Stream-Based Data Placement for Near-Data Processing with Extended Memory

Yiwei Li[†] Boyu Tian[†] Yi Ren[†] Mingyu Gao[†‡]
Tsinghua University[†] Shanghai Qi Zhi Institute[‡]
liyw19@mails.tsinghua.edu.cn tby20@mails.tsinghua.edu.cn
yi-ren20@mails.tsinghua.edu.cn gaomy@tsinghua.edu.cn

***Abstract*—The data access bottleneck in memory-intensive applications has motivated various architectural innovations in the main memory system, with Near-Data Processing (NDP) and Compute Express Link (CXL) as two recent prominent representations. In this work, we focus on addressing the memory capacity limitation of 3D-stacked NDP systems using CXL-based extended memory, where the DRAM space of the 3D NDP stacks is used as the cache of the CXL-based memory. Nevertheless, this architecture exhibits unique challenges to address the significant interconnect latency and expensive metadata management problems. We propose NDPExt, a hardware-software co-design approach to achieve efficient NDP with extended memory. On the hardware side, NDPExt uses coarse-grained data streams rather than conventional fine-grained cachelines to manage the NDP stacks as a distributed DRAM cache, in order to reduce metadata cost and apply custom optimizations to different data. On the software side, NDPExt periodically derives the optimized cache configuration to allocate the DRAM cache space to each stream based on profiled miss behaviors. The configuration co-optimizes capacity sizing, spatial placement, and data replication. Combining the two techniques allows NDPExt to achieve 1.41× on average and up to 2.43× performance improvements over state-of-the-art cache management solutions.**

***Index Terms*—near-data processing, CXL, caching, streams**

I. Introduction

Nowadays, big data, artificial intelligence, and in-memory databases have posed continuously increasing demands on the memory systems of modern computers, due to their large data footprints and excessive access rates. Data accesses to and from the conventional DRAM-based main memory have emerged as a crucial performance and energy bottleneck, known as the "memory wall" [52]. In order to adapt to such urgent demands of memory-intensive applications, two emerging architectures have gained significant attention recently. On one hand, Near-Data Processing (NDP) systems [11], [18], [23], [26], [38], [41], [46], [55] move the computations close to where the data reside, thereby enabling high-bandwidth and low-latency data accesses and reducing data movement overheads. On the other hand, the Compute Express Link (CXL) specification [16] outlines the interaction between the host processor's memory and the peripheral memories, making it efficient and convenient to extend system memory capacities [24], [25], [30], [31], [49], [51].

However, for those NDP systems based on 3D-stacked memory technologies [35], [54], despite the abundant data bandwidth to the memory dies and the flexible interconnection across the logic dies, a significant limitation is the overall capacity, which is currently limited to only about six stacks with several gigabytes each [14], [32]. We therefore propose to extend the current 3D-stacked NDP system with external memory modules that are connected through the CXL interface. We name such a design as *NDP with extended memory*. In this architecture, the memory regions of the NDP stacks are used as a distributed DRAM cache, while the extended CXL-based memory serves as the OS-visible physical memory space. To each core in one of the NDP stacks, some cache space is local while the others are remote, naturally exhibiting non-uniform access characteristics similar to the conventional non-uniform cache architecture (NUCA) [42].

However, NDP has its unique challenges, such as the much more severe interconnect latencies when accessing remote cache locations across the 3D memory stacks, and the substantial overheads of maintaining and looking up the metadata of the distributed DRAM cache. Previous NUCA solutions [6], [7], [56], [71] fail to address these two issues, because they do not sufficiently consider the data placement issue to reduce the interconnect distance, and they cannot adopt efficient and flexible data replication due to the metadata constraint. Even though NDP and NUCA are both well-explored topics, the above unique challenges make it ineffective to directly combine existing techniques and call for new innovations.

In this work, we propose NDPExt, an optimized architecture of NDP with extended memory, leveraging two hardware-software co-designed optimizations to address the above challenges. On the hardware side, NDPExt adopts software-defined data streams [74]–[76] as a coarse-grained abstraction to organize the distributed DRAM cache as *stream caches*. Compared to the traditional fine-grained, cacheline-level management, the stream cache not only reduces the metadata storage and lookup cost, but also offers a nice level of abstraction on which we can apply flexible and customized placement and replication strategies to individual data streams without much complexity.

On the software side, NDPExt periodically derives the *optimized cache configuration schemes* which partition and allocate the distributed DRAM space to the many different streams in the workload. The configuration algorithm is executed by a software runtime on the host processor, taking as input the cache miss behaviors of all the streams that are profiled by specially designed hardware samplers. We propose efficient sampling methods that can scale to many different

2379-315/24/$31.00 ©2024 IEEE
DOI 10.1109/MICRO61859.2024.00120

streams and many NDP cores in the system. Compared to existing NUCA algorithms, our algorithm co-optimizes the sizing of each stream cache and the placement across different cores. It also supports more flexible data replication where each stream can use its own custom replication scheme.

When evaluated on a wide range of memory-intensive workloads, NDPExt significantly outperforms the state-of-the-art NUCA solutions [6], [7], [56], [71] by 1.41× on average, and up to 2.43×. The reasons for such improvements are mainly two-fold: the much lower metadata access overheads enabled by our hardware stream cache design, and the better data placement and replication enabled by our software configuration algorithm. Overall, NDPExt represents a novel design that efficiently alleviates the memory capacity limitation of 3D-stacked NDP systems, and contributes towards large-scale practical NDP adoption.

II. Background and Related Work

A. Emerging Memory Architectures

We first introduce the two emerging techniques: Near-Data Processing (NDP) and Compute Express Link (CXL).

Near-Data Processing (NDP). NDP moves processing logic closer to data locations to enjoy the high bandwidth and low latency of data access and to reduce the data movement overheads. Currently, NDP systems targeting the main memory can be broadly categorized into those that retain the original DIMM form factor, and those that rely on 3D-stacked memories (e.g., HBM and HMC [35], [54]). DIMM-based NDP systems add computing logic on the separate rank-level buffer chips [11], [18], [41], [46], [55] or near the internal DRAM banks [23], [26], [38]. Because there are only links between the memory controller and each DIMM, the interconnect bandwidth among the NDP logic remains limited if not applying hardware modifications [70], [81], which restricts their use in complex applications. In contrast, 3D-stacked NDP systems put computing logic on the bottom dies of 3D memory stacks, where the memory dies on the top support fast data accesses [1], [8], [17], [20], [21], [43], [45], [69], [79], [82]. High-bandwidth interconnects could also be implemented on and between the logic dies with on-chip and off-chip links, connecting them into a memory network [44], [58], [60], [78] that supports flexible communication.

However, compared to DIMMs, 3D-stacked memory devices currently suffer from limited capacity due to fundamental technology difficulties. First, the number of memory stacks is constrained by the interposer area, with rapidly increasing cost for more stacks and more complex routing [39]. Second, the capacity of each stack is constrained by the die-stacking difficulty and their thermal conductivity. For example, each 3D stack such as HMC and HBM currently is limited to only a few gigabytes [35], [54]. Even when connected using multiple stacks, the total memory capacity is far below that of a typical modern server, which could have up to several terabytes.

Compute Express Link (CXL). The CXL standard [16] has enabled efficient and convenient interaction with the peripheral memories on the I/O buses. It encompasses several protocols tailored for different use cases. CXL.io provides a basic protocol for device configuration on top of PCIe. CXL.cache allows accelerators to cache data from the host memory and ensures coherence. CXL.mem maps the device memory addresses into the host address space for direct load/store accesses.

In this work, we primarily concentrate on the CXL.mem feature for memory pooling [24], [25], [49], [51], such as memory expanders as large as 512 GB per module [59]. To configure a CXL memory pool, at the boot time, the host-side driver recognizes and initializes the CXL-enabled devices. This includes mapping the control interface and the actual CXL memory into the host address space. Once mapped, any load/store to this space is handled by the CXL controller, which in turn accesses the corresponding CXL memory. CXL.mem enables efficient memory accesses with no polling or syscall software overheads. Compared with conventional DIMMs and networking-based memory pooling, CXL is more scalable. First, memory bandwidth can substantially improve with more PCIe lanes, e.g., with the most recent CXL 3.0 + PCIe 6.0, the bandwidth ranges from 7.6 GB/s (x1) to 121 GB/s (x16). Second, memory capacity can also grow using additional CXL switches to reach terabyte-scale pooling. Third, CXL.mem promises fewer pins without the need for multiple individual DDR controllers. For instance, an 8-lane CXL port can offer bandwidth comparable to a DDR5-4800 channel, using 5× fewer pins [12], [25], [49], [66].

B. Non-Uniform Cache Architecture (NUCA)

In classical chip multi-processors, the growing chip area and the increasing core count both lead to the increasingly larger last-level cache (LLC) capacity realized through multiple individual banks spread across the chip. The access latencies from a specific core to these different banks become non-uniform, motivating prior research to optimize such a non-uniform cache architecture (NUCA) [42]. Unlike traditional centralized caches, NUCA requires complex placement, migration, and replication strategies. NUCA designs can be categorized into static (S-NUCA) and dynamic (D-NUCA), based on how flexibly a cacheline could be migrated across banks. D-NUCA can be further classified as shared-cache-based [6], [13], [28], [56], [71] and private-cache-based [5], [9], [47], [61]. Shared-cache-based designs view all the banks as a shared cache and remap cachelines to different locations, while private-cache-based designs use cache coherence protocols to look up and migrate cachelines among the banks. Several state-of-the-art shared-cache-based D-NUCA designs have considered fine-grained and access-pattern-aware data placement. Jigsaw [6] partitioned the cache across multiple threads according to way-based sampling results. Whirlpool [56] distinguished different data structures during partitioning. Nexus [71] further supported replication for read-only data.

C. Stream Basics

Streams are a coarse-grained abstraction that expresses the memory address range and the expected access pattern within the range [57]. In this paper, we mainly focus on two types of

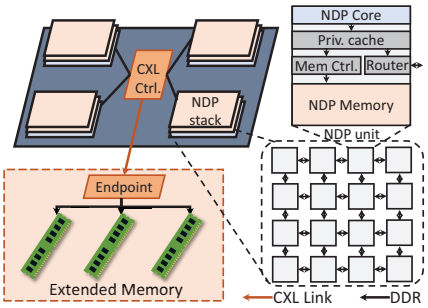

Fig. 1. Proposed architecture of NDP with extended memory. The NDP stacks use dedicated extended memory connected using CXL, which can be directly accessed without involving the host processor.

streams: affine and indirect. An affine stream has statically determined access addresses following an affine function, e.g., addr $= a \times i + b$, resembling the common sequential and strided access patterns. On the other hand, the addresses in an indirect stream are dynamically determined by the input data, e.g., addr $= s[i]$ where s is another affine or indirect stream. Distinguishing between affine and indirect streams provides hints about the access patterns for different data in the workload, enabling us to use different management mechanisms tailored for each address range. Prior work also used stream programming hints to facilitate prefetching [74], instruction offloading [75], [76], and automated parallelization [73].

III. NDP with Extended Memory

In this work, we focus on 3D-stacked NDP systems and address their main limitation of memory capacity, as discussed in Section II-A. Following the seminal idea of memory hierarchy in conventional computer architecture research, the fast but small NDP memory devices could be extended with more traditional memory modules that have larger capacity but possibly slower access speed. We call this architecture *NDP with extended memory*. While adding extended memory could now support large-scale applications, it does incur higher latency and energy to access, and may offset the benefits of NDP if not well designed. Therefore, careful architectural optimizations must be done to ensure we could achieve the best of both worlds, i.e., the fast access speed of NDP, and the large memory capacity of extended memory. We first discuss the choices of the overall architecture in Section III-A, and its specific design challenges in Section III-B. We propose novel optimizations to address the challenges in the next sections.

A. Overall Architecture

NDP. Our NDP modules align with prior 3D-stacked NDP designs [1], [8], [17], [20], [21], [43], [45], [69], [79], [82]. As shown in Fig. 1, several NDP stacks are interconnected with inter-stack links. Each stack consists of 16 NDP units arranged in an internal 4×4 mesh, with each NDP unit including the computing logic on the bottom die and the corresponding memory region above. The *NDP unit* is the basic hardware granularity in our architecture. We use general-purpose, energy-efficient cores as the NDP logic, but other types such as reconfigurable logic and ASIC can also fit in our architecture. Besides the computing logic, each unit has a memory controller to access the corresponding local memory, and a router to access remote memory regions through the inter-stack and intra-stack networks. We connect the NDP modules with the host processor through PCIe interfaces [1], similar to accelerators and GPUs. Coarse-grained program kernels could be offloaded to the NDP modules.

Connecting NDP to extended memory. We further connect CXL-based extended memory modules to the above NDP stacks, through a central CXL controller as shown in Fig. 1. The requests from NDP are parsed by the CXL endpoint in the extended memory, which then issues DDR commands to the corresponding memory channels. In this design, the NDP system and the extended memory module can be manufactured fully independently by different vendors to save the integration cost, as long as both expose the standard CXL interface to the other. The host processor also does not need any modification. We note that while we specifically use CXL, other technologies, such as adding traditional DIMM-based memory, or reusing the host memory by relaying accesses through the host processor, are also possible. Nevertheless, it is believed that CXL has much better scalability in bandwidth, capacity, and pin count than traditional DDR (Section II-A), and will gradually replace DDR in the future [12]. Because our system does not have any backward-compatibility issues as CPUs and GPUs, CXL is a particularly suitable choice.

Cache vs. part-of-memory. As we now have both the fast and small NDP memory and the slow and large extended memory, another decision to make is whether to manage the NDP as a cache of the extended memory space, or expose both as parts of the physical memory visible to the OS. This is a well-studied debate in previous heterogeneous memory research [37], [50], [53], [62], [67], [72]. In this work, we assume a cache-based approach, where data are initially loaded into the extended memory, and fetched on demand to the NDP stacks at runtime. This choice is preferred for its simplicity and transparency to the application software. In addition, the NDP stack capacity is relatively small compared to the extended memory, so treating the former as part of the memory does not significantly increase the overall capacity. Furthermore, we manage the combination of the NDP memory across all stacks as a global, distributed cache. Compared to making each NDP unit use its own private cache space, this would enable better data sharing and reuse, especially for applications with fine-grained irregular access patterns like graph computing.

B. Design Challenges

To our best knowledge, NDPExt is the first work to use the entire NDP memory as a cache, and to extend the system capacity using CXL-based memory. The key design goal in NDPExt is to efficiently manage the cache, which is composed of the 3D-stacked DRAM distributed across the many NDP units. We can quickly see the similarity between this architec-

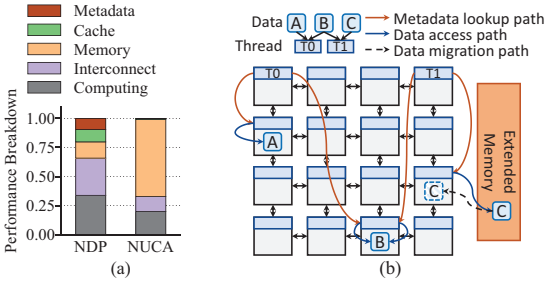

Fig. 2. (a) Comparison of the access latency breakdown in NDP and NUCA systems. (b) Challenges of a distributed DRAM cache in NDP systems.

ture and the conventional NUCA systems, which motivates us to design optimized data access and placement mechanisms.

Nevertheless, despite the same distributed cache organization with non-uniform access characteristics, NDP also exhibits unique challenges. To illustrate these differences, we simulate the PageRank workload on both systems using a simple static cacheline-interleaving policy. Both systems use 64 (processor or NDP) cores. The NUCA system mimics Jigsaw [6], with a 512 kB LLC bank per core, and a 9-cycle access latency plus a 3-cycle routing latency. The NDP system uses HMC-style memory, following Table II. Fig. 2(a) shows their average access latency breakdown. We observe two unique bottlenecks in NDP illustrated in Fig. 2(b).

(1) Interconnect overheads. In conventional static NUCA systems, all data elements are randomly cached across all units, resulting in high hop counts to gather all data to one core. By using the same random placement policy, the NDP system incurs a much higher latency portion on the interconnect (intra-stack and inter-stack) compared to the NUCA system, 32% vs. 13%. This overhead greatly affects the overall performance. NDP has higher interconnect latencies than NUCA because there are usually more NDP cores than NUCA cores and thus more interconnect hops, and we need to go off-chip more often due to the many 3D memory stacks. Actually, thanks to the much larger capacity, the NDP system achieves a significantly higher cache hit rate than the NUCA one (70% vs. 47%), which contributes to the much smaller portion of the next-level memory. However, most of these hits still exhibit long latencies, as they may be served from a remote NDP stack of multiple hops away.

(2) Metadata overheads. In the NDP system, the distributed DRAM cache is too large to have its metadata (i.e., tags) stored fully on-chip on the logic die, an issue similar to prior hardware-based DRAM caches [37], [50], [62], [67], [69], [72]. For instance, for a 256 MB DRAM cache, the metadata would need 16 MB space assuming each 64 B cacheline needs a 4-byte entry. Such a size cannot be entirely placed on-chip. As a result, there is significant extra access traffic (one metadata lookup per cacheline access, illustrated in Fig. 2(b)) consumed on accessing the cache metadata stored in the DRAM of local/remote NDP stacks. We observe that 10% of cycles are spent on remote tag accesses.

Consequently, design optimizations for the NDP cache should incorporate additional considerations beyond the existing NUCA schemes. Specifically, previous designs such as Jigsaw [6], CDCS [7], and Whirlpool [56] proposed to monitor the data use patterns of each core, and iteratively adjust the data placement to move data to the "center-of-mass" locations to reduce access distances. For Fig. 2(b), by placing the common data B in the middle of the T0 and T1 units, the interconnect hops can decrease from 4.5 to 1.5. If the cache capacity is insufficient, some data are gradually evicted to nearby places. These approaches have two issues. First, they treat the cache space at the center of the interconnect more valuable than the corner ones, because the preferred center-of-mass data locations are more likely to fall into the center NDP units. Second, if not all data can be put in the desired locations, some are placed at suboptimal positions, incurring extra hops. Both issues may not be severe in conventional NUCA because the interconnect impact is only a few cycles, but with NDP, data placement should be carefully handled.

One way to alleviate the long interconnect distance issue is to apply data replication, i.e., cache multiple data copies at different places so each core can get a nearby replica. Data replication is particularly suitable for our NDP scenario, where we could trade the abundant DRAM cache capacity for lower access distances. For example, in Fig. 2(b) if we replicate data B at both T0 and T1, we can eliminate remote accesses to it on the interconnect. Prior NUCA systems like R-NUCA [28] and Nexus [71] used a global replication degree that was applied uniformly to all the read-only data and instructions. This rather rigid scheme may not result in the best cache utilization, but the fine-grained cacheline-level management prohibits more complex designs. As mentioned above, the metadata lookup cost is already quite high in the NDP cache, not yet considering the needs of flexible data replication.

C. Our Approaches

NDPExt addresses the above issues using two co-designed techniques. On the hardware side (Section IV), NDPExt uses a *stream cache* organization on the NDP memory, with a coarse-grained stream abstraction instead of tracking fine-grained cachelines. The coarse granularity alleviates metadata cost and also supports more flexible data placement and replication at the stream level. On the software side (Section V), NDPExt uses a runtime on the host processor to periodically derive *optimized cache configuration schemes* that allocate the NDP cache space to the streams, based on efficient sampling techniques to profile the miss behaviors, and a novel algorithm to co-optimize capacity sizing, spatial placement, and data replication. Better allocation near data consumers and data replication both reduce the interconnect overheads and thus decrease the hit latency. However, replicating data may reduce the cache hit rate as well. Our algorithm carefully balances this hit rate vs. hit latency tradeoff.

IV. Stream Cache Design

NDPExt adopts software-defined streams as a coarse-grained abstraction for caching, in order to support flexible

data placement and replication, without excessive metadata overheads. Compared to conventional NUCA, tracking streams rather than individual cachelines reduces metadata storage and lookup cost. Furthermore, we can now use customized replication schemes for each stream rather than being limited to a single global policy as in R-NUCA [28] and Nexus [71].

A. Using Streams for NDP Caching

NDP is mainly used for memory-intensive applications, where there are large amounts of data to be accessed. These data are typically organized into certain dense or sparse structures and accessed through well-defined patterns. Streams, including the affine and indirect types, are thus a suitable coarse-grained abstraction to model such NDP data. For example, in parallelized graph traversal, the vertices are partitioned to multiple threads, and the edge list is sequentially accessed to obtain the outgoing edges for each vertex. Both the vertex list and the edge list are affine streams. In contrast, the order of the visited boolean array depends on the edge list, making it an indirect stream. As another example, our quantitative investigation finds that over 99% data accesses in PageRank in Fig. 2(a) can be captured by streams, with 55% as affine and 44% as indirect. These accesses are to various data structures like the vertex list, the edge list, and the rank score array. Many other NDP workloads also exhibit ample stream accesses [75].

Previous work has demonstrated that the stream model can effectively capture the access behaviors of various applications, and can be used to facilitate instruction prefetching [74], instruction offloading [75], [76], and automated parallelization [73]. We are the first to adopt the stream model to simplify DRAM cache organization and lookup. We adopt a similar model to prior work [74]–[76]. Streams are configured using the following API, which is made after each data allocation and before actual accesses to the data structure.

```
configure_stream(type, base, size, elemSize,
          [stride, length, order])
```

All streams should provide the first four arguments to configure the stream type (affine or indirect), the starting address, the total size, and the size of each element. Multi-dimensional affine streams can optionally supply the last three arguments, if the access order is different from the storage order, e.g., column-major accesses to a row-major matrix. We use similar strategies as in previous work [76] to support reordered affine iterators for at most 3 dimensions. The order is given in the 3-bit `order` argument. The hardware would cache the elements following their access order. Element reordering significantly improves data spatial locality, resulting in higher hit rates and reduced unnecessary DRAM row activations.

Table I explains the arguments in detail. These configurations are stored with a unique stream ID called `sid`, and will be sent to the hardware for stream management, as discussed below. Similar to previous work [73]–[76], currently we only support manually inserting stream configuration hints, and defer automatic compiler-based methods to future work. Note that the rest hardware components, such as the upper-level SRAM caches, still work with the conventional physical-address-based cachelines, and are not affected.

TABLE I
STREAM METADATA.

	Field	Bits	Description
Common	`sid`	9	Stream ID
	`base`	48	Base physical address
	`size`	48	Total stream size
	`elemSize`	4	Element size
	`readOnly`	1	Read-only or not
Affine only	`stride`	48×3	Access stride along dim X/Y/Z
	`length`	48×2	Stream length along dim Y/Z
	`order`	3	Access dimension order

B. Overall Caching Scheme

Similar to shared-cache-based D-NUCA, NDPExt maintains a remapping scheme to determine where a stream could be cached. Each stream could be remapped to multiple cache locations if it is read-only, supporting data replication to reduce the interconnect overheads when accessed. As we maintain the remapping metadata in the unit of coarse-grained streams rather than individual fine-grained cachelines, the metadata storage and lookup cost is well confined. Furthermore, the low-cost metadata scheme also enables us to use more complex replication policies, i.e., each stream is allowed to have its own replication scheme. This is the key difference from previous NUCA designs like Nexus [71].

We now describe the overall caching scheme in detail. Recall that the whole system is composed of multiple NDP units, each of which contains some local DRAM space that is used as part of the distributed DRAM cache capacity. Each stream is allocated a specific amount of cache space (in units of DRAM rows) from each NDP unit in the system. To further support data replication, the allocated cache space in these NDP units can be partitioned into multiple *replication groups*, and each group can independently cache a copy. For example, in Fig. 3(a), a specific stream A has cache space in all the four NDP units, organized into two replication groups of (0, 1) and (2, 3). The memory accesses missed from the local SRAM caches (❶) will first look up the local metadata structures (SLB and ATA, described later in Section IV-C), and then are either served by the local memory controller (❷a, ❸a) or routed to a remote NDP unit within its replication group (❷b, ❸b). If the data are not cached, we fetch from the extended memory.

As illustrated in Fig. 3(b), we use a structure called *remap shares* (`RShares`) to record the allocation of DRAM cache space to streams. `RShares` is a vector whose length is equal to the number of NDP units. Each item represents how many DRAM rows in the corresponding NDP unit are allocated to this stream as cache space. The exact locations are kept in another *remap row base* vector `RRowBase`. We use another structure called `RGroups` of the same length as `RShares`, to represent the replication group ID that each NDP unit belongs to. For example, with $\texttt{RShares} = (8,6,4,2)$ and $\texttt{RGroups} = (0,0,1,1)$, the first two units use 8 and 6 DRAM rows to

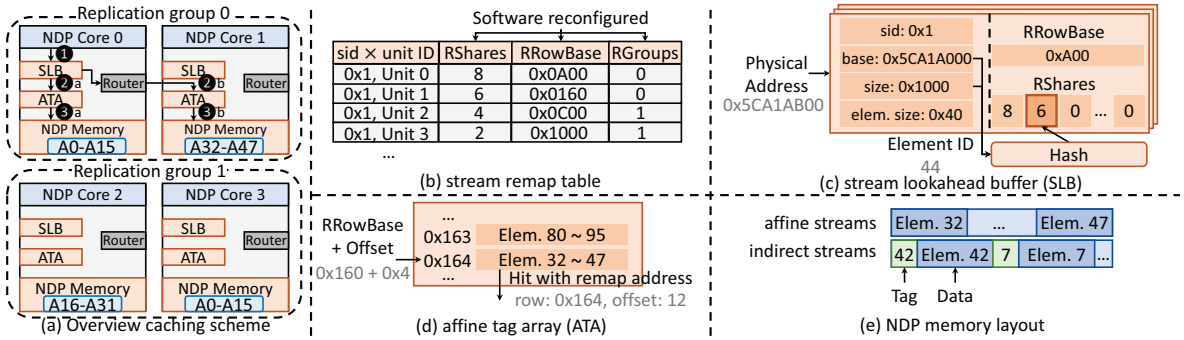

Fig. 3. (a) Overall caching scheme. SRAM caches and memory controllers are omitted. (b) Stream remap table for centralized remapping information. (c) Stream lookahead buffer (SLB) of unit 0. (d) Affine tag array (ATA) of unit 1. The valid and dirty bits are omitted. (e) Data layouts in DRAM caches for affine and indirect streams.

cache the stream as replication group 0, while the next two units contribute 4 and 2 DRAM rows to form replication group 1. Within each replication group, we use hash functions to determine the exact location for each element of the stream [6].

The above information fully determines the stream-based, distributed DRAM caching scheme in NDPExt. Such a scheme is configured by the runtime software (Section V) and enforced by the hardware (Section IV-C). To support dynamic changes in program behaviors, the scheme is also periodically reconfigured by the runtime. The runtime maintains a centralized *stream remap table* as Fig. 3(b) to keep the information, stored in the host processor memory together with the stream configurations (Table I). Currently, NDPExt supports up to 512 streams (9-bit `sid`) and 64 NDP units. Each item in `RShares`, `RRowBase`, and `RGroups` is 16 bits (at most 64k DRAM rows allocated in each NDP unit), 18 bits (256k DRAM rows in total per NDP unit), and 6 bits (maximum 64 groups). In total the stream remap table is $512 \times 64 \times 40\,\text{bits} = 160\,\text{kB}$.

Replication only for read-only streams. NDPExt only allows replication of read-only streams. For read-write streams, NDPExt allows only a single global replication group, so there is only a single copy of the stream data in the cache, ensuring coherence. To detect read-only streams, NDPExt keeps a `readOnly` bit for each stream, initialized to 1. In the event of a write operation to a read-only stream, an exception is sent to the host processor, which updates the stream remap table states [6], [7], [56], [71]. As the host processor has the information of all remap shares for all remap groups, it then sends invalidates to the relevant NDP units. No writebacks are needed because the data are still clean thus far, and this exception only happens at most once per stream. So the overhead is minor. Only stream status changes involve the host processor, while subsequent writes are handled by the NDP units. Such exceptions are rare and incur minor overheads in typical applications [28]. Alternatively, we can augment the stream API to let programmers mark read-only streams.

C. Hardware Structures

Besides the stream remap table as the global metadata, we further design a hardware structure called *stream lookahead buffer* (SLB) to locally cache the metadata in each NDP unit for faster probing, similar to TLB vs. page table. Fig. 3(c) shows the structure of SLB. Each SLB has 32 entries, one entry for one stream. Each entry is simplified and thus consumes less space compared to the full stream remap table entry. Specifically, we avoid storing `RGroups` by zeroing out all `RShares` items except for those in the same replication group, because only the cache space within the same group could be used by this NDP core. We also store only one remap row base item in `RRowBase` that corresponds to this NDP unit. Overall, each SLB only uses 4.6 kB of SRAM space, comparable to previous NDP hardware metadata cost [69].

Given a miss request, the physical address is searched in the local SLB using a modified content-addressable memory (CAM) structure, to identify the stream it falls in, by comparing with the `base` and `size` fields of each entry. It further calculates the corresponding element ID using the `elemSize`. In Fig. 3(c), `0x5CA1AB00` is within stream `0x1` and corresponds to an element ID of 44. To conduct range matching, we use a ternary CAM (TCAM) design [2]. We search for the common prefix of `base` and `base+size`, while the rest lower bits are set to "don't care". Then we check each of the hit ranges with digital comparators to identify the actual hit.

If the address misses in the SLB, we ask the host processor to check the complete stream remap table and refill the SLB entry. This method, akin to virtual memory translation, has been previously employed in NUCA systems [6], [7], [56], [71]. If the address is not identified as a stream element, we bypass the DRAM cache and directly access the extended memory. This case is rare (less than 0.1% in our experiments), because most of such data, e.g., local variables, are covered within the local SRAM caches. For simplicity, NDPExt currently supports associating one address with at most one stream. We ignore the rare cases where the same data participate in different access patterns in some applications. Otherwise, coherence is not guaranteed as these streams are separately cached and synonyms appear. On the other hand, dynamic-sized streams can be supported with oversubscription, e.g., over-allocating space, and updating stream configu-

rations and invalidating cached data for each memory reallocation. Untouched data are not cached, so oversubscription does not hurt performance. The rare stream configuration updates are done by the host processor.

With the obtained element ID in the stream, we further determine which NDP unit within this replication group should cache this element, using the `RShares` information in the SLB entry. For example in Fig. 3(c), the replication group has 8 and 6 DRAM rows in NDP units 0 and 1 as cache space. After hashing the element ID 44, we can determine this element is at NDP unit 1, with a certain DRAM row offset within its total 6 rows. Then the request is sent to the destination NDP unit through the interconnect. At the destination unit, we look up the SLB again to obtain the remap row base for this stream, and add the row offset to it to get the final DRAM row address.

With the DRAM row address, the last step is to determine the DRAM column address of the target element. NDPExt adopts different schemes for the two stream types.

Affine streams. Affine streams use a simple SRAM-based tag array (called the affine tag array, ATA, in Fig. 3(d)) to look up the exact DRAM column address of an element. We use a cache block size of 1 kB for affine streams, i.e., a 4-byte tag for each 1 kB data. A set-associative structure suffices for the ATA since affine streams are mostly regular. The block size should be large enough to amortize the tag overheads, but not exceed the available spatial locality. However, even with large 1 kB blocks, the required SRAM tag size is still too large to cover the entire 256 MB local DRAM. Fortunately, affine streams mostly exhibit scan-like patterns with mainly spatial locality, and do not need large cache space. Thus we restrict the total cache space per NDP unit for all affine streams to be within 16 MB, reducing the SRAM tags to a reasonable 64 kB on-chip storage. If there are too many streams or some streams are too long, they would miss in the DRAM cache and are accessed (mostly in a streaming manner) from the extended memory. Section VII-C assesses the performance impact.

Indirect streams. Data elements in indirect streams are cached individually without forming a larger block, considering that indirect streams exhibit little spatial locality, and sequentially prefetching additional data is useless. Because the caching granularity is a single element, much smaller than the block size in affine streams, indirect streams cannot use SRAM-based tag storage. To support flexible element sizes, NDPExt stores the tag with the data element, a strategy used in some DRAM cache designs [62]. The DRAM column address is also obtained through a hash function on the element ID. Then, a single DRAM access is performed to get both the tag and the data. Essentially, this is a direct-mapped cache where the location of an element is completely determined by hashing. Nevertheless, prior DRAM cache literature showed that high associativities only brought marginal hit rate benefits but introduced overheads for tags and replacement [62]. Some DRAM caches used small associativities with way prediction [15], [36]. We regard this as a potential alternative design. For simplicity we adopt the direct-mapped approach, but evaluate higher associativities in Section VII-C.

V. Cache Configuration Policies

As mentioned before, the software runtime on the host processor periodically (e.g., every 50 million cycles [6], called an *epoch*) generates the cache configuration, i.e., the stream remap table including `RShares`, `RRowBase`, and `RGroups`. Such a configuration essentially determines how to partition and allocate the DRAM cache space of all NDP units to the streams to achieve the highest overall efficiency. This section describes the configuration policies. First, we use hardware samplers to obtain the miss curve statistics of the data streams to guide our cache configuration (Section V-A), and develop an algorithm to assign the limited samplers to cover the many data streams in the workload (Section V-B). With the miss curve of each stream, we propose an effective approach to derive the optimized cache configuration (Section V-C), in which each stream can use its own best replication scheme, and the sizing and placement are co-optimized in one loop to avoid the inefficiencies in previous NUCA designs [6], [7], [56]. Finally, we discuss an optimization leveraging consistent hashing, to reduce data movements and cache misses during cache reconfiguration (Section V-D).

A. Set-Based Miss Curve Samplers

State-of-the-art NUCA partitioning policies [6], [7], [56] heavily rely on the miss curves (the miss rates at different cache capacities) of the workload to assign cache space to the data that can benefit the most from the extra space. Conventional LLCs usually have high associativities and thus can easily adopt way partitioning. However, as in Section IV, NDPExt uses DRAM caches with low associativities or even as direct-mapped, hence it can only be partitioned along sets. This poses challenges for miss curve profiling, as set partitioning does not have the stack property (if an access hits at a capacity C, it must also hit at capacities larger than C). As a result, previous utility monitor designs [63], where a single size-C monitor can capture the miss curve at all capacities from 1 to C, cannot be used anymore.

Nevertheless, because NDPExt uses hash functions to determine the set, we can assume all sets see relatively uniform accesses. We can thus sample a small number (i.e., $k = 32$) of sets to infer the overall miss behaviors of all sets. Specifically, for a stream that uses a total of K sets across multiple NDP units, if we sample k sets in one unit, we can scale the miss statistics by K/k for the total misses of this stream [6], [63].

Accordingly, we design our set-based hardware miss curve sampler as follows. Each sampler is used to derive the miss curve for one stream, by simultaneously capturing $c = 64$ different capacity cases, ranging from 32 kB to 256 MB (the full DRAM space per NDP unit). We geometrically partition [7] this range with a per-step multiplicative factor of $1.16 = \sqrt[63]{256\,\text{MB}/32\,\text{kB}}$. The complete curve can be interpolated as in [6]. Each capacity case needs $k = 32$ sets. We use simple static interleaving [63] to select the k sample sets among the total capacity in each case, and count the hits/misses to these sample sets. Without storing data, each set occupies 4 bytes for the address. In total, a sampler requires $32 \times 64 \times 4\,\text{B} = 8\,\text{kB}$

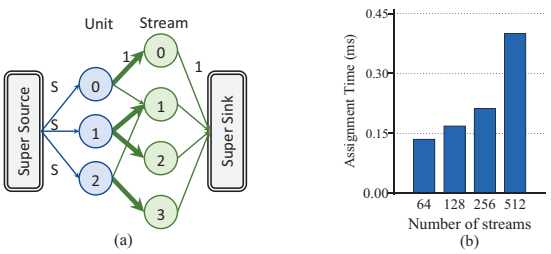

Fig. 4. (a) Modeling sampler assignment as a max-flow problem. The bold edges indicate the streams are selected by those units. (b) Host processor execution time to assign different numbers of streams.

storage. We put four samplers per unit, which are 32 kB, easily implemented with on-chip SRAM. Our sampler configuration is similar to prior work [6], [71] and could result in sufficient accuracy, which we further evaluate in Section VII-C.

B. Assigning Samplers to Streams

As described in Section V-A, each NDP unit has four miss curve samplers that each can monitor one stream. A constraint is that each sampler can only monitor a stream that is accessed by the local NDP unit. But each stream may be accessed by multiple units, so any sampler in these units can be used. Therefore, we need to collaboratively assign the samplers in different NDP units to the data streams in the workload, aiming to cover as many streams as possible.

First of all, we add a 512-length bitvector in each NDP unit, indicating which streams are accessed in this unit during this epoch. At the end of each epoch, the bitvectors of all units are sent to the host processor, as the input to determine the sampler assignment for the next epoch.

We then formalize the sampler assignment problem as a max-flow problem, which can be solved by the runtime software on the host processor, e.g., with the efficient Edmonds-Karp algorithm [19]. Specifically, as in Fig. 4(a), we construct a directed bipartite graph with NDP units and streams as the nodes. A unit-weight edge is added between an NDP unit and a stream if this unit has accessed this stream. We then connect the "super source" node to each unit, with an edge of weight $S = 4$, which is the number of samplers per unit, and thus the number of streams to be sampled on this unit. The "super sink" node is connected to all streams, with a unit-weight edge from each stream node. The max-flow algorithm identifies the maximum flow from the super source node to the super sink node while respecting to the capacity (weight) of each edge. The result would indicate that each stream node receives a unit flow from one of the unit nodes, and each unit node at most sends out S units of flow. This corresponds to the sampler assignment constraints. For example as shown by the bold edges in Fig. 4(a), if unit 0 samples stream 0, unit 1 samples streams 1 and 2, and unit 2 samples stream 3, all the edge capacity restrictions are satisfied while the total flow is maximized, because each stream node can flow to the super sink node, indicating every stream is sampled. This algorithm runs fast on the host processor, in less than half a millisecond to assign 512 streams as in Fig. 4(b).

When there are too many streams, it is possible that a fully covered solution cannot be obtained, meaning that some streams are not captured by any sampler. In these rare cases, we first sample a subset of the streams and buffer their results in the host processor, and in the next epoch sample the rest of the streams, until covering all streams. However, in our evaluated workloads we never encounter such a case.

C. Configuration Algorithm

With the miss curve information of all the streams sent to the host processor at the end of an epoch, the runtime will start its reconfiguration process to find the best scheme to allocate the DRAM cache space to the streams. Traditionally in NUCA, the sizing problem (i.e., how much capacity to allocate to each stream) and the placement problem (i.e., from which banks to obtain the allocated capacity) are solved separately [6], [7], [56]. Specifically, to determine sizing, the lookahead algorithm and its variants [6], [63] first identify the steepest slope at the current positions on all the miss curves, which implies the maximum utility margin. Then it allocates extra cache space to this data stream. However, in NDP systems, we know that the interconnect is a key bottleneck (Section III-B), calling for much more careful placement decisions. Furthermore, existing NUCA solutions have only limited support for data replication, requiring all data to apply the same replication degree.

To overcome these issues, we propose our configuration algorithm, with two key advantages. First, it determines the size of each stream cache and the corresponding placement across NDP units *simultaneously in an iterative manner*, so the interconnect overheads are thoroughly considered in the process. Second, it enables more flexible data replication, where each stream can use different replication schemes, corresponding to the replication groups (`RGroups`) in Section IV.

Algorithm 1 illustrates the algorithm details. The allocCap array stores the cache capacity allocated on each NDP unit to each stream. It is initialized to all zeros. At each iteration, the algorithm first finds the next steepest slope among the miss curves, and allocates cache space to this stream (Line 4), similar to the conventional lookahead algorithm [6], [63]. The space is allocated to all the units that have accessed this stream (i.e., `accUnits[sid]`, identified by the bitvector in Section V-B), assuming that each unit forms its own replication group at the beginning (Lines 6 to 8). When the available NDP memory is sufficient, this approach ensures maximum data replication and minimizes access distances.

As the allocation goes on, at some point we may fail to allocate the desired space on a unit. In this case, we consider either *extending the current replication group* to use space from nearby units, or *merging two existing groups* to reallocate space for the current unit. Note that these changes to the replication groups are for a specific stream, so different streams could evolve into different replication schemes. In a nutshell, each stream is initially replicated locally in each unit with the maximum replication degree. Later when local space is used up, we gradually use nearby unit space or reduce the replication degree to free up more space.

Algorithm 1: Cache configuration.

```
   input: number of streams S, number of NDP units N, stream miss
          curves missCurves[S], lists of units that have accessed each
          stream accUnits[S][].
   output: allocated capacity on each unit to each stream
           allocCap[N][S].
1  allocCap ← all 0;
2  do
3      found ← False;
4      sid, segSize ← NextSteepestSlopeSeg(missCurves);
5      for uid in accUnits[sid] do
6          if HasAvailSpace(segSize) then
7              allocCap[uid][sid] += segSize;
8              continue;
9          extendUnit ← NearestAvailableUnit(uid) ;
10         if extendUnit then
11             extendGroup ← (curRepGroup, extendUnit);
12             extendUtil ← CalcUtil(extendGroup) ;
13             found ← True;
14         groupA ← FindMergeGroup(uid) ;
15         groupB ← NearestGroup(groupA) ;
16         if groupA and groupB then
17             mergeGroup ← (groupA, groupB) ;
18             mergeUtil ← CalcUtil(mergeGroup) ;
19             found ← True;
20         if found then
21             allocCap ← AdjustAlloc(allocCap, extendUtil,
                 extendGroup, mergeUtil, mergeGroup);
22 while found;
```

To extend the current group (Lines 9 to 13), we start by searching for nearby units with sufficient space to accommodate the allocation. We use `CalcUtil` to select the grouping case with the highest utility. We apply an attenuation factor to the nearby unit to reflect the extra remote access cost when calculating the gained utility of this allocation. The attenuation factor is defined as the DRAM access latency divided by the sum of the DRAM latency and the interconnect latency, so farther units have smaller factors, decreasing their utility values. The utility of the extended group is the sum of the utilities of the units in it, weighted by their allocated space. For example, an existing replication group may contain 60 and 40 elements in units A and B, respectively. Its utility is thus $60+40\times k_{AB}=96$ for A and $40+60\times k_{BA}=94$ for B, in total 190. We assume all attenuation factors k are 0.9 here. To extend the next 20-element space to a nearby unit C, we calculate the utility of A as $60+40\times k_{AB}+20\times k_{AC}=114$. Similarly the utility of B is 112. The utility of the extended group is thus 226. Note that unit C does not access this stream so its utility contribution is 0.

To merge existing groups (Lines 14 to 19), we first find the group that contains the current unit, *and* has the lowest group utility (groupA). The low utility ensures that squeezing its space does not significantly affect performance. Then we try to merge this group with the nearest group of the same stream (groupB), to form a larger group. After merging, the algorithm frees up some elements of the stream in the current unit, and replaces them with those same ones in the units of the other merged group, which are remote. It then recalculates the group utility, where the attenuation factor is applied because some elements are now cached remotely.

Following the previous example, instead of extending to unit C, we merge the replication group (A, B) with another qualified replication group containing unit D with the same 100 elements. After merging, only one copy of the 100 elements are distributed to the three units in the new group, e.g., 30, 30, 40 for A, B, D, respectively. Hence the total utility for this stream decreases from 290 (190 for group A and B, plus 100 for D) to 280 (93 + 93 + 94 for A, B, D), but some space has been freed up. Note that the stream selected to be merged may not necessarily be the same as the stream currently under allocation. As long as a replication group could free up some space from merging, the allocation can continue.

Finally, the algorithm compares the gained utilities of group extending and group merging, and selects the better one (Line 21). By employing this iterative algorithm, NDPExt is able to simultaneously allocate and place cache space for each stream, with similar cost as in [6].

Similar to prior conventional NUCA placement schemes [7], our configuration process cannot always guarantee the global optimal placement. But it still works well in practice. For example, while sampling is based on previous epochs, using history to predict future is a widely used technique in computer architecture [33], [63]. Similarly, while our placement strategies are greedy, previous work has empirically demonstrated lookahead algorithms [63] work well. We leave complicated policies for future work. Furthermore, our configuration algorithm is able to result in proper degrees of replication for read-only streams to balance between high cache hit rates (less replication) and low remote access latencies (more replication). We find that for applications with most read-only streams, a modest amount of cache space is used by replicated data, e.g., up to 33% and 27% for `mv` and `gnn`, respectively. For `backprop` that contains two phases, the read-intensive `layerforward` kernel uses 91% cache space for replication, while no replication occurs in the `adjustweights` kernel which needs to write data.

D. Reducing Data Movements in Reconfiguration

After each periodical reconfiguration, the cache configuration is sent to the NDP stacks and applied. Prior work [6], [7] employed bulk invalidation to invalidate all cached data if the space was reassigned. Because the reconfiguration epoch is sufficiently long, e.g., 50 million cycles, even though the bulk invalidation is slow and takes up to 300k cycles [6], it does not significantly impact the overall performance.

Nevertheless, to reduce data movements and cache misses, we discuss an optimization in NDPExt that adopts consistent hashing [40] to keep as many cached data unchanged as possible. For each stream, we consider each possible DRAM row location (`RRowBase`) in each NDP unit as an individual spot in the circular space of consistent hashing, in total 65536×64 spots. During reconfiguration, NDPExt remaps the stream data to the nearest spot leveraging consistent hashing, therefore saving data movements. In our experiments, consistent hashing

TABLE II
SYSTEM CONFIGURATIONS.

NDP system	4×2 inter-stack mesh, 16 NDP cores per stack; 128 NDP cores in total
NDP core L1I L1D	2 GHz, in-order 2-way, 32 kB per core, 64 B cachelines, LRU 4-way, 64 kB per core, 64 B cachelines, LRU
NDP HBM	16 GB HBM 3.0, 1600 MHz, 256 MB/unit; RCD-CAS-RP: 24-24-24; RD/WR: 1.7 pJ/bit, ACT/PRE: 0.6 nJ
NDP HMC	16 GB HMC 2.1, 1250 MHz, 256 MB/unit; RCD-CAS-RP: 14-14-14;
Extended memory	DDR5-4800, 4 channels $\times$ 2 ranks $\times$ 16 banks; RCD-CAS-RP: 40-40-40; RD/WR: 3.2 pJ/bit, ACT/PRE: 3.3 nJ
Intra-stack network Inter-stack network CXL link	128-bit link, 1.5 ns/hop [65], [69]; 0.4 pJ/bit 32 GB/s per dir., 10 ns/hop [20], [22], [69]; 4 pJ/bit 16-lane; 200 ns link latency; 11.4 pJ/bit

reduces the invalidation traffic by 9.4% on average, and brings a 3.7% speedup compared to bulk invalidation.

VI. METHODOLOGY

System models. We use zsim [64] to conduct simulations of a 128-core NDP system, summarized in Table II. We extend zsim to support multi-stack mesh interconnection. Our NDP logic consists of simple in-order cores of 2 GHz. To show generality, we model two types of NDP memory, following HBM3 and HMC2, respectively, with 16 GB total capacity. For HBM, each stack is connected via a 2.5D interposer to a logic die, on which there are 16 cores accessing this HBM through a crossbar [22], [69]. For HMC, 16 cores are put at the logic die at the base of each stack, and each core directly accesses its corresponding vault [1], [22]. Effectively, with HBM, each full stack is a NUCA node, while HMC enables more fine-grained NUCA nodes of individual vaults. We show overall performance comparison with both memory types, but do performance analysis primarily using HBM since the two show similar trends. The extended memory is modeled as four DDR5-4800 channels. The memory timing and energy parameters are from datasheets and follow prior work [20], [21], [29], [34], [35], [48], [54]. The intra-stack NoC and inter-stack link models are obtained from [65]. We use a direct-attached multi-headed CXL Type-3 device, with a default link latency of 200 ns [68] (excluding DRAM access). We evaluate other CXL latency cases in Fig. 8(b).

Total SRAM cost. The timing and energy of the SRAM structures in NDPExt are modeled using CACTI 7 [3]. The added SRAM per NDP unit includes the following components: 1) The 32-entry SLB, consuming 4544 bytes. 2) The affine tag array with 16k entries, resulting in 64 kB. 3) The hardware miss curve samplers, using another 32 kB. 4) The bitvector that records which streams are accessed, which is 512 bits. Overall, the SRAM usage is well aligned with prior NDP designs [69] and remains acceptable.

Workloads. We use a diverse set of NDP-friendly applications, including tensor computation, graph computing, and other parallel workloads. Tensor workloads include DLRM-style recommendation system inference (`recsys`), matrix-vector multiplication (`mv`), graph convolution neural network (`gnn`) using the Reddit dataset [77] implemented with sparse-dense matrix multiplications. These workloads are implemented using SIMD instructions. We also port workloads from Rodinia [10], including `backprop`, `hotspot`, `lavaMD`, `lud`, and `pathfinder`. We exclude workloads with small memory footprints. Graph workloads are from GAP [4], including breadth-first search (`bfs`), pagerank (`pr`), connected components (`cc`), betweenness centrality (`bc`), and triangle counting (`tc`). We simulate their 2nd to 5th iterations. Multiple processes of the workload are executed until the total footprint exceeds the NDP memory. The number of streams ranges from 4 to 256 in different workloads. We only need to modify a few lines of code in each workload (with an average of 4.3 lines) to annotate these streams, with minor extra programming efforts.

Baseline designs. We compare NDPExt with Jigsaw [6], Whirlpool [56], and Nexus [71], where these conventional NUCA designs are adapted to our DRAM cache. These baselines need to issue a metadata access before the actual data access, and we use a 128 kB metadata cache per NDP unit for fair comparison. To further improve the baseline performance, we optimistically assume an idealized dual-granularity metadata cache design similar to Bi-Modal DRAM Cache [27], where metadata are stored per 512 B block but data migration is fine-grained at 64 B to avoid overfetch. Whirlpool partitions the cache for different statically classified data. We annotate streams as in NDPExt and manually classify these streams. Nexus replicates memory pages following regular-shaped groups using a global replication degree. In addition, we further compare with a non-NDP host processor with 64 cores and DDR5 main memory. It uses the Jigsaw-style NUCA design for its 32 MB LLC. Besides the above baselines, we also evaluate a version of NDPExt without the runtime reconfiguration (NDPExt-static), but the cache space is equally allocated to every stream.

VII. EVALUATION

We first compare the overall performance and energy between NDPExt and the baselines. Then we analyze the performance gains under different configurations, and present detailed sensitivity studies of various design parameters.

A. Overall Comparison

Fig. 5(a) and (b) present the performance comparisons for the HBM-style and HMC-style NDP systems, respectively. In both cases, NDP systems exhibit considerable performance advantages over non-NDP host execution, with gains ranging from 4.3$\times$ to 7.3$\times$. Among the NDP-based ones, NDPExt consistently achieves the best performance, outperforming the second-best one, Nexus, by 1.41$\times$ with HBM and 1.48$\times$ with HMC on average. The `recsys` workload shows notable benefits, with up to 2.43$\times$ and 2.17$\times$ improvements under HBM

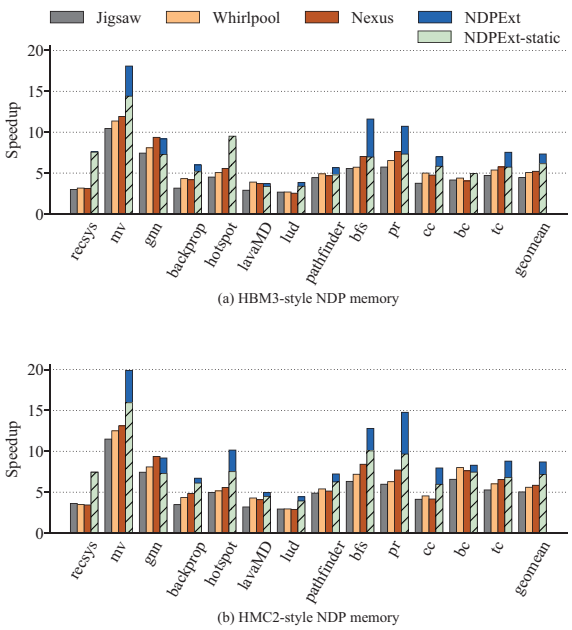

Fig. 5. Overall performance comparison between NDPExt and the baselines using (a) HBM-style and (b) HMC-style NDP. Normalized to non-NDP host.

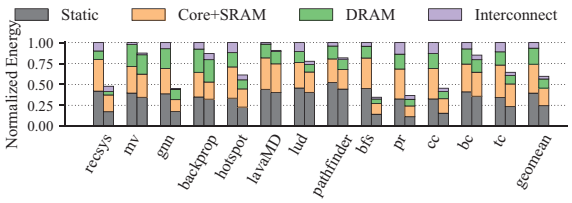

Fig. 6. Overall energy comparison between NDPExt (right) and Nexus (left) using HBM-style NDP. Normalized to Nexus.

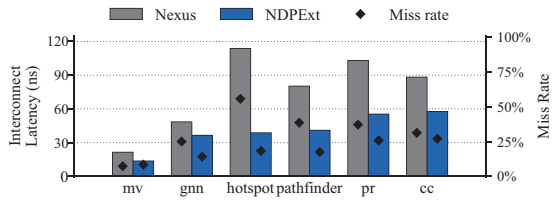

Fig. 7. Interconnect latency (bars) and miss rate (dots) comparisons between NDPExt and Nexus.

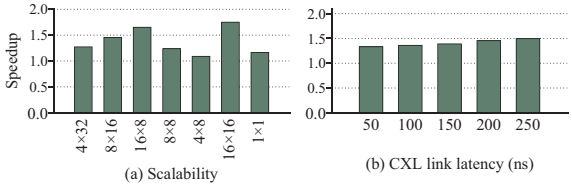

Fig. 8. Speedups of NDPExt over Nexus with (a) different NDP core counts, as # stacks × # NDP cores per stack, and (b) different CXL link latencies.

and HMC. Although the HBM-style system has fewer NUCA nodes than the HMC-style one, they exhibit similar speedups. This is because the performance is mainly bound by the inter-stack links that have much lower bandwidth and higher latency than the intra-stack NoC. Finally, NDPExt surpasses NDPExt-static by 1.2× on average, and shows substantial speedups up to 1.7× on irregular workloads like `pr`, which would require more dynamic cache configuration decisions.

The performance gains of NDPExt can be attributed to two reasons: metadata access elimination and better stream-level placement. For metadata, although the 128 kB dual-granularity metadata cache in the baselines achieves over 95% hit rates for high-locality workloads, with large-scale graph workloads the hit rates drastically decrease to 47%. Since each metadata miss requires at least a local memory access on the critical path, the performance suffers. In contrast, NDPExt uses coarse-grained stream-level metadata that are much smaller and can stay local, alleviating the metadata access overheads.

For data placement, Fig. 7 further presents both the average interconnect latencies and the miss rates in Nexus and NDPExt, for a selection of representative workloads. The interconnect latency reflects the interconnect overheads, while the miss rate quantifies the number of requests serviced by the extended memory. NDPExt significantly reduces the interconnection overheads with better data placement and proper data replication. For example, in `hotspot`, Nexus suffers from a long average interconnect latency of 113 ns, while NDPExt uses several small replication groups each containing 1 or 2 units, besides a large group of 10 units. The interconnect latency is thus reduced to 38 ns. For the miss rate, using the stream abstraction effectively enables prefetching that exploits spatial locality, e.g., `hotspot` and `pathfinder`. Although for some workloads like `mv` the miss rate may slightly increase due to replication, overall NDPExt exhibits better performance than the baseline. The results indicate NDPExt can properly size streams and form good replication groups for each stream.

Fig. 6 exhibits the energy consumption breakdown for all workloads. On average, NDPExt saves a significant portion of 40.3% energy consumption compared to Nexus. The static energy follows the execution time. Since NDPExt eliminates additional tag accessing, and the miss rate to the higher-energy extended memory decreases (Fig. 7), the DRAM energy is reduced by 8.3%. Thanks to better placement schemes, the interconnect energy is reduced from 6.6% to 3.2%. This energy reduction confirms the previous reduction in access latency.

B. Performance Analysis

From now on, we focus on the HBM-style system, and only present the average performance results due to space limit.

NDP core count scalability. In Fig. 8(a), we first change the stack counts and keep the same total core count (first three bars). With more distant cores across more stacks, the interconnection cost reduction in NDPExt is more critical, leading to higher speedups of up to 1.65× for 16 stacks. We next scale down NDP cores from 128 cores to 32 cores by using fewer stacks (4th and 5th bars). NDPExt still achieves

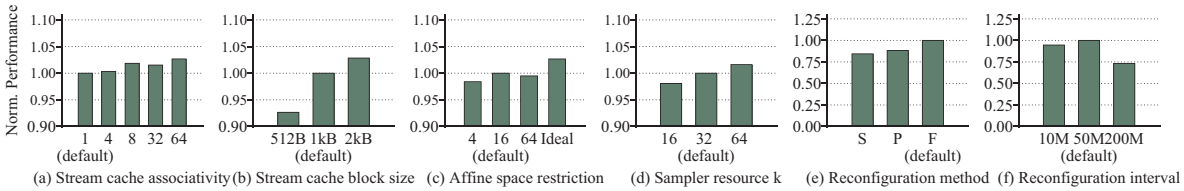

Fig. 9. Impact of various design decisions in NDPExt. The results in each case are normalized to the default value as marked.

9% higher performance for a small 4-stack system. We then test a large 16-stack, 256-core system. More cores stress the interconnect with more accesses, and thus NDPExt achieves a higher speedup to 1.75×. Finally, we show the speedup of only one NDP unit. The NDP system falls back to a conventional DRAM cache. In this case, cache configuration in Section V is no longer needed, and we eliminate its cost in the results, similar to the static mode in Fig. 5. NDPExt still offers a speedup of 1.16× from the more efficient stream abstraction.

CXL link latency impact. In Fig. 8(b), we evaluate different CXL link latencies. We use a more practical 200 ns CXL link latency, but also consider optimistic 50 to 70 ns projections in earlier reports [49], [51], [66], [80]. Higher link latencies make misses to the extended memory more expensive. In this case, the placement scheme in NDPExt has higher benefits compared to the center-of-mass method used in the baselines. Since NDPExt effectively reduces such misses as in Fig. 7, it obtains higher speedups with slower CXL links, from 1.33× to 1.50×.

Reconfiguration overheads. The overheads consist of two parts. First, the host processor assigns samplers to streams (Section V-B), whose cost is evaluated in Fig. 4(b). Second, data migration will happen during reconfiguration. We find that data migration requests only account for 1.3% of all access requests, thanks to the optimization in Section V-D.

C. Design Choice Studies

In this subsection we study how to choose the proper values for various design parameters in NDPExt.

Stream cache associativity and block size. Recall that NDPExt directly hashes elements in indirect streams, so the associativity is fixed to 1 way. In this idealized experiment, we only use higher associativities to show the potential performance loss of using such a direct-mapped cache. Fig. 9(a) shows that the direct-mapped cache is acceptable. Even a 64-way organization only brings minor gains. This aligns well with prior DRAM cache work [62]. Graph computing workloads are those that can benefit the most from higher associativities, with about 10% to 20% speedups at 64 ways. Fig. 9(b) further shows the impact of using different block sizes for affine streams in our stream cache. Using blocks larger than 1 kB brings slightly better performance, especially for applications with good spatial locality. We leave reconfigurable block sizes for different applications as future work.

Affine space restriction. To fit affine metadata on-chip, NDPExt restricts the total allocated space for affine streams (Section IV-C). Fig. 9(c) shows the results of different affine space restrictions. We find that for most applications this 16 MB restriction has a negligible impact on performance. We also evaluate an ideal case without any restrictions. It shows about 2% average speedup, mostly on `mv` and `gnn` that have the most affine streams.

Sampling resources. Fig. 9(d) evaluates different numbers of sampling sets k. Using different k values does not have a noticeable impact on the overall performance, because the behavior of the cache can be approximated using only a few sampling sets [63], [71].

Reconfiguration method and interval. Fig. 9(e) evaluates different reconfiguration methods. "S(tatic)" equally allocates the cache space for each stream on each unit, without dynamic reconfiguration. "P(artial)" reconfigures only during the first 200M cycles, and assumes the workload behaviors are stabilized afterwards. "F(ull)" is adopted in NDPExt, periodically reconfiguring as the workloads execute. We find that partial reconfiguration cannot unleash all performance potentials, especially for applications with many streams like `mv`, and applications with dynamic tasks like `pr`, respectively 14.7% and 20.7% slower than NDPExt. Fig. 9(f) investigates the reconfiguration interval. We find that using 50M cycles is sufficient. Using a 100M cycle interval exhibits a considerable 26% performance drop due to application dynamism.

VIII. Conclusions

In this paper, we make a case for a new architecture paradigm, called near-data processing (NDP) with extended memory, in order to address the capacity limitation of existing NDP systems. We further propose novel hardware-software co-designed optimizations to address the performance bottlenecks in this architecture. A stream cache design is used to alleviate the metadata cost of the distributed DRAM cache in this system. An effective configuration algorithm can derive the optimized schemes about allocating cache space to different data. Our design significantly improves performance on large-scale memory-intensive workloads.

Acknowledgment

The authors thank the anonymous reviewers for their valuable suggestions, and the Tsinghua IDEAL group members for constructive discussion. This work was supported by the National Natural Science Foundation of China (62072262). Mingyu Gao is the corresponding author.

REFERENCES

[1] J. Ahn, S. Hong, S. Yoo, O. Mutlu, and K. Choi, "A Scalable Processing-in-Memory Accelerator for Parallel Graph Processing," in *42nd International Symposium on Computer Architecture (ISCA)*, 2015, pp. 105–117.

[2] M. J. Akhbarizadeh, M. Nourani, D. S. Vijayasarathi, and P. T. Balsara, "PCAM: A Ternary CAM Optimized for Longest Prefix Matching Tasks," in *22nd IEEE International Conference on Computer Design (ICCD)*, 2004, pp. 6–11.

[3] R. Balasubramonian, A. B. Kahng, N. Muralimanohar, A. Shafiee, and V. Srinivas, "CACTI 7: New Tools for Interconnect Exploration in Innovative Off-Chip Memories," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 14, no. 2, pp. 14:1–14:25, 2017.

[4] S. Beamer, K. Asanović, and D. Patterson, "The GAP Benchmark Suite," *arXiv preprint arXiv:1508.03619*, Aug. 2015.

[5] B. M. Beckmann, M. R. Marty, and D. A. Wood, "ASR: Adaptive Selective Replication for CMP Caches," in *39th International Symposium on Microarchitecture (MICRO)*, 2006, pp. 443–454.

[6] N. Beckmann and D. Sanchez, "Jigsaw: Scalable Software-Defined Caches," in *22nd International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2013, pp. 213–224.

[7] N. Beckmann, P. Tsai, and D. Sanchez, "Scaling Distributed Cache Hierarchies Through Computation and Data Co-scheduling," in *21st International Symposium on High Performance Computer Architecture (HPCA)*, 2015, pp. 538–550.

[8] A. Boroumand, S. Ghose, Y. Kim, R. Ausavarungnirun, E. Shiu, R. Thakur, D. Kim, A. Kuusela, A. Knies, P. Ranganathan, and O. Mutlu, "Google Workloads for Consumer Devices: Mitigating Data Movement Bottlenecks," in *23rd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2018, pp. 316–331.

[9] P. Caheny, L. Alvarez, M. Casas, and M. Moretó, "TD-NUCA: Runtime Driven Management of NUCA Caches in Task Dataflow Programming Models," in *2017 International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2022, pp. 80:1–80:15.

[10] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S. Lee, and K. Skadron, "Rodinia: A Benchmark Suite for Heterogeneous Computing," in *International Symposium on Workload Characterization (IISWC)*, 2009, pp. 44–54.

[11] D. Chen, H. He, H. Jin, L. Zheng, Y. Huang, X. Shen, and X. Liao, "MetaNMP: Leveraging Cartesian-Like Product to Accelerate HGNNs with Near-Memory Processing," in *50th International Symposium on Computer Architecture (ISCA)*, 2023, pp. 56:1–56:13.

[12] A. Cho, A. Saxena, M. Qureshi, and A. Daglis, "A Case for CXL-Centric Server Processors," *arXiv preprint arXiv:2305.05033*, May 2023.

[13] S. Cho and L. Jin, "Managing Distributed, Shared L2 Caches through OS-Level Page Allocation," in *39th International Symposium on Microarchitecture (MICRO)*, 2006, pp. 455–468.

[14] J. Choquette, "Nvidia Hopper GPU: Scaling Performance," in *2022 IEEE Hot Chips 34 Symposium (HCS)*, 2022, pp. 1–46.

[15] 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 *47th International Symposium on Microarchitecture (MICRO)*, 2014, pp. 1–12.

[16] CXL Consortium, "Compute Express Link 3.0 White Paper," 2022, https://www.computeexpresslink.org/_files/ugd/0c1418_1798ce97c1e6438fba818d760905e43a.pdf.

[17] G. Dai, T. Huang, Y. Chi, J. Zhao, G. Sun, Y. Liu, Y. Wang, Y. Xie, and H. Yang, "GraphH: A Processing-in-Memory Architecture for Large-Scale Graph Processing," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, vol. 38, no. 4, pp. 640–653, 2019.

[18] G. Dai, Z. Zhu, T. Fu, C. Wei, B. Wang, X. Li, Y. Xie, H. Yang, and Y. Wang, "DIMMining: Pruning-Efficient and Parallel Graph Mining on Near-Memory-Computing," in *49th International Symposium on Computer Architecture (ISCA)*, 2022, pp. 130–145.

[19] J. Edmonds and R. M. Karp, "Theoretical Improvements in Algorithmic Efficiency for Network Flow Problems," *Journal of the ACM (JACM)*, vol. 19, no. 2, pp. 248–264, 1972.

[20] M. Gao, G. Ayers, and C. Kozyrakis, "Practical Near-Data Processing for In-Memory Analytics Frameworks," in *24th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2015, pp. 113–124.

[21] M. Gao, J. Pu, X. Yang, M. Horowitz, and C. Kozyrakis, "TETRIS: Scalable and Efficient Neural Network Acceleration with 3D Memory," in *23rd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2017, pp. 751–764.

[22] C. Giannoula, N. Vijaykumar, N. Papadopoulou, V. Karakostas, I. Fernandez, J. Gómez-Luna, L. Orosa, N. Koziris, G. I. Goumas, and O. Mutlu, "SynCron: Efficient Synchronization Support for Near-Data-Processing Architectures," in *27th International Symposium on High Performance Computer Architecture (HPCA)*, 2021, pp. 263–276.

[23] J. Gómez-Luna, I. E. Hajj, I. Fernandez, C. Giannoula, G. F. Oliveira, and O. Mutlu, "Benchmarking a New Paradigm: An Experimental Analysis of a Real Processing-in-Memory Architecture," *arXiv preprint arXiv:2105.03814*, May 2021.

[24] D. Gouk, M. Kwon, H. Bae, S. Lee, and M. Jung, "Memory Pooling With CXL," *IEEE Micro*, vol. 43, no. 2, pp. 48–57, 2023.

[25] D. Gouk, S. Lee, M. Kwon, and M. Jung, "Direct Access, High-Performance Memory Disaggregation with DirectCXL," in *2022 USENIX Annual Technical Conference (ATC)*, 2022, pp. 287–294.

[26] P. Gu, X. Xie, Y. Ding, G. Chen, W. Zhang, D. Niu, and Y. Xie, "iPIM: Programmable In-Memory Image Processing Accelerator Using Near-Bank Architecture," in *47th International Symposium on Computer Architecture (ISCA)*, 2020, pp. 804–817.

[27] N. D. Gulur, M. Mehendale, R. Manikantan, and R. Govindarajan, "Bi-Modal DRAM Cache: Improving Hit Rate, Hit Latency and Bandwidth," in *47th International Symposium on Microarchitecture (MICRO)*, 2014, pp. 38–50.

[28] N. Hardavellas, M. Ferdman, B. Falsafi, and A. Ailamaki, "Reactive NUCA: Near-Optimal Block Placement and Replication in Distributed Caches," in *36th International Symposium on Computer Architecture (ISCA)*, 2009, pp. 184–195.

[29] M. He, C. Song, I. Kim, C. Jeong, S. Kim, I. Park, M. Thottethodi, and T. N. Vijaykumar, "Newton: A DRAM-maker's Accelerator-in-Memory (AiM) Architecture for Machine Learning," in *53rd International Symposium on Microarchitecture (MICRO)*, 2020, pp. 372–385.

[30] L. Huang, Z. Zhang, S. Li, D. Niu, Y. Guan, H. Zheng, and Y. Xie, "Practical Near-Data-Processing Architecture for Large-Scale Distributed Graph Neural Network," *IEEE Access*, vol. 10, pp. 46 796–46 807, 2022.

[31] W. Huangfu, K. T. Malladi, A. Chang, and Y. Xie, "BEACON: Scalable Near-Data-Processing Accelerators for Genome Analysis near Memory Pool with the CXL Support," in *55th International Symposium on Microarchitecture (MICRO)*, 2022, pp. 727–743.

[32] Intel, "4th Gen Intel Xeon Processor Scalable Family, Sapphire Rapids," 2023, https://www.intel.com/content/www/us/en/developer/articles/technical/fourth-generation-xeon-scalable-family-overview.html.

[33] A. Jain and C. Lin, "Hawkeye: Leveraging Belady's Algorithm for Improved Cache Replacement," in *2nd Cache Replacement Competition*, 2017, pp. 1–4.

[34] JEDEC, "DDR5 SDRAM Standard," 2022, https://www.jedec.org/standards-documents/docs/jesd-79-5b.

[35] JEDEC, "High Bandwidth Memory (HBM3) DRAM," 2023, https://www.jedec.org/standards-documents/docs/jesd238a.

[36] D. Jevdjic, G. H. Loh, C. Kaynak, and B. Falsafi, "Unison Cache: A Scalable and Effective Die-Stacked DRAM Cache," in *47th International Symposium on Microarchitecture (MICRO)*, 2014, pp. 25–37.

[37] D. Jevdjic, S. Volos, and B. Falsafi, "Die-Stacked DRAM Caches for Servers: Hit Ratio, Latency, or Bandwidth? Have It All with Footprint Cache," in *40th International Symposium on Computer Architecture (ISCA)*, 2013, pp. 404–415.

[38] H. Kang, Y. Zhao, G. E. Blelloch, L. Dhulipala, Y. Gu, C. McGuffey, and P. B. Gibbons, "PIM-tree: A Skew-resistant Index for Processing-in-Memory," *Proceedings of the VLDB Endowment*, vol. 16, no. 4, pp. 946–958, 2022.

[39] A. Kannan, N. D. E. Jerger, and G. H. Loh, "Enabling Interposer-Based Disintegration of Multi-Core Processors," in *48th International Symposium on Microarchitecture (MICRO)*, 2015, pp. 546–558.

[40] D. R. Karger, E. Lehman, F. T. Leighton, R. Panigrahy, M. S. Levine, and D. Lewin, "Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web," in *29th Annual ACM Symposium on the Theory of Computing (ToC)*, 1997, pp. 654–663.

[41] L. Ke, U. Gupta, B. Y. Cho, D. Brooks, V. Chandra, U. Diril, A. Firoozshahian, K. M. Hazelwood, B. Jia, H. S. Lee, M. Li, B. Maher, D. Mudigere, M. Naumov, M. Schatz, M. Smelyanskiy, X. Wang, B. Reagen, C. Wu, M. Hempstead, and X. Zhang, "RecNMP: Acceler-

ating Personalized Recommendation with Near-Memory Processing," in *47th International Symposium on Computer Architecture (ISCA)*, 2020, pp. 790–803.

[42] C. Kim, D. Burger, and S. W. Keckler, "An Adaptive, Non-Uniform Cache Structure for Wire-Delay Dominated On-Chip Caches," in *10th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2002, pp. 211–222.

[43] D. Kim, J. Kung, S. M. Chai, S. Yalamanchili, and S. Mukhopadhyay, "Neurocube: A Programmable Digital Neuromorphic Architecture with High-Density 3D Memory," in *43rd International Symposium on Computer Architecture (ISCA)*, 2016, pp. 380–392.

[44] G. Kim, J. Kim, J. H. Ahn, and J. Kim, "Memory-Centric System Interconnect Design with Hybrid Memory Cubes," in *22nd International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2013, pp. 145–155.

[45] J. S. Kim, D. S. Cali, H. Xin, D. Lee, S. Ghose, M. Alser, H. Hassan, O. Ergin, C. Alkan, and O. Mutlu, "GRIM-Filter: Fast Seed Location Filtering in DNA Read Mapping Using Processing-in-Memory Technologies," *BMC Genomics*, vol. 19, no. S2, 2018.

[46] Y. Kwon, Y. Lee, and M. Rhu, "TensorDIMM: A Practical Near-Memory Processing Architecture for Embeddings and Tensor Operations in Deep Learning," in *52nd International Symposium on Microarchitecture (MICRO)*, 2019, pp. 740–753.

[47] H. Lee, S. Cho, and B. R. Childers, "CloudCache: Expanding and Shrinking Private Caches," in *17th International Symposium on High Performance Computer Architecture (HPCA)*, 2011, pp. 219–230.

[48] S. H. Lee, S. Kang, J. Lee, H. Kim, E. Lee, S. Seo, H. Yoon, S. Lee, K. Lim, H. Shin, J. Kim, S. O, A. Iyer, D. Wang, K. Sohn, and N. S. Kim, "Hardware Architecture and Software Stack for PIM Based on Commercial DRAM Technology : Industrial Product," in *48th International Symposium on Computer Architecture (ISCA)*, 2021, pp. 43–56.

[49] H. Li, D. S. Berger, L. Hsu, D. Ernst, P. Zardoshti, S. Novakovic, M. Shah, S. Rajadnya, S. Lee, I. Agarwal, M. D. Hill, M. Fontoura, and R. Bianchini, "Pond: CXL-Based Memory Pooling Systems for Cloud Platforms," in *28th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2023, pp. 574–587.

[50] Y. Li and M. Gao, "Baryon: Efficient Hybrid Memory Management with Compression and Sub-Blocking," in *29th International Symposium on High Performance Computer Architecture (HPCA)*, 2023, pp. 137–151.

[51] H. A. Maruf, H. Wang, A. Dhanotia, J. Weiner, N. Agarwal, P. Bhattacharya, C. Petersen, M. Chowdhury, S. O. Kanaujia, and P. Chauhan, "TPP: Transparent Page Placement for CXL-Enabled Tiered-Memory," in *28th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2023, pp. 742–755.

[52] S. A. McKee, "Reflections on the Memory Wall," in *1st Conference on Computing Frontiers (CF)*, 2004, p. 162.

[53] 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 *21st International Symposium on High Performance Computer Architecture (HPCA)*, 2015, pp. 126–136.

[54] Micron, "Hybrid Memory Cube – HMC Gen2," 2018, https://www.micron.com/-/media/client/global/documents/products/data-sheet/hmc/gen2/hmc_gen2.pdf.

[55] H. A. Moghaddam, Y. H. Son, J. H. Ahn, and N. S. Kim, "Chameleon: Versatile and Practical Near-DRAM Acceleration Architecture for Large Memory Systems," in *49th International Symposium on Microarchitecture (MICRO)*, 2016, pp. 50:1–50:13.

[56] A. Mukkara, N. Beckmann, and D. Sanchez, "Whirlpool: Improving Dynamic Cache Management with Static Data Classification," in *21st International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2016, pp. 113–127.

[57] T. Nowatzki, V. Gangadhar, N. Ardalani, and K. Sankaralingam, "Stream-Dataflow Acceleration," in *44th International Symposium on Computer Architecture (ISCA)*, 2017, pp. 416–429.

[58] M. Ogleari, Y. Yu, C. Qian, E. L. Miller, and J. Zhao, "String Figure: A Scalable and Elastic Memory Network Architecture," in *25th International Symposium on High Performance Computer Architecture (HPCA)*, 2019, pp. 647–660.

[59] S. J. Park, H. Kim, K. Kim, J. So, J. Ahn, W. Lee, D. Kim, Y. Kim, J. Seok, J. Lee, H. Ryu, C. Y. Lee, J. Prout, K. Ryoo, S. Han, M. Kook, J. S. Choi, J. Gim, Y. S. Ki, S. Ryu, C. Park, D. Lee, J. Cho, H. Song, and J. Lee, "Scaling of Memory Performance and Capacity with CXL Memory Expander," in *2022 IEEE Hot Chips 34 Symposium (HCS)*, 2022, pp. 1–27.

[60] M. Poremba, I. Akgun, J. Yin, O. Kayiran, Y. Xie, and G. H. Loh, "There and Back Again: Optimizing the Interconnect in Networks of Memory Cubes," in *44th International Symposium on Computer Architecture (ISCA)*, 2017, pp. 678–690.

[61] M. K. Qureshi, "Adaptive Spill-Receive for Robust High-Performance Caching in CMPs," in *15th International Symposium on High Performance Computer Architecture (HPCA)*, 2009, pp. 45–54.

[62] M. K. Qureshi and G. H. Loh, "Fundamental Latency Trade-off in Architecting DRAM Caches: Outperforming Impractical SRAM-Tags with a Simple and Practical Design," in *45th International Symposium on Microarchitecture (MICRO)*, 2012, pp. 235–246.

[63] M. K. Qureshi and Y. N. Patt, "Utility-Based Cache Partitioning: A Low-Overhead, High-Performance, Runtime Mechanism to Partition Shared Caches," in *39th International Symposium on Microarchitecture (MICRO)*, 2006, pp. 423–432.

[64] D. Sanchez and C. Kozyrakis, "ZSim: Fast and Accurate Microarchitectural Simulation of Thousand-Core Systems," in *40th International Symposium on Computer Architecture (ISCA)*, 2013, pp. 475–486.

[65] Y. S. Shao, J. Clemons, R. Venkatesan, B. Zimmer, M. Fojtik, N. Jiang, B. Keller, A. Klinefelter, N. R. Pinckney, P. Raina, S. G. Tell, Y. Zhang, W. J. Dally, J. S. Emer, C. T. Gray, B. Khailany, and S. W. Keckler, "Simba: Scaling Deep-Learning Inference with Multi-Chip-Module-Based Architecture," in *52nd International Symposium on Microarchitecture (MICRO)*, 2019, pp. 14–27.

[66] D. D. Sharma, R. Blankenship, and D. S. Berger, "An Introduction to the Compute Express Link (CXL) Interconnect," *arXiv preprint arXiv:2306.11227*, June 2023.

[67] J. Sim, A. R. Alameldeen, Z. Chishti, C. Wilkerson, and H. Kim, "Transparent Hardware Management of Stacked DRAM as Part of Memory," in *47th International Symposium on Microarchitecture (MICRO)*, 2014, pp. 13–24.

[68] Y. Sun, Y. Yuan, Z. Yu, R. Kuper, C. Song, J. Huang, H. Ji, S. Agarwal, J. Lou, I. Jeong, R. Wang, J. H. Ahn, T. Xu, and N. S. Kim, "Demystifying CXL Memory with Genuine CXL-Ready Systems and Devices," in *56th International Symposium on Microarchitecture (MICRO)*, 2023, pp. 105–121.

[69] B. Tian, Q. Chen, and M. Gao, "ABNDP: Co-optimizing Data Access and Load Balance in Near-Data Processing," in *28th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2023, pp. 3–17.

[70] B. Tian, Y. Li, L. Jiang, S. Cai, and M. Gao, "NDPBridge: Enabling Cross-Bank Coordination in Near-DRAM-Bank Processing Architectures," in *51st International Symposium on Computer Architecture (ISCA)*, 2024, pp. 628–643.

[71] P. Tsai, N. Beckmann, and D. Sanchez, "Nexus: A New Approach to Replication in Distributed Shared Caches," in *26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2017, pp. 166–179.

[72] E. Vasilakis, V. Papaefstathiou, P. Trancoso, and I. Sourdis, "Hybrid2: Combining Caching and Migration in Hybrid Memory Systems," in *26th International Symposium on High Performance Computer Architecture (HPCA)*, 2020, pp. 649–662.

[73] Z. Wang, C. Liu, A. Arora, L. K. John, and T. Nowatzki, "Infinity Stream: Portable and Programmer-Friendly In-/Near-Memory Fusion," in *28th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2023, pp. 359–375.

[74] Z. Wang and T. Nowatzki, "Stream-based Memory Access Specialization for General Purpose Processors," in *46th International Symposium on Computer Architecture (ISCA)*, 2019, pp. 736–749.

[75] Z. Wang, J. Weng, S. Liu, and T. Nowatzki, "Near-Stream Computing: General and Transparent Near-Cache Acceleration," in *28th International Symposium on High Performance Computer Architecture (HPCA)*, 2022, pp. 331–345.

[76] Z. Wang, J. Weng, J. Lowe-Power, J. Gaur, and T. Nowatzki, "Stream Floating: Enabling Proactive and Decentralized Cache Optimizations," in *27th International Symposium on High Performance Computer Architecture (HPCA)*, 2021, pp. 640–653.

[77] H. Zeng, H. Zhou, A. Srivastava, R. Kannan, and V. K. Prasanna, "GraphSAINT: Graph Sampling Based Inductive Learning Method," in *8th International Conference on Learning Representations (ICLR)*, 2020.

[78] J. Zhan, I. Akgun, J. Zhao, A. Davis, P. Faraboschi, Y. Wang, and Y. Xie, "A Unified Memory Network Architecture for In-Memory Computing in Commodity Servers," in *49th International Symposium on Microarchitecture (MICRO)*, 2016, pp. 29:1–29:14.

[79] M. Zhang, Y. Zhuo, C. Wang, M. Gao, Y. Wu, K. Chen, C. Kozyrakis, and X. Qian, "GraphP: Reducing Communication for PIM-Based Graph Processing with Efficient Data Partition," in *24th International Symposium on High Performance Computer Architecture (HPCA)*, 2018, pp. 544–557.

[80] Z. Zhou, Y. Chen, T. Zhang, Y. Wang, R. Shu, S. Xu, P. Cheng, L. Qu, Y. Xiong, and G. Sun, "Toward CXL-Native Memory Tiering via Device-Side Profiling," *arXiv preprint arXiv:2403.18702*, March 2024.

[81] Z. Zhou, C. Li, F. Yang, and G. Sun, "DIMM-Link: Enabling Efficient Inter-DIMM Communication for Near-Memory Processing," in *29th International Symposium on High Performance Computer Architecture (HPCA)*, 2023, pp. 302–316.

[82] Y. Zhuo, C. Wang, M. Zhang, R. Wang, D. Niu, Y. Wang, and X. Qian, "GraphQ: Scalable PIM-Based Graph Processing," in *52nd International Symposium on Microarchitecture (MICRO)*, 2019, pp. 712–725.