming an 8-byte entry size. The average performance improvements for the 1:8, 1:16, and 1:32 ratios are 8%, 5%, and 4%, respectively. Notably, the 1:8 ratio can achieve up to a 10% speedup in specific scenarios.

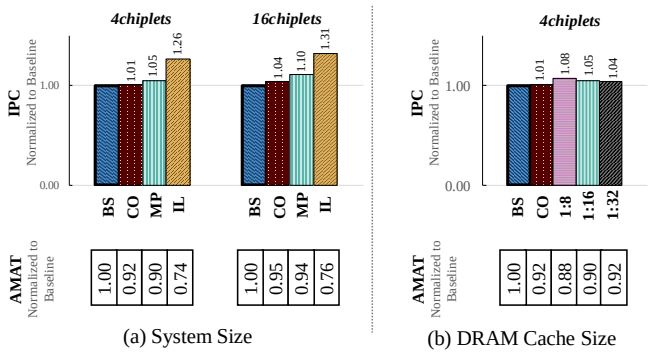


Figure 9: Sensitivity analysis on system size and DRAM cache size. All values are the geometric mean of all workload mixes normalized to the Baseline.

6 Conclusion

Multi-chiplet chips provide a cost-effective solution by delivering higher manufacturing yields. However, they encounter performance challenges due to the NUMA memory architecture and inter-chiplet communication bottlenecks. In this study, we analyzed these overheads, showing that an ideal 4- and 16-chiplet system would offer 26% and 31% higher performance, respectively, compared to a baseline with NUMA-aware data placement. To address these challenges, we proposed MEMPLEX, a novel architecture that enables data replication and migration across multiple memory nodes within a multi-chiplet system. MEMPLEX efficiently dedicates a portion of each memory node for a DRAM cache, while the remaining capacity is utilized as a shared flat address space with hardware migration. Thereby, MEMPLEX enhances data locality, bringing frequently accessed data closer to the processor and managing migration based on usage patterns. As a result, MEMPLEX reduces remote memory traffic by 80%, leading to a significant 44% dynamic memory energy consumption. In a 4-chiplet system, MEMPLEX achieves up to 7% speedup and an average of 5% when dedicating $\frac{1}{16}$ of each HBM for caching. When $\frac{1}{8}$ of the HBM capacity is used for caching, the performance gain increases to up to 10%, with an average of 8%. Finally, the performance benefits of MEMPLEX performance are more pronounced in larger systems, with the average speedup improving from 5% to 10% as the system size increases from 4 to 16 chiplets.

Acknowledgments

This work was supported by the Swedish Foundation for Strategic Research (contract number CHI19-0048) under the PRIDE project. We also express our gratitude to the anonymous reviewers for their valuable comments.

References

[1] Advanced Micro Devices, Inc. 2023. AMD CDNA™ 3 Architecture. Retrieved 7 Feb 2025 from https://www.amd.com/content/dam/amd/en/documents/instinct-tech-docs/white-papers/amd-cdna-3-white-paper.pdf

[2] Karl Bridge. 2022. NUMA Support - Win32 Apps. Retrieved 7 Feb 2025 from https://learn.microsoft.com/en-us/windows/win32/procthread/numa-support

[3] Adrian Chadd. 2018. FreeBSD Manual Pages. Retrieved 7 Feb 2025 from https://man.freebsd.org/cgi/man.cgi?query=numa&sektion=4&manpath=FreeBSD%2B14.0-RELEASE%2Band%2BPorts

[4] Jack Choquette, Wishwesh Gandhi, Olivier Giroux, Nick Stam, and Ronny Krashinsky. 2021. NVIDIA A100 Tensor Core GPU: Performance and Innovation. *IEEE Micro* 41, 2 (Mar 2021), 29–35. https://doi.org/10.1109/MM.2021.3061394

[5] Chiachen Chou, Aamer Jaleel, and Moinuddin K. Qureshi. 2015. BEAR: Techniques for Mitigating Bandwidth Bloat in Gigascale DRAM Caches. In *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*. 198–210. https://doi.org/10.1145/2749469.2750387

[6] Chiachen Chou, Aamer Jaleel, and Moinuddin K. Qureshi. 2016. CANDY: Enabling Coherent DRAM Caches for Multi-Node Systems. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 1–13. https://doi.org/10.1109/MICRO.2016.7783738

[7] Ayse Coskun, Furkan Eris, Ajay Joshi, Andrew B. Kahng, Yenai Ma, and Vaishnav Srinivas. 2018. A Cross-Layer Methodology for Design and Optimization of Networks in 2.5D Systems. In *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. 1–8. https://doi.org/10.1145/3240765.3240768

[8] Fredrik Dahlgren and Josep Torrellas. 1999. Cache-Only Memory Architectures. *Computer* 32, 6 (Jun 1999), 72–79. https://doi.org/10.1109/2.769448

[9] Babak Falsafi and David A. Wood. 1997. Reactive NUMA: A Design For Unifying S-COMA And CC-NUMA. In *24th Annual International Symposium on Computer Architecture (ISCA)*. 229–240. https://doi.org/10.1145/264107.264205

[10] Yinxiao Feng and Kaisheng Ma. 2022. Chiplet Actuary: A Quantitative Cost Model and Multi-Chiplet Architecture Exploration. In *Proceedings of the 59th ACM/IEEE Design Automation Conference (DAC)*. 121–126. https://doi.org/10.1145/3489517.3530428

[11] Erik Hagersten, Ashley Saulsbury, and Anders Landin. 1994. Simple COMA Node Implementations. In *1994 Proceedings of the Twenty-Seventh Hawaii International Conference on System Sciences (HICSS)*, Vol. 1. 522–533. https://doi.org/10.1109/HICSS.1994.323136

[12] Jiri Herrmann. 2019. Automatic NUMA Balancing Red Hat Enterprise Linux 7. Retrieved 7 Feb 2025 from https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/7/html/virtualization_tuning_and_optimization_guide/sect-virtualization_tuning_optimization_guide-numa-auto_numa_balancing

[13] Cheng-Chieh Huang, Rakesh Kumar, Marco Elver, Boris Grot, and Vijay Nagarajan. 2016. C^{3}D: Mitigating the NUMA Bottleneck via Coherent DRAM Caches. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 1–12. https://doi.org/10.1109/MICRO.2016.7783739

[14] Cheng-Chieh Huang and Vijay Nagarajan. 2014. ATCache: Reducing DRAM Cache Latency via a Small SRAM Tag Cache. In *2014 23rd International Conference on Parallel Architecture and Compilation Techniques (PACT)*. 51–60. https://doi.org/10.1145/2628071.2628089

[15] Hakbeom Jang, Yongjun Lee, Jongwon Kim, Youngsok Kim, Jangwoo Kim, Jinkyu Jeong, and Jae W. Lee. 2016. Efficient Footprint Caching for Tagless DRAM Caches. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 237–248. https://doi.org/10.1109/HPCA.2016.7446068

[16] Djordje Jevdjic, Gabriel H. Loh, Cansu Kaynak, and Babak Falsafi. 2014. Unison Cache: A Scalable and Effective Die-Stacked DRAM Cache. In *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 25–37. https://doi.org/10.1109/MICRO.2014.51

[17] Djordje Jevdjic, Stavros Volos, and Babak Falsafi. 2013. Die-stacked DRAM Caches for Servers: Hit Ratio, Latency, or Bandwidth? Have It All with Footprint Cache. In *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA)* (Tel-Aviv, Israel). 404–415. https://doi.org/10.1145/2485922.2485957

[18] Nan Jiang, Daniel U. Becker, George Michelogiannakis, James Balfour, Brian Towles, D. E. Shaw, John Kim, and William J. Dally. 2013. A Detailed and Flexible Cycle-accurate Network-on-Chip Simulator. In *2013 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 86–96. https://doi.org/10.1109/ISPASS.2013.6557149

[19] Ajaykumar Kannan, Natalie Enright Jerger, and Gabriel H. Loh. 2015. Enabling Interposer-based Disintegration of Multi-core Processors. In *2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 546–558. https://doi.org/10.1145/2830772.2830808

[20] Apostolos Kokolis, Dimitrios Skarlatos, and Josep Torrellas. 2019. PageSeer: Using Page Walks to Trigger Page Swaps in Hybrid Memory Systems. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 596–608. https://doi.org/10.1109/HPCA.2019.00012

[21] Jagadish B. Kotra, Haibo Zhang, Alaa R. Alameldeen, Chris Wilkerson, and Mahmut T. Kandemir. 2018. CHAMELEON: A Dynamically Reconfigurable Heterogeneous Memory System. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 533–545. https://doi.org/10.1109/MICRO.2018.00050

[22] Christoph Lameter and Minchan Kim. 2016. Page Migration. Retrieved 7 Feb 2025 from https://www.kernel.org/doc/Documentation/vm/page_migration

[23] James Laudon and Daniel Lenoski. 1997. The SGI Origin: A ccNUMA Highly Scalable Server. In *24th International Symposium on Computer Architecture (ISCA)*. 241–251. https://doi.org/10.1145/384286.264206

[24] Yongjun Lee, Jongwon Kim, Hakbeom Jang, Hyunggyun Yang, Jangwoo Kim, Jinkyu Jeong, and Jae W. Lee. 2015. A Fully Associative, Tagless DRAM Cache. In *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*. 211–222. https://doi.org/10.1145/2749469.2750383

[25] Daniel Lenoski, James Laudon, Kourosh Gharachorloo, W-D Weber, Anoop Gupta, John Hennessy, Mark Horowitz, and Monica S. Lam. 1992. The Stanford Dash Multiprocessor. *Computer* 25, 3 (Apr 1992), 63–79. https://doi.org/10.1109/2.121510

[26] Shang Li, Zhiyuan Yang, Dhiraj Reddy, Ankur Srivastava, and Bruce Jacob. 2020. DRAMsim3: A Cycle-Accurate, Thermal-Capable DRAM Simulator. *IEEE Computer Architecture Letters (CAL)* 19, 2 (Jul 2020), 106–109. https://doi.org/10.1109/LCA.2020.2973991

[27] Haikun Liu, Yujie Chen, Xiaofei Liao, Hai Jin, Bingsheng He, Long Zheng, and Rentong Guo. 2017. Hardware/Software Cooperative Caching for Hybrid DRAM/NVM Memory Architectures. In *Proceedings of the International Conference on Supercomputing (ICS)*. 26:1–26:10. https://doi.org/10.1145/3079079.3079089

[28] Gabriel Loh and Mark D. Hill. 2012. Supporting Very Large DRAM Caches with Compound-Access Scheduling and MissMap. *IEEE Micro* 32, 3 (May 2012), 70–78. https://doi.org/10.1109/MM.2012.25

[29] Gabriel H. Loh and Mark D. Hill. 2011. Efficiently Enabling Conventional Block Sizes for Very Large Die-stacked DRAM Caches. In *2011 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 454–464. https://doi.org/10.1145/2155620.2155673

[30] Tom Lovett and Russell Clapp. 1996. STiNG: A CC-NUMA Computer System for the Commercial Marketplace. In *23rd Annual International Symposium on Computer Architecture (ISCA)*. 308–317. https://doi.org/10.1145/232973.233006

[31] Neethu Bal Mallya, Panagiotis Strikos, Bhavishya Goel, Ahsen Ejaz, and Ioannis Sourdis. 2025. A Performance Analysis of Chiplet-Based Systems. In *Design, Automation and Test in Europe Conference (DATE)*. 1–7. https://doi.org/10.23919/DATE64628.2025.10992969

[32] Mitesh R. Meswani, Sergey Blagodurov, David Roberts, John Slice, Mike Ignatowski, and Gabriel H. Loh. 2015. Heterogeneous Memory Architectures: A HW/SW Approach for Mixing Die-stacked and Off-package Memories. In *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*. 126–136. https://doi.org/10.1109/HPCA.2015.7056027

[33] Samuel Naffziger, Noah Beck, Thomas Burd, Kevin Lepak, Gabriel H. Loh, Mahesh Subramony, and Sean White. 2021. Pioneering Chiplet Technology and Design for the AMD EPYC™ and Ryzen™ Processor Families : Industrial Product. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. 57–70. https://doi.org/10.1109/ISCA52012.2021.00014

[34] Lifeng Nai, Yinglong Xia, Ilie G. Tanase, Hyesoon Kim, and Ching-Yung Lin. 2015. GraphBIG: Understanding Graph Computing in the Context of Industrial Solutions. In *SC '15: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–12. https://doi.org/10.1145/2807591.2807626

[35] Nevine Nassif, Ashley O. Munch, Carleton L. Molnar, Gerald Pasdast, Sitaraman V. Lyer, Zibing Yang, Oscar Mendoza, Mark Huddart, Srikrishnan Venkataraman, Sireesha Kandula, Rafi Marom, Alexandra M. Kern, Bill Bowhill, David R. Mulvihill, Srikanth Nimmagadda, Varma Kalidindi, Jonathan Krause, Mohammad M. Haq, Roopali Sharma, and Kevin Duda. 2022. Sapphire Rapids: The Next-Generation Intel Xeon Scalable Processor. In *2022 IEEE International Solid-State Circuits Conference (ISSCC)*. 44–46. https://doi.org/10.1109/ISSCC42614.2022.9731107

[36] NEC CORPORATION. 2018. SX-Aurora TSUBASA Architecture Guide Revision 1.1. Retrieved 7 Feb 2025 from https://sxauroratsubasa.sakura.ne.jp/documents/guide/pdfs/Aurora_ISA_guide.pdf

[37] Reena Panda, Shuang Song, Joseph Dean, and Lizy K. John. 2018. Wait of a Decade: Did SPEC CPU 2017 Broaden the Performance Horizon?. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 271–282. https://doi.org/10.1109/HPCA.2018.00032

[38] Erez Perelman, Greg Hamerly, and Brad Calder. 2003. Picking Statistically Valid and Early Simulation Points. In *2003 12th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 244–255. https://doi.org/10.1109/PACT.2003.1238020

[39] Steven J. Plimpton, Ron Brightwell, Courtenay Vaughan, Keith Underwood, and Mike Davis. 2006. A Simple Synchronous Distributed-Memory Algorithm for the HPCC RandomAccess Benchmark. In *IEEE International Conference on Cluster Computing (Cluster 2006)*. 1–7. https://doi.org/10.1109/CLUSTR.2006.311859

[40] Andreas Prodromou, Mitesh Meswani, Nuwan Jayasena, Gabriel Loh, and Dean M. Tullsen. 2017. MemPod: A Clustered Architecture for Efficient and Scalable Migration in Flat Address Space Multi-level Memories. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 433–444. https://doi.org/10.1109/HPCA.2017.39

[41] Moinuddin K. Qureshi and Gabe H. Loh. 2012. Fundamental Latency Trade-off in Architecting DRAM Caches: Outperforming Impractical SRAM-Tags with a Simple and Practical Design. In *2012 45th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 235–246. https://doi.org/10.1109/MICRO.2012.30

[42] Jee Ho Ryoo, Mitesh R. Meswani, Andreas Prodromou, and Lizy K. John. 2017. SILC-FM: Subblocked InterLeaved Cache-Like Flat Memory Organization. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 349–360. https://doi.org/10.1109/HPCA.2017.20

[43] Daniel Sanchez and Christos Kozyrakis. 2013. ZSim: Fast and Accurate Microarchitectural Simulation of Thousand-core Systems. In *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA)*. 475–486. https://doi.org/10.1145/2485922.2485963

[44] Kanoj Sarcar. 1999. What is NUMA? Retrieved 7 Feb 2025 from https://www.kernel.org/doc/html/v5.4/vm/numa.html

[45] Jaewoong Sim, Alaa R. Alameldeen, Zeshan Chishti, Chris Wilkerson, and Hyesoon Kim. 2014. Transparent Hardware Management of Stacked DRAM as Part of Memory. In *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 13–24. https://doi.org/10.1109/MICRO.2014.56

[46] Per Stenstrom, Truman Joe, and Anoop Gupta. 1992. Comparative Performance Evaluation of Cache-Coherent NUMA and COMA Architectures. In *Proceedings the 19th Annual International Symposium on Computer Architecture (ISCA)*. 80–91. https://doi.org/10.1109/ISCA.1992.753306

[47] Dylan Stow, Yuan Xie, Taniya Siddiqua, and Gabriel H. Loh. 2017. Cost-Effective Design of Scalable High-performance Systems Using Active and Passive Interposers. In *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. 728–735. https://doi.org/10.1109/ICCAD.2017.8203849

[48] Panagiotis Strikos, Ahsen Ejaz, and Ioannis Sourdis. 2024. BZSim: Fast, Large-Scale Microarchitectural Simulation with Detailed Interconnect Modeling. In *2024 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 167–178. https://doi.org/10.1109/ISPASS61541.2024.00025

[49] John R Tramm, Andrew R Siegel, Tanzima Islam, and Martin Schulz. 2014. XSBench - The Development and Verification of a Performance Abstraction for Monte Carlo Reactor Analysis. In *Proceedings of the International Conference on Physics of Reactors (PHYSOR 2014)*.

[50] Evangelos Vasilakis, Vassilis Papaefstathiou, Pedro Trancoso, and Ioannis Sourdis. 2019. Decoupled Fused Cache: Fusing a Decoupled LLC with a DRAM Cache. *ACM Transactions on Architecture and Code Optimization (TACO)* 15, 4, Article 65 (Jan 2019), 23 pages. https://doi.org/10.1145/3293447

[51] Evangelos Vasilakis, Vassilis Papaefstathiou, Pedro Trancoso, and Ioannis Sourdis. 2019. LLC-Guided Data Migration in Hybrid Memory Systems. In *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 932–942. https://doi.org/10.1109/IPDPS.2019.00101

[52] Evangelos Vasilakis, Vassilis Papaefstathiou, Pedro Trancoso, and Ioannis Sourdis. 2020. Hybrid^2: Combining Caching and Migration in Hybrid Memory Systems. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 649–662. https://doi.org/10.1109/HPCA47549.2020.00059

[53] Ben Verghese, Scott Devine, Anoop Gupta, and Mendel Rosenblum. 1996. Operating System Support for Improving Data Locality on CC-NUMA Compute Servers. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (Cambridge, Massachusetts, USA). 279–289. https://doi.org/10.1145/237090.237205

[54] Steven J.E. Wilton and Norman P. Jouppi. 1996. CACTI: An Enhanced Cache Access and Cycle Time Model. *IEEE Journal of Solid-State Circuits (JSSC)* 31, 5 (1996), 677–688. https://doi.org/10.1109/4.509850

[55] Zheng Zhang and Josep Torrellas. 1997. Reducing Remote Conflict Misses: NUMA with Remote Cache versus COMA. In *Proceedings Third International Symposium on High-Performance Computer Architecture (HPCA)*. 272–281. https://doi.org/10.1109/HPCA.1997.569686