NDPBridge: Enabling Cross-Bank Coordination in Near-DRAM-Bank Processing Architectures

Boyu Tian[†], Yiwei Li[†], Li Jiang[§‡¶], Shuangyu Cai[†] and Mingyu Gao[†‡]
Tsinghua University[†] Shanghai Qi Zhi Institute[‡] Shanghai Jiao Tong University[§] Huawei Technologies Co., Ltd.[¶]
tby20@mails.tsinghua.edu.cn liyw19@mails.tsinghua.edu.cn jiangli@cs.sjtu.edu.cn
caisy21@mails.tsinghua.edu.cn gaomy@tsinghua.edu.cn

***Abstract*—Various near-data processing (NDP) designs have been proposed to alleviate the memory wall challenge for data-intensive applications. Among them, near-DRAM-bank NDP architectures, by incorporating logic near each DRAM bank, promise the highest efficiency and have already been commercially available now. However, due to physical isolation, fast and direct cross-bank communication is impossible in these architectures, limiting their usage to only simple parallel patterns. Applications may also suffer from severe load imbalance if each bank contains data with diverse computation loads. We thus propose NDPBridge, with novel hardware-software co-design to enable cross-bank communication and dynamic load balancing for near-bank NDP systems. We introduce hardware bridges along the DRAM hierarchy to coordinate message transfers among banks. The hardware changes are constrained and do not disrupt the existing DDR links and protocols. We further enable hierarchical and data-transfer-aware load balancing, built upon the above hardware communication path and a task-based programming model. The data transfer overheads are minimized with several novel optimizations to hide latency, avoid congestion, and reduce traffic. Our evaluation shows that NDPBridge significantly outperforms existing NDP designs by $2.23\times$ to $2.98\times$ on average.**

***Index Terms*—near-data processing, processing-in-memory, DRAM, communication, load balance**

I. Introduction

In the big data era, the well-known memory wall problem [59] makes memory access a critical bottleneck that limits the overall system performance scaling. As a result, near-data processing (NDP) has been proposed as a promising solution, which aims to minimize the data movement cost by bringing computation closer to data locations. Prior research exploration and industrial prototypes have realized many variants of NDP systems for various applications, including graph analytics [2], [17], [61], [85], [90], neural networks [1], [27], [45], [88], genome analysis [13], [33], [48], databases [12], and many more [25], [35], [40], [42], [50].

Among the various types of DRAM-based NDP realization, in this work we mainly focus on the NDP variant that tightly integrates compute logic closest to data, at the lowest bank level in the modern DRAM hierarchy [1], [20], [30], [31], [40], [49], [52]. Specifically, each DRAM bank is associated with a small general-purpose core and some SRAM caches/buffers, forming an *NDP unit*. All the NDP units, usually on the order of several thousands, can process and access data from their local DRAM banks in parallel. Such DRAM-bank NDP architectures not only promise the highest bandwidth and energy efficiency gains due to short data access distances, but have also been made commercially available nowadays with several industrial products [20], [49], [52].

However, putting logic too close to DRAM banks also exhibits critical issues. The NDP units are now spread across different DRAM chips on different DIMMs and channels, far away from each other and without direct interconnect links among them. Thus direct **cross-bank communication** is typically not possible in DRAM-bank NDP systems, and we must rely on host CPU forwarding. Some previous designs realized cross-DIMM communication for NDP by adding extra physical links [73], [89], but such an approach is not practical when it comes to the bank level because of expensive intrusive modifications to the DRAM chips. In addition, because the NDP units can only directly access their local bank data, their workloads may suffer from severe imbalance if these data require different amounts of computations in complex and irregular applications. Effective **load balancing** must be done to ensure high-performance processing.

In this work, we propose NDPBridge, which *enables cross-bank communication and load balancing through hardware-software co-design* in near-DRAM-bank NDP architectures. **On the hardware level**, to support efficient cross-bank communication, we introduce several levels of *NDP bridges* that coordinate message transfers between NDP units. Each unit puts its outgoing messages in a local mailbox, and the upper-level bridge would proactively and periodically gather and scatter these messages across units. The communication is naturally hierarchical and matches the existing DRAM hierarchy of banks/ranks/channels. Therefore it localizes data movements and minimizes global transfers. To avoid prohibitive design complexity and reduce cost, we specifically limit any hardware changes to places that are already subject to modifications in prior NDP designs, and use existing physical links and DDR commands without altering the DIMM form factor. We also propose a dynamic triggering mechanism to balance communication cost and timeliness.

On the software level, we enable *hierarchical and data-transfer-aware load balancing* built on top of the above cross-bank communication capability. We adopt a task-based message-passing model that is expressive enough to represent most NDP applications [30], [40], [57], [62], and also allows us to flexibly schedule tasks among different NDP units. Nevertheless, a critical aspect of load balancing on NDP ar-

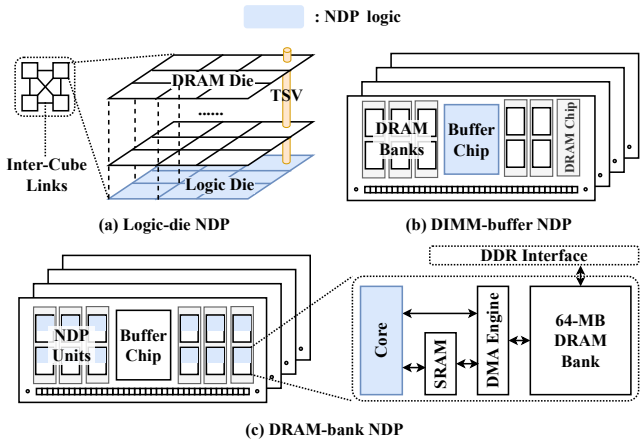

Fig. 1. Different variants of NDP architectures based on DRAM.

chitectures compared to traditional shared-memory systems, is the substantial cost of migrating data along with the scheduled tasks. We propose efficient metadata schemes to track migrated tasks/data, as well as novel data transfer optimizations that hide the latency, avoid the congestion, and reduce the traffic.

We evaluate NDPBridge with a wide range of NDP applications, by comparing it with existing DRAM-bank NDP architectures. Overall, NDPBridge enables an average 2.98× speedup and up to 5.22× with the combined hardware and software techniques. Specifically, the hardware bridge-based communication provides 1.51× performance gains over solely relying on the host CPU to forward messages. On top of that, our load balancing scheme adds another 1.98× by efficiently scheduling tasks and data among NDP units. NDPBridge is 2.23× faster than RowClone [70] which supports direct bank-to-bank transfers within each chip. We thus demonstrate that NDPBridge significantly enhances existing near-bank NDP systems with efficient cross-bank coordination capabilities, and extends their applicability to more complex applications.

II. Background and Motivation

A. Near-Data Processing

In this work, we mainly focus on NDP architectures based on DRAM technologies, which have been demonstrated to show immense potential for performance and energy efficiency. We categorize existing NDP architectures based on the location of compute logic. With 3D-stacked DRAM technologies which integrate multiple DRAM dies on top of a logic base, e.g., HMC and HBM [36], [39], [60], *logic-die NDP* places the NDP compute logic on the logic die [2], [9], [11], [17], [23], [25]–[27], [45], [61], [72], [76], [78], [84], [85], [90]. As shown in Figure 1(a), the NDP logic can access data stored in the DRAM dies using through-silicon vias (TSVs) with high bandwidth in parallel. On the other hand, to work with commercial DRAM like DDR4, the NDP logic can also be added along the DRAM hierarchical organization. *DIMM-buffer NDP* in Figure 1(b) integrates logic into the buffer chip of each DDR DIMM at the rank level, enabling higher bandwidth for data aggregation before transferring them through the memory channel [6], [18], [42], [50], [73], [88], [89]. This reduces data traffic and enhances system efficiency. *DRAM-bank NDP* associates logic with one or several DRAM banks as in Figure 1(c) [1], [20], [30], [31], [40], [49], [52]. More aggressive NDP architectures incorporate fine-grained compute logic *at DRAM sense amplifiers* [4], [53], [54]. In this work, we focus on DRAM-bank NDP.

B. DRAM-Bank NDP

The high-level architecture of DRAM-bank NDP is depicted in Figure 1(c). For each of the many DRAM banks in the system, a processing core is associated with it, which forms an *NDP unit*. Through a DMA engine, the core can access data stored in the local bank. An SRAM buffer is also included to ease programming and optimize memory access performance. The SRAM buffer can be implemented either as a cache or as a programmer-controlled scratchpad in different designs.

DRAM-bank NDP has been not only widely studied in academia, but also there are already several commercial products available [20], [49], [52]. For example, in UPMEM [20], the first commercial NDP architecture, the NDP units are called DPUs. Each DPU contains a wimpy core, a 64 MB DRAM bank, and 88 kB SRAM buffers for the instruction and data scratchpads. A typical UPMEM server can scale to 20 dual-rank DIMMs, with 128 DPUs per DIMM. These 2560 DPUs work in parallel and their aggregated DRAM bandwidth can reach 1.7 TB/s. Another DRAM-bank NDP product, HBM-PIM [49], [52], is based on 3D-stacked HBM [36], [39]. Similarly, each HBM bank is incorporated with an execution engine comprising a floating-point SIMD unit. HBM-PIM uses command and scalar register files instead of SRAM buffers.

To better use DRAM-bank NDP, the software also needs certain adjustments, of which one important issue is data interleaving [30], [62]. To ensure an NDP unit holds a contiguous range of data needed for computation in its local bank, the UPMEM SDK provides a data transposition procedure to realize coarse-grained interleaving [20]. HBM-PIM also provides an automatic data layout rearrangement mechanism in its BLAS API [52]. We assume a similar design exists in this work so that each NDP unit has its data in the local bank.

C. Limitations of DRAM-Bank NDP

Despite the significant benefits in access bandwidth and energy efficiency offered by DRAM-bank NDP architectures, there are still inherent limitations that hinder their widespread adoption. We identify the two most critical issues as the *lack of cross-bank communication support* and *load imbalance*.

Lack of cross-bank communication support. In DRAM, direct data communication between two DRAM banks is usually not possible. In the context of DRAM-bank NDP, each NDP unit corresponds to a DRAM bank. The NDP core can only access data stored in the local bank. We refer to this model as *data-local execution*. In existing designs [20], [52], cross-unit communication requires moving data to the host CPU first, and then back to the intended receiving

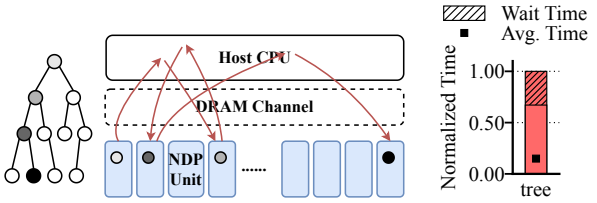

Fig. 2. Inefficiencies when executing tree traversal on the baseline DRAM-bank NDP architecture. Simulated in a 512-unit system with 2 memory channels (Section VII). Data transfers are done by host CPU forwarding.

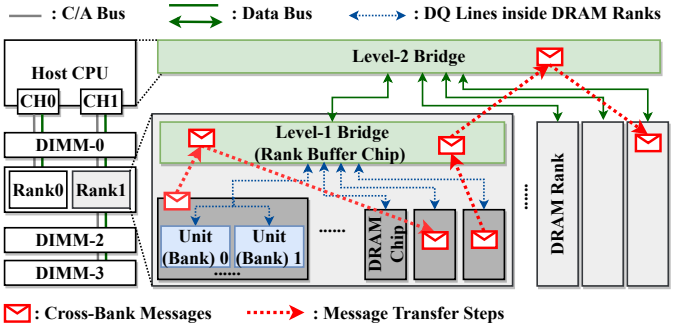

Fig. 3. NDPBridge overall architecture.

unit. Such back-and-forth data transfers on bandwidth-limited DRAM channels dwarf the NDP benefits and incur prohibitive overheads. Specifically, Figure 2 left shows the workflow of tree traversal on DRAM-bank NDP architectures. Each tree node processing executes in the bank that stores the node. When going to the child node, if it is stored in another bank, it cannot be processed by the current NDP unit due to data-local execution. The host CPU must fetch the results and forward to the destination units. As shown in Figure 2 right, in the total execution time, we highlight the 32.9% wait time as the difference between the total time and the time for actual task execution, which is mainly caused by cross-bank data communication. As a result, the applicability of DRAM-bank NDP is effectively restricted to simple embarrassingly parallel applications where different units rarely share data.

It is challenging to support such cross-bank communication. Traditional wisdom for communication support in NDP systems relies on adding extra physical links between units. In logic-die NDP, the NDP units are directly connected through inter- and intra-stack memory networks, through which they can communicate with each other [46], [47], [63], [65], [83]. Initially, DIMM-buffer NDP relies on the host CPU to forward cross-DIMM packets, which is inefficient. Recent designs like DIMM-Link [89] add external physical links between DIMMs to enable fast inter-DIMM data transfers, but do not enable bank-to-bank communication within a DIMM. However, adding physical links no longer works when it comes to the bank level. This will require extensive modifications inside DRAM chips, which disrupts the form factor of standard DRAM and is severely restricted by the available metal layers in the DRAM technology, as also noticed in previous work [62]. RowClone [70] utilizes the existing data bus shared by all banks in a DRAM chip for cross-bank communication within a chip, which has been adopted by several follow-up designs [44], [74], [87]. However, banks in different DRAM chips remain isolated.

Load imbalance. While providing abundant parallelism, the thousands of NDP units make it challenging to fully utilize the compute logic resources and achieve load balance when processing complex applications. Due to the lack of cross-unit communication capability, traditional DRAM-bank NDP designs only use static workload assignment, which not only burdens the programmer with manual workload distribution, but in most cases is difficult to achieve good enough balance. In Figure 2 right, the overall time is determined by the slowest NDP unit. We also mark the average time of all units in the figure. The big difference between the maximum and the average time indicates significant load imbalance.

Nevertheless, even after we enable efficient cross-unit communication, supporting dynamic load balancing is still non-trivial. Traditional dynamic scheduling techniques in shared-memory systems only need to transfer some computation tasks from busy units to idle units, but not the data, because all units can access the data in roughly the same performance. However, in DRAM-bank NDP, remote data accesses are significantly slower than local accesses. If only moving the computation to the new unit but leaving the data at the original location, too many remote accesses will occur during the computation and eliminate any performance gains. Therefore, dynamic load balancing in DRAM-bank NDP should not only transfer the computation tasks but also the associated data, which we call as *data-first scheduling*. To support this, first, we need to do additional bookkeeping to track those data moved out of their original locations and ensure coherence. Second, both the data and computation movements could incur significant delays under limited cross-unit communication speed. This communication overhead has a crucial impact on the performance of the dynamic load balancing (see Section VI-C).

III. NDPBridge Overview

In this work, we propose NDPBridge, aiming to resolve the two limitations of cross-bank communication and load imbalance in DRAM-bank NDP systems through *hardware-software co-design*. Figure 3 shows the overall architecture.

The hardware aspect of NDPBridge supports efficient cross-bank communication by introducing multiple levels of *bridges* into the DRAM hierarchy (Section V). The NDP logic at each bank puts its outgoing messages in a dedicated mailbox region in its local DRAM, and the upper-level bridges would proactively gather and scatter the messages in a hierarchical manner, realizing flexible cross-bank communication. Such message gathering/scattering *uses existing physical links and DDR commands*, and only *adds limited hardware structures to places in DRAM that are already subject to modifications* in NDP systems. Therefore the implementation complexity and cost are well bounded.

The software aspect of NDPBridge enables effective load balancing across all near-bank processing units in the system, built on top of the hardware cross-bank communication capability. First, NDPBridge embraces *a task-based message-passing programming and execution model* (Section IV), where NDP applications are naturally represented by large amounts of parallel data-centric tasks. Tasks can be moved across banks as messages to execute near their data, and further be scheduled to achieve load balance. The load balancing scheme in NDPBridge is *hierarchical and data-transfer-aware* (Section VI). We propose key optimizations to alleviate the impact of long-latency data migration during load balancing, including hiding transfer latency, avoiding transfer congestion, and reducing transfer traffic.

IV. Programming Model

NDPBridge uses a task-based message-passing programming model for DRAM-bank NDP architectures, similar to prior designs [38], [57], [76]. A task represents the operations on one data element, e.g., a graph vertex in graph processing, a tree node in tree traversal, or a matrix row in matrix-vector multiplication. Such a data-centric task definition matches well with applications that apply near-data processing on a large amount of data elements. It also provides a natural abstraction for workload scheduling and allows the runtime system to automatically assign and schedule in the task granularity.

More specifically, a task consists of the following attributes: a function pointer, a timestamp for synchronization, a physical address of the data that this task operates on, an optional estimation of the task's workload, and any number of additional arguments. The timestamp is used to support bulk-synchronization [37], [38], [76], [82]. Tasks with the same timestamp can be executed in parallel. The estimated workload is used to aid load balancing, as discussed later in Section VI. It can be inaccurate or even unspecified because our task scheduling is done dynamically. If a task needs to trigger operations on other data elements, it can generate child tasks, in a similar fashion of remote procedure calls. The child task generation is done by the parent calling the following API:

```
enqueue_task(func_ptr, timestamp, data_ptr,
             workload, args...)
```

Then the new task is scheduled, and pushed to the unit where the corresponding data element resides for processing, which aligns well with data-local execution in DRAM-bank NDP. Algorithm 1 shows an example of implementing tree traversal using our programming model.

Our programming model mainly has two differences compared to previous task-based models. First, each task is associated with one data element. While it may initially appear as a strict constraint, numerous data-intensive applications are inherently composed of separate computations performed on distinct data elements, especially for those executed on DRAM-bank NDP that adopts data-local execution (Section II-C). Previous NDP studies have already presented many applications adhered to this criterion [30], [40], [57], [62].

Algorithm 1: Tree traversal task.

```
1 function TreeTrav(ts, n, q):
      input: timestamp ts, tree node n, query q.
2     if n.value == q then return n.id ;
3     else if n.value < q && n.leftPtr ≠ nullptr then
4         return enqueue_task(TreeTrav, ts, n.leftPtr, 1, q)
5     else if n.value > q && n.rightPtr ≠ nullptr then
6         return enqueue_task(TreeTrav, ts, n.rightPtr, 1, q)
7     return invalid_id;
```

Furthermore, we can still support tasks accessing multiple data elements in the way described below. Second, for applications where one task involves multiple data elements, the communication is done through *pushing tasks* to remote units by message passing instead of *pulling data* to local units in typical shared-memory systems. For example, stencil computing can be implemented through two steps: (1) each pixel pushes its current value (by invoking tasks) to all its neighbors; (2) each pixel uses the received value to update its own value. Pushing tasks has several benefits over pulling data. First, it eliminates the need of maintaining coherence of multiple data copies across thousands of units, which is unlikely to scale. Second, pulling data requires sending requests and then receiving data, while pushing tasks only needs one-way transmission, reducing traffic. Lastly, pulling remote data usually exhibits very long latencies in DRAM-bank NDP due to inefficient communication, which incurs idle waiting cycles in NDP cores. Although techniques such as multi-threading may help, they incur extra hardware and software complexity. In contrast, neither the execution of the local nor remote cores is blocked during task pushing. Therefore, the push-based model is also widely adopted in previous NDP systems [2], [46], [64].

Our programming model complies with the philosophy of MapReduce in that users focus on the local action of each data element, and the system composes these actions to process a large dataset. The programming burden of our model is acceptable and worthwhile. Programmers do need to manually invoke tasks and specify data elements, which can both be done only with application knowledge. Note that NDP systems usually allocate large and contiguous virtual address ranges [2], [3], [15], [46], [51], [61] or directly work on the physical address space [20], [30], [44], [64], so specifying data element addresses is feasible in user applications. In exchange, they are free from concerning the underlying communication implementation and load balancing, both of which require deep understanding of the DRAM-bank NDP hardware.

V. Hardware Support for Communication

NDPBridge realizes hierarchical message communication across NDP units, with the help of multiple levels of *bridges*. At each level, the children are passive; only the parent bridge actively gathers/scatters messages among the children. Our specific system uses two levels. The level-1 bridges enable message passing among NDP units within each rank, without

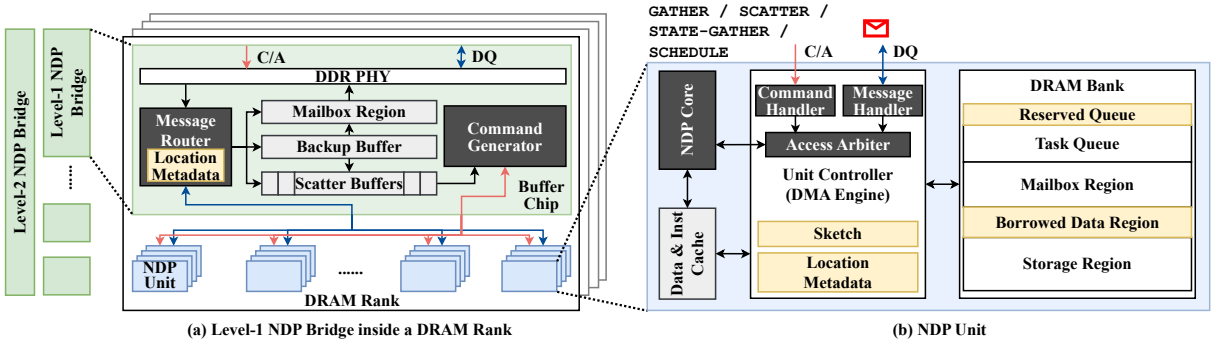

Fig. 4. Detailed hardware structures at the NDP units and the bridges in NDPBridge. The modules marked in yellow are the new hardware support for load balancing, which will be elaborated in Section VI-A.

data traffic from/to the host CPU. The level-2 bridges further connect the level-1 bridges for cross-rank communication.

A. Hardware Modifications

The hardware modifications are mainly in two places of the system: at the NDP units and at the bridges.

NDP units. Figure 4(b) demonstrates the architecture of an NDP unit in NDPBridge. Similar to previous DRAM-bank NDP designs (Section II-B), there is a DRAM bank, a wimpy core, and SRAM buffers as data and instruction caches. The core accesses the DRAM bank through a DMA engine, which is extended with more functionalities in our design and renamed as the *unit controller*. We add a task queue in the local DRAM to support task-based execution in Section IV. The core fetches tasks from the local task queue to execute.

The majority of the local DRAM space is used as the normal data storage region for local computation. Besides, we statically allocate a portion of the DRAM bank, called the *mailbox region*, to store cross-unit communication messages waiting to be sent out. It is implemented as a ring buffer, with its head and tail pointers maintained in the unit controller. New messages are added to the tail. When the region is full, the unit controller stalls the next message enqueue.

The parent bridge gathers and scatters messages to the NDP unit by issuing standard DDR commands through the existing DDR interface (C/A and DQ links). We explain the details in Section V-B. The gather/scatter requests are parsed by the *command handler*. For a gather request, it fetches messages from the head of the mailbox region. For a scatter request, the messages are directly written to their destinations by the *message handler* in the unit controller. For example, a received task is enqueued to the task queue, and a received data element (for load balancing discussed below) is stored into the *borrowed data region* in the DRAM.

Since now the DRAM accesses could come from both the local NDP core and the upper-level bridge, we add an *access arbiter* to coordinate the accesses from the two, similar to previous work [15]. Note that such arbitration is needed even in the baseline design to support concurrent accesses from the NDP core and the host CPU. Such bank-side arbitration eliminates the need for complex modifications to the host memory controller. The controller can transmit DDR commands from the host CPU seamlessly, while actual memory accesses from both the host and the NDP unit are coordinated near the bank. We follow previous designs to optimize issues like DRAM row buffer conflicts and write-read turn-around delays [15].

Our dynamic load balancing scheme requires extra hardware support. In-SRAM *location metadata* track the migrated data elements. An in-DRAM *borrowed data region* stores the data elements that are scheduled in. The SRAM-based *sketch* structure and the in-DRAM *reserved queue* are used to identify the hottest data and the corresponding tasks for scheduling. We explain how these modules are used in Section VI.

Bridges. We explain the design of the level-1 (rank-level) bridges as in Figure 4(a). The level-1 bridge is implemented in the DIMM buffer chip, with mainly three SRAM data buffers and some auxiliary logic. Our design requires DRAM modules with buffer chips. For a rank with N banks, we add N *scatter buffers* to store the messages to be scattered to these banks. The gathered messages are routed by the *message router* into these scatter buffers based on their destinations. For the messages whose destinations are in other ranks and should be sent to the upper level, we store them in a *mailbox region* similar to the one inside the NDP unit. If any of the scatter buffers or the mailbox region is full, the gathered messages are temporarily stored in a *backup buffer*. If the backup buffer is also full, the bridge will pause gathering messages. The scatter buffer is 1 kB per bank, in total 64 kB for 64 banks in a rank. We use 128 kB for the mailbox region, and 64 kB for the backup buffer. A *command generator* forges the DDR commands needed for gathering/scattering messages from/to the banks (Section V-B). Note that it is not a full memory controller (e.g., no scheduling) so its logic is quite simple. We do not need to schedule these forged DDR commands here, because the actual DDR accesses are scheduled and arbitrated by the access arbiter at the bank, as mentioned above [15]. Finally, in-SRAM *location metadata* are needed to support load balancing, similar to those in the NDP units.

The upper-level bridges have similar hardware architectures. The simplest way is to implement the level-2 bridge in the host CPU using a software runtime (as in our evaluation), similar to the baseline but now we only deal with cross-rank

messages rather than all cross-bank messages. Alternatively, we can leverage previous DIMM-buffer NDP communication designs, such as adding peer-to-peer links between DIMMs as in DIMM-Link [89], or broadcast links between DIMMs as in ABC-DIMM [73]. NDPBridge is orthogonal to and can work in tandem with them. In addition, these previous designs only implemented physical links, while NDPBridge also needs extra metadata bookkeeping for load balancing.

Hardware cost. We emphasize that the hardware modifications of NDPBridge incur low implementation complexity and cost. All the added components are confined to the places that have already been subject to modifications in existing NDP systems, e.g., at the DIMM buffer chips [6], [42], [50], [89] and besides the DRAM banks [20], [30], [49], [52]. Some of these changes have even been justified by commercial products [20], [30], [43], [49], [52]. First, the DIMM form factor could remain unaltered. NDPBridge does not require additional physical pins or links inside the DIMM. Moreover, the DDR interface between the DIMMs and the host CPU remains unaffected. Second, the area cost is negligible. We implement the added logic modules in Verilog and synthesize them in the TSMC 28 nm technology. We use CACTI 7.0 [7] to model the added SRAM storage. For the bridge, the logic is only 0.002 52 mm^2. The SRAM has 1.25 MB capacity in total (Table I) and occupies 1.46 mm^2. Together the area overheads are only 1.46% of the rank buffer chip area [42]. For an NDP unit, we need to add 0.000 134 mm^2 logic plus 20.2 kB SRAM, which are minor compared to the existing core and caches.

Compatibility with split DIMM buffers. Our current level-1 bridge design relies on a *unified* buffer chip per rank for both data and C/A links. This requirement follows most previous DDR4-based DIMM-buffer NDP designs [24], [42], [50]. Nevertheless, it has been pointed out that since DDR4, separate data buffers (DBs) and a registering clock driver (RCD) chip are preferred to further improve the signal integrity of DQ [6]. NDPBridge can also be implemented with such a DIMM type. Since in this case each DB chip is attached to a set of DRAM chips, a level-1 bridge among these DRAM chips can be implemented in the DB chip. The DQ bus multiplexing mechanism in Chameleon [6] can be used to transfer the generated commands from DB chips to DRAM banks. Specifically, Chameleon puts NDP accelerators in the DBs, which are connected to their corresponding DRAM chips through only the DQ links. Therefore it proposes temporal and spatial multiplexing to transfer not only data but also C/A commands on these DQ links, respectively called *chameleon-t* and *chameleon-s* [6]. Such multiplexing would slightly sacrifice data bandwidth for dispatching C/A. In our design, the level-1 bridge is in the DB, and can use similar designs to multiplex the DQ links for sending generated commands to NDP units. The RCD and the BCOM interface between the RCD and the DBs are not needed. This design would incur slightly higher hardware overheads than our default implementation with a unified buffer, because more level-1 bridges must be implemented in multiple DB chips. The message router and the command generator must be replicated in each DB, and consume 0.0201 mm^2 in total assuming eight DBs per rank. In contrast, the added SRAM buffers could be split among the DBs to keep the total size the same, i.e., 1.46 mm^2 as above. The hardware cost of link multiplexing is negligible as reported in Chameleon [6]. Among different DB chips, the level-2 bridge can use the host CPU or requires an extra chip in the DIMM, which may need form factor changes. We show the performance evaluation of this design in Section VIII-A.

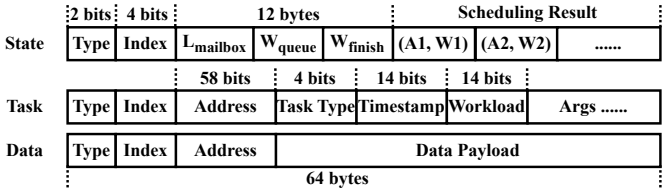

Fig. 5. Message formats.

B. Communication Protocols

Message types. There are three types of messages in NDPBridge: task messages, data messages, and state messages. Figure 5 shows their formats, with a type field to distinguish the types. The maximum message size is 64 bytes. If a message is too large, we divide it into multiple small sub-messages. The index field indicates such a sequence.

The state message is used by the upper-level bridges to gather state information of their children, including the length of the mailbox region L_{mailbox}, the workload amount of tasks in the task queue W_{queue}, and the finished workload amount W_{finish}. These states are used to decide whether to trigger a new round of message gathering and load balancing, which will be introduced in Sections V-C and VI. When the children need to respond to such a scheduling round, they also append the list of data to be scheduled out (the addresses and the associated workload amounts) in the state message.

The task message is used to transfer a task to another NDP unit, either because the data element to be processed is remote, or due to load balancing. It contains the address of the corresponding data element, the task type that selects the task function pointer, and the other arguments following the programming API in Section IV.

Finally, the data message transfers a chunk of local data to another NDP unit, in order to allow the corresponding tasks on this data chunk to be scheduled to other units for load balancing (Section VI). If the data chunk is longer than the message payload, multiple messages are composed.

Communication procedures. NDPBridge uses three procedures for cross-unit communication: *state gathering*, *message gathering*, and *message scattering*. We introduce three commands in the bridge: `STATE-GATHER`, `GATHER`, and `SCATTER`. These commands are implemented using existing DDR commands and sent through existing C/A links, but target specially reserved row and column addresses outside the physical DRAM array range. We denote the reserved row/column as `R_ROW` and `R_COL`, respectively. The command handler in

the NDP unit controller (Figure 4(b)) detects these reserved addresses and decodes accordingly. Other command encoding approaches are also possible [3], [6], [31], [42].

A STATE-GATHER command to a bank is a DDR ACTIVATE command to R_ROW. The child's unit controller prepares the state information and responds with a state message as above. Since the state information is maintained inside the unit controller and not stored in the mailbox, state messages will not be blocked by other messages, and can be delivered timely.

The bridge can also issue a GATHER command to gather task/data messages from its children, by sending a DDR READ command to R_COL. After decoding the command in the command handler, the unit controller fetches messages from the head of the mailbox region, and returns it to the bridge. The bridge decides the destination NDP unit of the gathered messages based on the address field in the message, following the default address mapping if the data element is not migrated, or using the location metadata in the bridge otherwise. The message is then stored into one of the scatter buffers or the mailbox region, according to the destination.

Finally, the bridge scatters the task/data messages in the scatter buffers to their destinations. It sends a SCATTER command, which is a DDR WRITE to R_COL. For the lowest level, upon receiving a task message, the unit controller inserts the task into the task queue to wait for execution. If it is a data message, the data chunk is stored into the borrowed data region, and the location metadata are updated. If the child is another bridge, the received message will be routed to the corresponding scatter buffer and later sent to the lower level.

Although each message is 64 bytes, we use a larger granularity G_{xfer} for GATHER/SCATTER message transfers to achieve better channel bandwidth utilization. G_{xfer} is by default 256 bytes in NDPBridge. We empirically decide it in Section VIII-C. It is also used as the granularity of load balancing in Section VI.

We further leverage memory-level parallelism in commercial DRAM architectures to optimize the communication. Note that all DRAM chips in a rank share the same C/A links, and their DQ links are bundled into the wider channel. Therefore, at the level-1 bridge, a single command can be issued to the same banks in all the chips simultaneously, with different messages gathered/scattered. For example, assume eight x8 DRAM chips compose a 64-bit channel. A GATHER command can access bank 0 across all eight chips in parallel. After a 64-bit data piece is received in each half-cycle, it is decomposed into eight 8-bit slices, each from one bank. The bridge internally handles such layout transformation. Then the bridge gathers from bank 1 across all chips, and so on.

C. Dynamic Communication Triggering

In NDPBridge, the upper-level bridge actively initiates communication among its lower-level children. The state gathering is lightweight as only one message is returned per bank. It happens periodically every I_{state} cycles, by default 2000 cycles (we evaluate other values in Section VIII-C). However, how frequently to do a round of task/data message gathering/scattering needs careful consideration. We denote the minimum interval as I_{min}, which is the time to finish communication across all children (recall that each bank in a chip is round-robin accessed). Too frequent communication may cause bandwidth waste. This is because message gathering/scattering is done at a fixed access granularity G_{xfer}. If there is no pending message, a GATHER would be wasted. In contrast, insufficient communication may leave many tasks in the mailbox regions not sent to their destinations in time, incurring idle core cycles.

We propose a simple but efficient dynamic scheme to trigger message gathering/scattering. The key idea is that, when there are idle units, we communicate more frequently to promptly deliver tasks; otherwise, we wait until we can fully utilize bandwidth. Specifically, if an NDP unit's mailbox is empty, the bridge does not gather. If the length L_{mailbox} exceeds the transfer granularity G_{xfer}, the bridge immediately starts a message gathering. Otherwise, the bridge only gathers if there are idle children based on W_{queue}, with the frequency of I_{min}. L_{mailbox} and W_{queue} are available in the collected state message.

VI. Load Balancing Scheme

The capability of cross-unit communication in NDPBridge further enables dynamic load balancing. However, as discussed in Section II-C, dynamic load balancing for DRAM-bank NDP faces unique challenges and requires a data-first scheduling paradigm, where the data must be migrated first before a task can be scheduled to another unit. Next we propose *hierarchical and data-transfer-aware load balancing*, including efficient metadata tracking of migrated data, and scheduling policies aware of data migration cost.

We call the idle unit that receives tasks as the *receiver*, and the busy unit that sends out tasks as the *giver*. The giver would also *lend* the corresponding data to the receiver.

A. Load Balancing Workflow

We design a hierarchical load-balancing workflow. At each level, the bridge periodically collects state messages from its children. When it detects idle children (precisely defined in Section VI-C), it initiates the load balancing process. To leverage the DRAM hierarchy and reduce expensive cross-rank communication, NDPBridge first tries to balance loads within each rank. Only if all local NDP units in a rank are idle, the level-1 bridge would report to the level-2 bridge, which then initiates cross-rank load balancing. This hierarchical procedure is similar to prior NUMA-aware load balancing schemes [8], [14], [21], [22], [66], [71].

The load balancing workflow includes the following steps as shown in Figure 6. ❶ **Bridge commands scheduling:** The bridge randomly matches each idle receiver with one or more non-idle givers, and uses a SCHEDULE command (see below) to send each giver a *budget* value representing the amount of workload that should be scheduled out of this giver. The budget value of a giver is the sum of the required workload of all receivers matched to the giver. The required workload of each receiver is introduced in Section VI-C. ❷ **Giver chooses tasks:** When a giver receives the budget value, it locally chooses enough tasks to satisfy this budget (Section VI-C) and

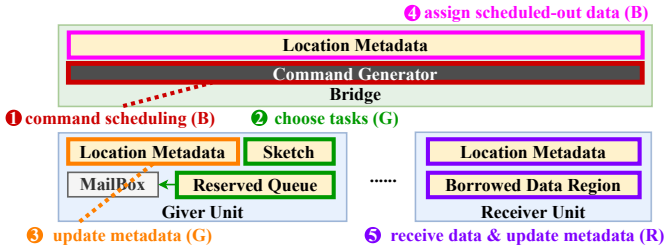

Fig. 6. Load balancing workflow. Only hardware modules involved in load balancing are depicted, and marked by the colored boxes corresponding to the execution steps they are involved in. B, G, and R represent bridge, giver, and receiver, respectively.

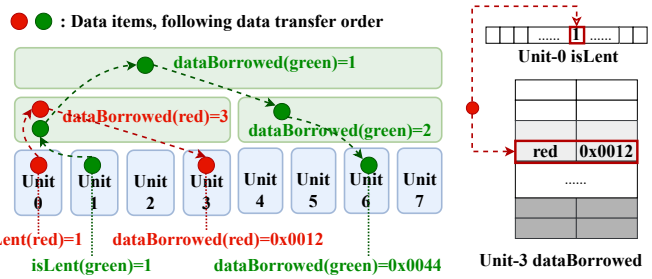

Fig. 7. Data migration workflow and corresponding metadata structures.

puts the selected tasks and their corresponding data into the mailbox. ❸ **Giver updates metadata:** The list of scheduled-out tasks/data is temporally kept in the unit controller, and is returned to the bridge through the next state message. ❹ **Bridge assigns scheduled-out data:** After the bridge collects the scheduling result, it assigns the data to the receivers based on the previous matching. The bridge also maintains the location metadata accordingly (Section VI-B). ❺ **Receiver receives data and updates metadata:** When a data message containing new data elements arrives at the receiver, the receiver updates the local metadata and stores the data elements to the borrowed data region in the DRAM. The metadata scheme is further elaborated in Section VI-B. Note that load balancing happens hierarchically at each level. The level-2 bridge only assigns budgets and coordinates data among the level-1 bridges, while each level-1 bridge is responsible for locally lending out enough tasks from its own children to satisfy the budget, or assigning received tasks to the children.

The `SCHEDULE` command can be implemented in several ways, similar to Section V-B or following previous work [3], [42]. We use the DDR `ACTIVATE` command. The row address encodes the budget, by concatenating a non-existing row address prefix (recall that we reserve some row address bits) with the actual budget value.

B. Metadata Scheme

In NDPBridge, both the NDP units and the bridges need to keep metadata to track where data have been migrated. Each NDP unit has two metadata structures. The first one is a bitmap `isLent`, one bit per each size-G_{xfer} block, to indicate whether the block is currently lent to another unit. The other one is a set-associative table called `dataBorrowed`, which tracks all received data blocks borrowed from other units by storing their current addresses in the borrowed data region in the local DRAM. Each entry is essentially a key-value pair of ⟨ original block address → remapped block address ⟩. A table entry may be replaced (following LRU), in which case the corresponding data block is returned to its original home unit. In each rank-level bridge, we also maintain a `dataBorrowed` table, but its entry only stores the receiver unit ID instead of the detailed address. The two levels of `dataBorrowed` tables are inclusive. Here we do not need an `isLent` bitmap to mark which data blocks are lent to other ranks. If a block is marked `isLent` $= 1$ in its home NDP unit, but does not appear in the rank-level `dataBorrowed` table, it means the block is in another rank.

Figure 7 shows how to manage these metadata structures when lending data. For the red data block lent from unit 0 to 3, we set its `isLent` in unit 0, and update `dataBorrowed` in level-1 bridge 0 with ⟨ red → 3 ⟩ and that in unit 3 with ⟨ red → `0x0012` ⟩. For the green data block lent from unit 1 to 6 across ranks, we set `isLent` in unit 1, update `dataBorrowed` in both levels of bridges, and finally that in unit 6. These metadata updates happen along with data message passing. Therefore, updating `dataBorrowed` in the receiver is always later than updating `isLent` in the giver, ensuring that a block is never simultaneously available at both units. For a block whose original address is x, if this is the home unit of x and `isLent`$(x) = 0$, the block is local. If this is not the home unit but `dataBorrowed` has an entry for x, it is also locally stored. Other than these two cases, the block is not available locally.

In our default configuration (Table I), we use a 2 kB `isLent` table and a 16 kB, 8-way `dataBorrowed` table in each NDP unit (0.19 mm^2). We add a 1 MB, 16-way `dataBorrowed` table in each bridge (1.18 mm^2 or 1.18% of the buffer chip).

C. Data-Transfer-Aware Scheduling Policy

An obvious but vital characteristic of data-first scheduling is that *data transfers take time*. As a result, there could be a long delay between issuing a load balancing request and actually receiving the tasks/data at the receiver unit. During this period, the scheduling is not yet effective and the receiver unit remains idle, yielding limited load balancing benefits or even adversely impacting performance due to extra data transfers. This is not an issue in traditional shared-memory systems that only schedule tasks but not migrating data. Hence in NDPBridge, data transfers should be our primary focus of consideration, and call for specific *data-transfer-aware* optimizations.

With Figure 8, we summarize three goals of data-transfer-aware scheduling, namely *hiding transfer latency*, *avoiding transfer congestion*, and *reducing transfer traffic*. We show that directly applying traditional work stealing fails to achieve these goals. Then we propose our optimizations for each goal.

Hiding transfer latency. Traditional load balancing policies steal tasks from others when the local task queue is empty. If data transfers are slow, it would result in notable idle time in the receiver unit, as shown in Figure 8 left. NDPBridge

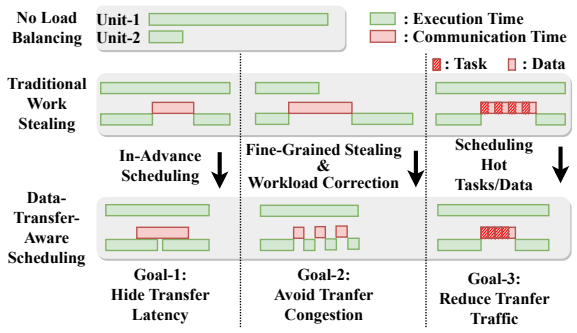

Fig. 8. Three key design considerations in data-transfer-aware scheduling.

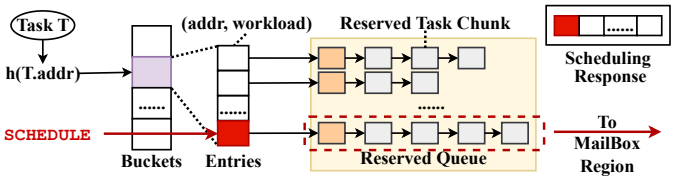

Fig. 9. SRAM-based sketch and in-DRAM reserved task queue to identify and schedule hot data/tasks in NDPBridge.

schedules tasks *in advance* before the task queue becomes empty, thus overlapping the transfer latency with useful computations. The key question is how advance it should be. We set a threshold W_{th}. When the remaining task queue workload $W_{\text{queue}} < W_{\text{th}}$, the upper-level bridge should start load balancing to refill the task queue. We determine the threshold to fully hide the data transfer latency, as $W_{\text{th}} = \frac{2 \times G_{\text{xfer}} \times S_{\text{exe}}}{S_{\text{xfer}}}$. Here S_{exe} and S_{xfer} represent the average speeds of executing task workloads and transferring data blocks between NDP units and the bridge. The bridge can easily estimate their values through the gathered state messages. The factor of 2 accounts for transfers to and from the bridge.

Avoiding transfer congestion. The amount of stolen tasks also needs re-calibration. Traditional wisdom steals half of the victim queue length [10], [71], which may be too much for NDPBridge and cause back-and-forth task migration due to several reasons. First, task transfers now take longer time, which adds to the receiver's overall time and may make it a new straggler as in Figure 8 middle. Second, tasks are dynamically generated in our model as time goes on. When the stolen tasks arrive after too long, the original idle unit may already become busy. Third, with data-first scheduling, the migrated data elements would automatically attract more tasks afterwards. Fourth, when tasks have been assigned to a receiver but are still being transferred, the receiver should not request more tasks. Therefore, NDPBridge uses *fine-grained stealing*, to only transfer a small amount of workload to the receiver. We set the amount to a small multiple of W_{th}, e.g., $2 \times W_{\text{th}}$. The idea is to just ensure the unit has enough work to do before the next load balancing round, which will transfer tasks to it again if there are still not enough tasks. Furthermore, we apply *workload correction* to account for the already scheduled but still transferring tasks. The bridge keeps an `toArrive` counter for each NDP unit for such pending loads, and uses $W_{\text{queue}} + \texttt{toArrive}$ to estimate the actual workload.

Reducing transfer traffic. The previous optimizations mainly find a better timing to transfer data. But if there are simply too many data to transfer, the overheads would still dominate. Next we reduce the amount of data transfers through better selecting which data blocks and tasks to lend out.

Traditional work stealing simply chooses tasks from the tail of the victim queue. However, in our case, stealing different tasks would lead to quite different data traffic, because data elements have drastically different hotness. *Selecting hot data blocks* allows for less data traffic for the same amount of task workload. As in the right side of Figure 8, to schedule four tasks, with cold data elements each only processed by one task, four elements must be sent. Obviously, a better way is to migrate a single hot data element associated with four tasks.

Following this insight, NDPBridge aims to *identify and schedule hot data elements and their corresponding tasks* when doing load balancing. We use a *sketch*-based solution to track hot data in the block granularity. The sketch [16], [79]–[81], [86] is a data structure widely used in networking measurement, using a small storage to efficiently and approximately identify hot items in a data stream. We use a simplified design similar to HeavyGuardian [79], while we do not need to track cold data. Each NDP unit maintains a sketch as a set-associative buffer. It is composed of several buckets indexed by the data address, and each bucket stores a list of entries as $\langle$ address, workload $\rangle$, as shown in Figure 9 left. When a task on data x and of workload w comes to the unit, it searches address x in the sketch. On a hit, w is added to the stored entry. On a miss, if there is empty space, a new entry $\langle x, w \rangle$ is inserted. Otherwise, we find the entry e_{min} in the bucket with the lowest workload, and decay e_{min}.workload by w with a probability $\mathcal{P} = b^{-e_{\text{min}}.\text{workload}}$, where $b = 1.08$ offers the best accuracy as mathematically proved [79]. If e_{min}.workload becomes negative, we remove it and insert $\langle x, w \rangle$. We use sixteen 16-entry buckets, with 58-bit block addresses and 1-byte workload counters. The sketch size is just 2 kB.

The tasks associated with each data block in the sketch are reserved in a special *reserved queue* in the local DRAM, and will be prioritized for load balancing. The reserved queue is organized in size-G_{xfer} chunks. As shown in Figure 9 right, each sketch entry is statically assigned a chunk as the initial space to store its associated tasks. If this chunk is full, new chunks are dynamically allocated to form a linked list. A bitmap is used to track such allocation, 1 bit per chunk. We manage 1280 chunks per NDP unit, for about 10000 tasks.

When we need to extract tasks to schedule out, we first find the hottest entry from the sketch. We move the corresponding data block, and the associated tasks from the reserve queue chunks, into the mailbox region waiting to be transferred. This entry is then removed from the sketch, and the chunks are cleared/freed. If more tasks are needed, the next hottest sketch entry is extracted. If the sketch becomes empty, we fall back to use the tasks from the tail of the task queue.

TABLE I
SYSTEM CONFIGURATIONS.

NDP system	2 channels × 4 ranks × 8 chips × 8 banks 512 NDP units in total (1 units per bank) 32 GB capacity in total (64 MB per bank)
NDP core **L1-D cache** **L1-I cache** **DRAM bank**	In-order, 400 MHz, 10 mW 64 kB, 4-way, 64 B cachelines, LRU 32 kB, 2-way, 64 B cachelines, LRU 64 bits I/O; 17 ns CAS/RCD/RP; 150 pJ/64bit RD/WR
NDP Unit	SRAM: 2 kB `isLent`; 16 kB 8-way `dataBorrowed` DRAM: 1 MB mailbox, 1 MB borrowed data region
NDP Bridge	SRAM: 64 kB scatter buffers, 64 kB backup buffer SRAM: 128 kB mailbox, 1 MB 16-way `dataBorrowed`
Sketch	16 buckets, 16 entries/bucket, 1-byte counter per entry
Comm.	G_{xfer} = 256 byte, I_{state} = 2000 cycles 2400 MT/s × 64 bits between level-1 and level-2 2400 MT/s × 8 bits between NDP units and level-1

TABLE II
EVALUATED DRAM-BANK NDP SYSTEMS.

Design	Communication	Load balancing
C	Forwarded by host CPU	None
B	Using bridges (**ours**)	None
W	Using bridges (**ours**)	Work stealing
O	Using bridges (**ours**)	Data-transfer-aware (**ours**)

Hardware cost. We summarize the newly introduced hardware structures to support load balancing in NDPBridge. Each unit needs a `toArrive` counter, a sketch, and a bitmap for reserve queue allocation. The total SRAM capacity is less than 2.2 kB. The bridge needs logic to estimate S_{exe} and S_{xfer}, by processing gathered state information.

VII. METHODOLOGY

System models. We use zsim [68], a Pin-based simulator, to model DRAM-bank NDP systems. We extend zsim with task-based execution similar to Swarm [38] and modify the previous DRAM modules [25] to support independent accesses to individual DRAM banks from NDP cores. Table I summarizes our system configurations. We assume a wimpy in-order core in each NDP unit at 400 MHz following UPMEM [20]. Each core consumes 10 mW similar to ARM Cortex M3 [5]. We use CACTI 7.0 [7] to model the timing and energy of SRAM caches/buffers and metadata storage. Our DRAM parameters follow the configurations of UPMEM [20], [30], i.e., each UPMEM module is a DDR4-2400 style DIMM with 8 DRAM chips and 8 banks per chip. Each 64-bit DRAM read/write to a bank costs 150 pJ based on evaluations on UPMEM [20]. The data movement energy on the off-chip channels is obtained from [25]. For cross-rank communication, we do not use DIMM-Link [89] but simply use the original DDR channels. DIMM-Link cannot help communication within a rank.

Applications. We evaluate eight data-intensive parallel applications: linked list traversal (`ll`), hash table (`ht`), tree traversal (`tree`), sparse matrix-vector multiplication (`spmv`), breadth-first search (`bfs`), single-source shortest path (`sssp`), Page Rank (`pr`), and weakly connected component (`wcc`). We port the workloads into our task-based message-passing model, similar to existing parallel benchmark suites [30], [58]. We use real-world graphs [55] and datasets [19] for the graph applications and `spmv`. We generate data and queries for `ll`, `ht`, and `tree` following a Zipfian distribution [91].

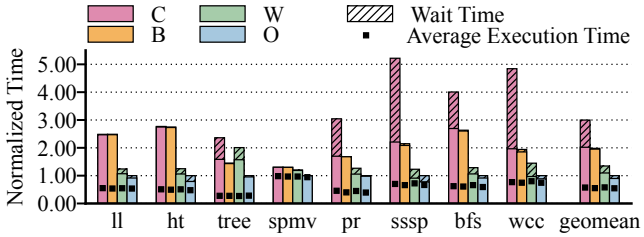

Fig. 10. Overall performance comparison between NDPBridge and baselines. Normalized to **O**. The bars and the square marks show the maximum and the average time across NDP units.

Baselines. We compare NDPBridge with three baseline DRAM-bank NDP designs summarized in Table II. The baseline **C** (for CPU) relies on host CPU forwarding to do cross-unit communication and applies no dynamic load balancing. This is the same as the execution model of existing DRAM-bank NDP products [20]. **B** (for bridges) uses the proposed hardware bridges for communication, but without doing load balancing. **W** (for work-stealing) further uses traditional work stealing approaches for load balancing. We also apply workload correction (Section VI-C) to **W**, since it is a simple and natural trick for load balancing with centralized control at the bridge. Finally **O** combines all of our designs in NDPBridge.

We also compare with two baselines with different architectures. First, **H** is a non-NDP system that only uses the host CPU to execute the same task-based applications. The host CPU has 16 out-of-order cores at 2.6 GHz, a 20 MB last-level cache, and two DDR4-2400 channels. With shared memory, each core can freely steal tasks from other cores' task queues. Second, to compare with previous cross-bank communication schemes, we include **R**, which leverages RowClone [70] for cross-bank data transfers within each DRAM chip. Communication between different chips is done through forwarding by the host, the same as **C**. We do not apply load balancing in **R** since it cannot be implemented only using the hardware modifications in RowClone.

VIII. EXPERIMENTAL RESULTS

A. Overall Comparison

Figure 10 compares the execution time of different designs. Similar to Figure 2, we show the overall time determined by the slowest NDP unit, and the average time of all units. Their difference indicates load imbalance. We also highlight the wait time due to communication.

First, for the communication overhead, the wait time takes a significant portion of the total execution time in **C**, up to 57.7% for `sssp`. **B** reduces this overhead to only 1.4%, and thus

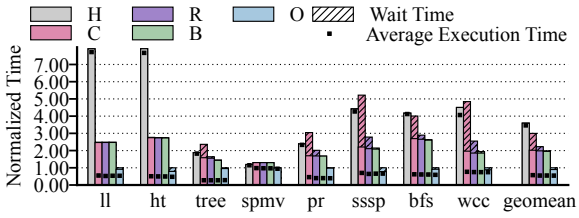

Fig. 11. Overall performance comparison between NDPBridge and other architectures: host execution without NDP (**H**), intra-chip cross-bank communication in RowClone (**R**). Normalized to **O**.

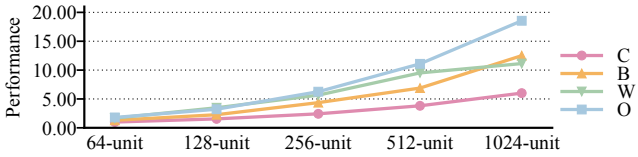

Fig. 12. Scalability comparison between NDPBridge and baselines when running `pr`. Normalized to **C** with 64 units.

achieves 1.51× better performance than **C** on average, which demonstrates the efficiency of using our hardware bridges for cross-unit communication rather than using host CPU forwarding. Applications `ll`, `ht`, and `spmv` do not exhibit communication overheads because their data are distributed in a way that does not need cross-unit communication when load balancing is not enabled. For example, each linked list in `ll` and each bucket of `ht` are fully stored in one NDP unit [30], [57], and thus each query goes only to this unit.

However, **B** still suffers from the load imbalance problem. The average execution time is only 22.4% of the maximum. **W** alleviates load imbalance by using work stealing through bridge-enabled communication. It achieves a 1.45× speedup than **B** and 2.23× than **C** on average. The average execution time now becomes 47.0% of the maximum. However, **W** is not aware of the message transfer overheads during task scheduling. It incurs more message transfers than **B** for many applications, and the wait time increases from 1.4% to 18.6% of the execution time on average. This restricts the performance gains of scheduling, and sometimes even hurts the performance (e.g., `tree`).

Finally, with both the efficient bridge-enabled cross-unit communication capability and the specially designed data-transfer-aware load balancing policy in NDPBridge, **O** achieves the best performance among the designs, 2.98× faster than **C** and 1.98× than **B**. It achieves fairly good load balancing, with the average execution time being 59.0% of the maximum. But scheduling tasks to balance loads indeed incurs more message communication, so the wait time in **O** is 10.0% of the overall time, higher than **B** without load balancing. Nevertheless, **O** is able to reach a good tradeoff between the two. The communication time is much lower than using the basic work-stealing, resulting in a 1.35× speedup over **W**.

Notice that although **W** and **O** use dynamic scheduling, neither can balance the loads perfectly as in shared-memory systems where the average time is almost the same as the maximum. This is due to two reasons. First, the NDP unit state information is collected periodically and hierarchically, which may be inaccurate and stale. Second, the load balancing cost is higher in NDP due to more expensive message transfers. Sometimes it may be better to not schedule out tasks because transferring them to other units may take even longer time.

Figure 11 shows the comparison with non-NDP host execution (**H**) and RowClone (**R**). **C** is only 19.9% faster than **H** and sometimes even performs worse, due to the wimpy NDP cores as well as the high communication cost and the severe load imbalance. With all of our optimizations applied, **O** can achieve an average 3.59× speedup than **H**. With **R**, communication between banks in the same DRAM chip can be accelerated. Thus it is 1.35× faster than **C** due to the more efficient communication (75.2% less wait time). However, in the evaluated workloads the communication across chips is more critical, so using bridges in **B** achieves a 1.12× speedup than **R**. **O** further enables load balancing (which is not discussed in RowClone [70]) and achieves an overall 2.23× speedup than **R**. Notice that regarding the communication within a DRAM chip, RowClone's scheme is faster than ours since it only needs one instruction and does not need to store messages inside the bridges. We believe NDPBridge and RowClone can be combined to accelerate both intra-chip and inter-chip data communication.

Recall that NDPBridge is also compatible with split DIMM buffers, as discussed at the end of Section V-A. In this design, since the DQ pins for data transfers are multiplexed for C/A dispatching, the communication bandwidth between NDP units and level-1 bridges is lower than our default implementation, which hurts performance. We now evaluate this implementation. We use the *chameleon-s* design in which two out of the eight DQ pins of each DRAM chip are dedicated to C/A, which shows the best performance in Chameleon [6]. We use the host CPU as the level-2 bridge. The evaluation shows a 9.1% performance degradation compared to the default design, with a 35.3% increase in the wait time.

Scalability. Figure 12 measures the scalability of NDPBridge by using different system scales from 64 to 1024 units, corresponding to 1 to 16 ranks (all with 64 units per rank). We can see that the speedup of NDPBridge over the baselines gets larger as the system scales up. As the number of NDP units increases, data become more distributed and tasks become more imbalanced, making both cross-unit communication and load balancing more critical. Our techniques have good scalability. Specifically, even at 1024 units, **O** still gets a 1.68× speedup compared to its 512-unit setting. The multi-level bridge hierarchy is the key. NDPBridge can confine intra-rank communications under a level-1 bridge, without involving the level-2 bridge. As a result, the traffic through the level-2 bridge is significantly less than those of the level-1 bridges, e.g., only 40.4% at 512 units. This also allows for using the same-sized message buffers in the level-2 bridge, even though they

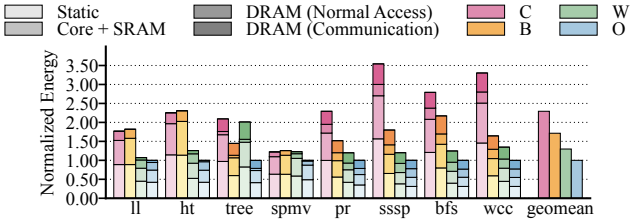

Fig. 13. Overall energy comparison between NDPBridge and baselines. Normalized to **O**.

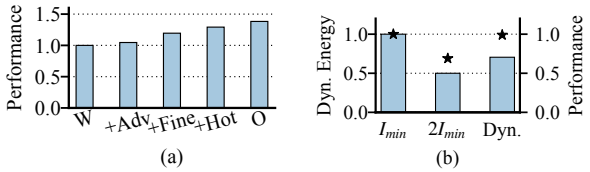

Fig. 14. (a) Impact of data-transfer-aware load balancing techniques. (b) Impact of dynamic communication triggering. Averaged across all applications.

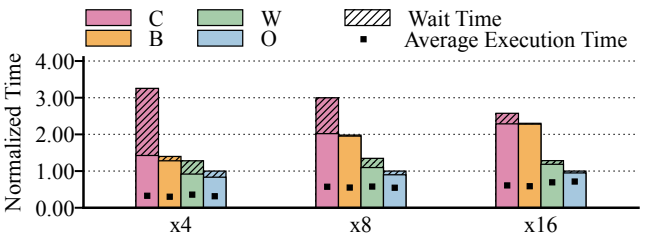

Fig. 15. Impact of DRAM chip DQ pin widths. Averaged across all applications. Normalized to **O** within each configuration.

are smaller than the aggregated size of all the level-1 bridge buffers. Nevertheless, the cross-rank traffic would still increase as the system further scales, e.g., to 56.1% at 1024 units. Future work could investigate more complex interconnection across ranks and better data partitioning schemes.

Notice that **W** fails to outperform **B** at 1024 units, because the task scheduling traffic gets too much and the communication cost outweighs the load balancing benefit. In contrast, data-transfer-aware load balancing in **O** is always beneficial.

Energy. The energy consumption of each design is shown in Figure 13, which breaks down into four components: 1) NDP cores and SRAM caches, 2) local DRAM bank accesses, 3) DRAM bank accesses for cross-unit communication, and 4) static. The static energy and the core energy roughly follow the performance trend. Although the data traffic is generally higher with load balancing enabled (**W** and **O**), sometimes the communication energy may be lower than **C** and **B** (e.g., `pr`, `bfs`). This is because with balanced load and faster execution, we can avoid doing unnecessary message gathering/scattering and thus reduce communication. Since `ll`, `ht`, and `spmv` do not incur communication if not enabling load balancing, **B** does not provide any energy saving over **C**, but consumes more energy due to the added SRAM structures. Overall, NDPBridge consumes the least energy over all designs, with an average 56.4% reduction compared to **C**.

B. Detailed Performance Analysis

We now study the detailed tradeoffs in the NDPBridge optimizations. Figure 14(a) investigates data-transfer-aware load balancing in Section VI-C. The baseline is **W**. We individually apply each of the three techniques, in-advanced scheduling (**+Adv**) to hide latency, fine-grained stealing (**+Fine**) to avoid congestion, and scheduling hot tasks/data (**+Hot**) to reduce traffic. Using in-advance scheduling alone provides the lowest performance gain, only 4.6% on average. This is because there is a large amount of data traffic and it is hard to hide their latency. Fine-grained stealing achieves 1.19×. Leveraging hot tasks/data could directly reduce data traffic, and thus offers the highest speedup of 1.29×. The three optimizations complement each other from different perspectives. When combined, **O** yields 1.35× performance gains over **W**.

Figure 14(b) studies the effect of dynamic communication triggering in Section V-C. Compared with using a fixed $I_{\min}$ interval, dynamic triggering reduces the access energy by 29.5% by avoiding unnecessary gathering/scattering. The performance is not sacrificed, degrading only 0.4%, showing our scheme is effective in ensuring prompt message transfers. In contrast, simply reducing the communication frequency to $2I_{\min}$ significantly hurts the performance by 31.0%.

Our previous experiments assume x8 DRAM chips and eight chips per rank to compose the 64-bit channel. We now investigate the performance benefits of NDPBridge when using other DQ pin widths (x4 and x16) while keeping the same channel I/O width of 64 bits. Specifically, each rank can be made of sixteen x4 chips, or four x16 chips. We also maintain the same number of ranks in the system to ensure similar ratios of intra-rank vs. inter-rank communication. Consequently, there are 1024 (2 channels × 4 ranks/channel × 16 chips/rank × 8 banks/chip), 512, and 256 banks in the x4, x8, and x16 configurations, respectively. Figure 15 shows that, NDPBridge (**O**) can also work efficiently with x4 and x16 DRAM chips, achieving 3.26× and 2.58× speedups over **C**, respectively. Due to the narrower DQ links to each chip, the communication bottleneck is more critical in the x4 DRAM configuration. Therefore, our bridge-based communication (**B**) yields the highest performance gain of 2.33× over **C** with x4 chips, while it is only 1.12× with x16 chips. On the other hand, communication bandwidth also influences load balancing, as limited communication bandwidth may cause congestion in the mailbox and prevent scheduled data and tasks from being quickly transferred out. So **W** and **O** provide higher performance gains against **B** with x16 chips, 1.79× and 2.3× respectively. In contrast, they only achieve 1.09× and 1.4× with x4 chips. These results demonstrate the necessity of optimizing both the hardware communication mechanism and the software load balancing strategy, as NDPBridge does, in order to achieve robust performance at various configurations.

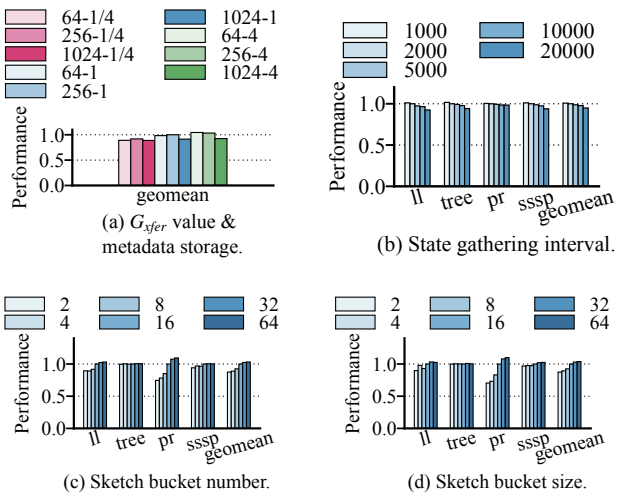

Fig. 16. Impact of various design parameter choices in NDPBridge. The geomean is averaged across all applications.

C. Design Parameter Choices

Figure 16(a) shows the impact of different G_{xfer} values as well as the sizes of the metadata tables (`isLent` and `dataBorrowed`). We use 64 B, 256 B (default), and 1024 B for G_{xfer}, and sweep 1/4, 1, and 4 times for the metadata storage size. The results show that the current 256-byte granularity is a good balance. Using 64 B can provide slightly better performance if permitting $4\times$ metadata storage.

Figure 16(b) shows the influence of I_{state} which is how often to do state information gathering with `STATE-GATHER`. 2000 cycles could retain sufficient performance.

Figure 16(c,d) investigate the sketch size (the number of buckets and the number of entries per bucket). Our default 16-bucket, 16-entry configuration is sufficiently good. Although a larger sketch may provide slightly better performance for some applications, it occupies larger area and increases design cost.

IX. Related Work

DRAM-bank NDP architectures. DRAM-bank NDP provides the highest aggregated memory access bandwidth and parallelism among all the NDP variants. Both research prototypes [1], [31], [77] and commercialized products [20], [30], [49], [52] have been proposed following this paradigm. Previous work has also optimized specific applications on the commercial NDP products [28]–[30], [34], [40], [41]. Our software programming model eases programming on DRAM-bank NDP, and our hardware modifications enable cross-bank communication and dynamic load balancing.

Communication support for NDP. Communication support is a vital topic for all three kinds of NDP architectures. Logic-die NDP implements cross-unit communication through inter/intra-stack interconnects [46], [47], [63], [65], [83]. Designs have been proposed to reduce data traffic on the interconnect by better data placement [75], [85], caching remote data [76], and reducing intermediate results on the way [32], [69]. NoM [67] directly connected the many banks in a 3D-stacked memory using additional physical links, which incurred prohibitive hardware modifications and would be difficult to apply to 2D DRAM chips. For DIMM-buffer NDP, which also relies on expensive host CPU forwarding for communication, previous research added peer-to-peer links [89] or broadcast support [73] between DIMMs. These techniques cannot be applied to DRAM-bank NDP directly, since there are no direct interconnects between banks and the physical constraints are much more severe inside DRAM chips.

HBM-PIM [52] required data layout changes to boost performance for BLAS-style computations. Its API allowed for automatically rearranging the data layout by the host processor when bringing the weight matrix into the memory. But there was no direct cross-bank communication support. Each unit operated on the partial data of a vector/matrix resided in the local bank. For other irregular data types, ensuring data locality would also require additional data interleaving. RowClone-based designs [44], [70], [74], [87] utilized the shared data bus inside a DRAM chip to support cross-bank communication only within a chip. This intra-chip communication mechanism can be combined with our inter-chip design.

Load balancing for NDP. Effective load balancing is crucial to the performance of NDP architectures that consist of massive parallel processing units. Most previous work relied on static partitioning techniques to balance loads [2], [17], [40], [56], [64], [85], [88], [90]. PIM-tree [40] particularly optimized ordered index workloads for DRAM-bank NDP architectures. However, these domain-specific designs put great burdens on programmers and cannot be applied to other applications. Dynamic load balancing has also been studied for other types of NDP [76] and other platforms such as NUMA systems [8], [14], [21], [22], [66], [71]. These designs cannot be applied to vanilla DRAM-bank NDP architectures that lack communication support. Building upon the communication support in NDPBridge, we further propose several new data-transfer-aware scheduling policies to address the unique load balancing challenges in DRAM-bank NDP.

X. Conclusions

This paper proposes a hardware-software co-design approach to improve DRAM-bank NDP architectures, through enabling cross-bank communication and load balancing. Hardware bridges are added along the DRAM hierarchy to hierarchically forward data among banks. Computation tasks can be scheduled to idle bank logic to avoid imbalanced loads, with novel optimizations to reduce data transfer overheads. Combining the hardware and software techniques allows our system to outperform existing baselines.

Acknowledgment

The authors thank the anonymous shepherd and 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] S. Aga, N. Jayasena, and M. Ignatowski, "Co-ML: A Case for Collaborative ML Acceleration Using Near-Data Processing," in *International Symposium on Memory Systems (MEMSYS)*, 2019.

[2] J. Ahn, S. Hong, S. Yoo, O. Mutlu, and K. Choi, "A Scalable Processing-in-Memory Accelerator for Parallel Graph Processing," in *42nd ACM/IEEE Annual International Symposium on Computer Architecture (ISCA)*, 2015.

[3] J. Ahn, S. Yoo, O. Mutlu, and K. Choi, "PIM-Enabled Instructions: A Low-Overhead, Locality-Aware Processing-in-Memory Architecture," in *42nd ACM/IEEE Annual International Symposium on Computer Architecture (ISCA)*, 2015.

[4] M. A. Z. Alves, P. C. Santos, M. Diener, and L. Carro, "Opportunities and Challenges of Performing Vector Operations Inside the DRAM," in *International Symposium on Memory Systems (MEMSYS)*, 2015.

[5] ARM, "Arm Cortex-M3," 2004, https://developer.arm.com/Processors/Cortex-M3.

[6] H. Asghari-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 Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2016.

[7] 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, 2017.

[8] N. Ben-David, Z. Scully, and G. E. Blelloch, "Unfair Scheduling Patterns in NUMA Architectures," in *28th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2019.

[9] M. Besta, R. Kanakagiri, G. Kwasniewski, R. Ausavarungnirun, J. Beránek, K. Kanellopoulos, K. Janda, Z. Vonarburg-Shmaria, L. Gianinazzi, I. Stefan, J. G. Luna, J. Golinowski, M. Copik, L. Kapp-Schwoerer, S. Di Girolamo, N. Blach, M. Konieczny, O. Mutlu, and T. Hoefler, "SISA: Set-Centric Instruction Set Architecture for Graph Mining on Processing-in-Memory Systems," in *54th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2021.

[10] R. D. Blumofe and C. E. Leiserson, "Scheduling Multithreaded Computations by Work Stealing," *Journal of the ACM (JACM)*, vol. 46, no. 5, 1999.

[11] 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.

[12] A. Boroumand, S. Ghose, G. F. Oliveira, and O. Mutlu, "Polynesia: Enabling Effective Hybrid Transactional/Analytical Databases with Specialized Hardware/Software Co-Design," in *38th IEEE International Conference on Data Engineering (ICDE)*, 2022.

[13] D. S. Cali, G. S. Kalsi, Z. Bingöl, C. Firtina, L. Subramanian, J. S. Kim, R. Ausavarungnirun, M. Alser, J. Gomez-Luna, A. Boroumand, A. Norion, A. Scibisz, S. Subramoneyon, C. Alkan, S. Ghose, and O. Mutlu, "GenASM: A High-Performance, Low-Power Approximate String Matching Acceleration Framework for Genome Sequence Analysis," in *53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2020.

[14] Q. Chen, Y. Chen, Z. Huang, and M. Guo, "WATS: Workload-Aware Task Scheduling in Asymmetric Multi-Core Architectures," in *IEEE 26th International Parallel and Distributed Processing Symposium (IPDPS)*, 2012.

[15] B. Y. Cho, Y. Kwon, S. Lym, and M. Erez, "Near Data Acceleration with Concurrent Host Access," in *ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, 2020.

[16] G. Cormode and S. Muthukrishnan, "An Improved Data Stream Summary: The Count-Min Sketch and Its Applications," *Journal of Algorithms*, vol. 55, no. 1, 2005.

[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, 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 Annual International Symposium on Computer Architecture (ISCA)*, 2022.

[19] T. A. Davis and Y. Hu, "The University of Florida Sparse Matrix Collection," *ACM Transactions on Mathematical Software (TOMS)*, vol. 38, no. 1, 2011.

[20] F. Devaux, "The True Processing in Memory Accelerator," in *IEEE Hot Chips 31 Symposium (HCS)*, 2019.

[21] A. Drebes, A. Pop, K. Heydemann, A. Cohen, and N. Drach, "Scalable Task Parallelism for NUMA: A Uniform Abstraction for Coordinated Scheduling and Memory Management," in *25th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2016.

[22] A. Drebes, A. Pop, K. Heydemann, N. Drach, and A. Cohen, "NUMA-Aware Scheduling and Memory Allocation for Data-Flow Task-Parallel Applications," in *21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2016.

[23] A. Farmahini-Farahani, J. H. Ahn, K. Morrow, and N. S. Kim, "NDA: Near-DRAM Acceleration Architecture Leveraging Commodity DRAM Devices and Standard Memory Modules," in *21st IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2015.

[24] S. Feng, X. He, K.-Y. Chen, L. Ke, X. Zhang, D. Blaauw, T. Mudge, and R. Dreslinski, "MeNDA: A Near-Memory Multi-Way Merge Solution for Sparse Transposition and Dataflows," in *49th ACM/IEEE Annual International Symposium on Computer Architecture (ISCA)*, 2022.

[25] 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.

[26] M. Gao and C. Kozyrakis, "HRL: Efficient and Flexible Reconfigurable Logic for Near-Data Processing," in *22nd IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2016.

[27] M. Gao, J. Pu, X. Yang, M. Horowitz, and C. Kozyrakis, "Tetris: Scalable and Efficient Neural Network Acceleration with 3D Memory," in *20nd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2017.

[28] C. Giannoula, I. Fernandez, J. Gómez-Luna, N. Koziris, G. Goumas, and O. Mutlu, "Towards Efficient Sparse Matrix Vector Multiplication on Real Processing-in-Memory Architectures," *ACM SIGMETRICS Performance Evaluation Review*, vol. 50, no. 1, 2022.

[29] J. Gómez-Luna, Y. Guo, S. Brocard, J. Legriel, R. Cimadomo, G. F. Oliveira, G. Singh, and O. Mutlu, "Evaluating Machine Learning Workloads on Memory-Centric Computing Systems," in *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2023.

[30] J. Gómez-Luna, I. E. Hajj, I. Fernandez, C. Giannoula, G. F. Oliveira, and O. Mutlu, "Benchmarking a New Paradigm: Experimental Analysis and Characterization of a Real Processing-in-Memory System," *IEEE Access*, vol. 10, 2022.

[31] 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 *47nd ACM/IEEE Annual International Symposium on Computer Architecture (ISCA)*, 2020.

[32] J. Huang, R. Reddy Puli, P. Majumder, S. Kim, R. Boyapati, K. H. Yum, and E. J. Kim, "Active-Routing: Compute on the Way for Near-Data Processing," in *25th IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2019.

[33] W. Huangfu, X. Li, S. Li, X. Hu, P. Gu, and Y. Xie, "MEDAL: Scalable DIMM Based Near Data Processing Accelerator for DNA Seeding Algorithm," in *52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2019.

[34] M. Item, G. F. Oliveira, J. Gómez-Luna, M. Sadrosadati, Y. Guo, and O. Mutlu, "TransPimLib: Efficient Transcendental Functions for Processing-in-Memory Systems," in *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2023.

[35] J. Jang, J. Heo, Y. Lee, J. Won, S. Kim, S. J. Jung, H. Jang, T. J. Ham, and J. W. Lee, "Charon: Specialized Near-Memory Processing Architecture for Clearing Dead Objects in Memory," in *52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2019.

[36] JEDEC, "High Bandwidth Memory (HBM) DRAM," 2021, https://www.jedec.org/standards-documents/docs/jesd235a.

[37] M. C. Jeffrey, S. Subramanian, M. Abeydeera, J. Emer, and D. Sanchez, "Data-Centric Execution of Speculative Parallel Programs," in *49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2016.

[38] M. C. Jeffrey, S. Subramanian, C. Yan, J. Emer, and D. Sanchez, "A Scalable Architecture for Ordered Parallelism," in *48th Annual*

IEEE/ACM International Symposium on Microarchitecture (MICRO), 2015.

[39] H. Jun, J. Cho, K. Lee, H.-Y. Son, K. Kim, H. Jin, and K. Kim, "HBM (High Bandwidth Memory) DRAM Technology and Architecture," in *IEEE International Memory Workshop (IMW)*, 2017.

[40] 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," *Proc. VLDB Endow.*, vol. 16, no. 4, dec 2022.

[41] ——, "PIM-trie: A Skew-Resistant Trie for Processing-in-Memory," in *35th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2023.

[42] L. Ke, U. Gupta, B. Y. Cho, D. Brooks, V. Chandra, U. Diril, A. Firoozshahian, K. Hazelwood, B. Jia, H.-H. S. Lee, M. Li, B. Maher, D. Mudigere, M. Naumov, M. Schatz, M. Smelyanskiy, X. Wang, B. Reagen, C.-J. Wu, M. Hempstead, and X. Zhang, "RecNMP: Accelerating Personalized Recommendation with Near-Memory Processing," in *ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, 2020.

[43] L. Ke, X. Zhang, J. So, J.-G. Lee, S.-H. Kang, S. Lee, S. Han, Y. Cho, J. H. Kim, Y. Kwon, K. Kim, J. Jung, I. Yun, S. J. Park, H. Park, J. Song, J. Cho, K. Sohn, N. S. Kim, and H.-H. S. Lee, "Near-Memory Processing in Action: Accelerating Personalized Recommendation with AxDIMM," *IEEE Micro*, vol. 42, no. 1, 2021.

[44] D. Kim, J.-Y. Kim, W. Han, J. Won, H. Choi, Y. Kwon, and J.-Y. Kim, "MPU: Towards Bandwidth-Abundant SIMT Processor via Near-Bank Computing," *arXiv preprint arXiv:2305.13970*, 2023.

[45] D. Kim, J. Kung, S. Chai, S. Yalamanchili, and S. Mukhopadhyay, "NeuroCube: A Programmable Digital Neuromorphic Architecture with High-Density 3D Memory," in *43rd ACM/IEEE Annual International Symposium on Computer Architecture (ISCA)*, 2016.

[46] G. Kim, N. Chatterjee, M. O'Connor, and K. Hsieh, "Toward Standardized Near-Data Processing with Unrestricted Data Placement for GPUs," in *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2017.

[47] 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.

[48] 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. 2, 2018.

[49] Y.-C. Kwon, S. H. Lee, J. Lee, S.-H. Kwon, J. M. Ryu, J.-P. Son, O. Seongil, H.-S. Yu, H. Lee, S. Y. Kim, Y. Cho, J. G. Kim, J. Choi, H.-S. Shin, J. Kim, B. Phuah, H. Kim, M. J. Song, A. Choi, D. Kim, S. Kim, E.-B. Kim, D. Wang, S. Kang, Y. Ro, S. Seo, J. Song, J. Youn, K. Sohn, and N. S. Kim, "25.4 A 20nm 6GB Function-in-Memory DRAM, Based on HBM2 with a 1.2TFLOPS Programmable Computing Unit Using Bank-Level Parallelism, for Machine Learning Applications," in *IEEE International Solid- State Circuits Conference (ISSCC)*, vol. 64, 2021.

[50] Y. Kwon, Y. Lee, and M. Rhu, "TensorDIMM: A Practical Near-Memory Processing Architecture for Embeddings and Tensor Operations in Deep Learning," in *52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2019.

[51] J. H. Lee, J. Sim, and H. Kim, "BSSync: Processing Near Memory for Machine Learning Workloads with Bounded Staleness Consistency Models," in *International Conference on Parallel Architecture and Compilation (PACT)*, 2015.

[52] S. Lee, S.-h. Kang, J. Lee, H. Kim, E. Lee, S. Seo, H. Yoon, S. Lee, K. Lim, H. Shin, J. Kim, O. Seongil, 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 *ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, 2021.

[53] M. Lenjani, A. Ahmed, M. Stan, and K. Skadron, "Gearbox: A Case for Supporting Accumulation Dispatching and Hybrid Partitioning in PIM-Based Accelerators," in *49th Annual International Symposium on Computer Architecture (ISCA)*, 2022.

[54] M. Lenjani, P. Gonzalez, E. Sadredini, S. Li, Y. Xie, A. Akel, S. Eilert, M. R. Stan, and K. Skadron, "Fulcrum: A Simplified Control and Access Mechanism Toward Flexible and Practical In-Situ Accelerators," in *IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2020.

[55] J. Leskovec and R. Sosič, "SNAP: A General-Purpose Network Analysis and Graph-Mining Library," *ACM Transactions on Intelligent Systems and Technology (TIST)*, vol. 8, no. 1, 2016.

[56] H. Liu, L. Zheng, Y. Huang, C. Liu, X. Ye, J. Yuan, X. Liao, H. Jin, and J. Xue, "Accelerating Personalized Recommendation with Cross-Level Near-Memory Processing," in *50th Annual International Symposium on Computer Architecture (ISCA)*, 2023.

[57] E. Lockerman, A. Feldmann, M. Bakhshalipour, A. Stanescu, S. Gupta, D. Sanchez, and N. Beckmann, "Livia: Data-Centric Computing Throughout the Memory Hierarchy," in *25th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2020.

[58] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: A System for Large-Scale Graph Processing," in *ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 2010.

[59] S. A. McKee, "Reflections on the Memory Wall," in *1st Conference on Computing Frontiers*, 2004.

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

[61] L. Nai, R. Hadidi, J. Sim, H. Kim, P. Kumar, and H. Kim, "GraphPIM: Enabling Instruction-Level PIM Offloading in Graph Computing Frameworks," in *23rd IEEE Symposium on High Performance Computer Architecture (HPCA)*, 2017.

[62] J. Nider, C. Mustard, A. Zoltan, J. Ramsden, L. Liu, J. Grossbard, M. Dashti, R. Jodin, A. Ghiti, J. Chauzi, and A. Fedorova, "A Case Study of Processing-in-Memory in Off-the-Shelf Systems," in *USENIX Annual Technical Conference (USENIX ATC)*, 2021.

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

[64] M. Orenes-Vera, E. Tureci, D. Wentzlaff, and M. Martonosi, "Dalorex: A Data-Local Program Execution and Architecture for Memory-Bound Applications," in *IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2023.

[65] 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 ACM/IEEE Annual International Symposium on Computer Architecture (ISCA)*, 2017.

[66] I. Psaroudakis, T. Scheuer, N. May, A. Sellami, and A. Ailamaki, "Adaptive NUMA-Aware Data Placement and Task Scheduling for Analytical Workloads in Main-Memory Column-Stores," in *46th International Conference on Very Large Data Bases (VLDB)*, 2016.

[67] S. H. S. Rezaei, M. Modarressi, R. Ausavarungnirun, M. Sadrosadati, O. Mutlu, and M. Daneshtalab, "NoM: Network-on-Memory for Inter-Bank Data Transfer in Highly-Banked Memories," *IEEE Computer Architecture Letters (IEEE CAL)*, 2020.

[68] D. Sanchez and C. Kozyrakis, "ZSim: Fast and Accurate Microarchitectural Simulation of Thousand-Core Systems," in *40th ACM/IEEE Annual International Symposium on Computer Architecture (ISCA)*, 2013.

[69] K. Sangaiah, M. Lui, R. Kuttappa, B. Taskin, and M. Hempstead, "SnackNoC: Processing in the Communication Layer," in *IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2020.

[70] V. Seshadri, Y. Kim, C. Fallin, D. Lee, R. Ausavarungnirun, G. Pekhimenko, Y. Luo, O. Mutlu, P. B. Gibbons, M. A. Kozuch, and T. C. Mowry, "RowClone: Fast and Energy-Efficient In-DRAM Bulk Data Copy and Initialization," *46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2013.

[71] S. Shiina and K. Taura, "Almost Deterministic Work Stealing," in *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2019.

[72] G. Singh, J. Gómez-Luna, G. Mariani, G. F. Oliveira, S. Corda, S. Stuijk, O. Mutlu, and H. Corporaal, "Napel: Near-Memory Computing Application Performance Prediction via Ensemble Learning," in *56th ACM/IEEE Design Automation Conference (DAC)*, 2019.

[73] W. Sun, Z. Li, S. Yin, S. Wei, and L. Liu, "ABC-DIMM : Alleviating the Bottleneck of Communication in DIMM-Based Near-Memory Processing with Inter-DIMM Broadcast," *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, 2021.

[74] N. Talati, A. H. Ali, R. Ben Hur, N. Wald, R. Ronen, P.-E. Gaillardon, and S. Kvatinsky, "Practical Challenges in Delivering the Promises of

Real Processing-in-Memory Machines," in *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2018.

[75] X. Tang, O. Kislal, M. Kandemir, and M. Karakoy, "Data Movement Aware Computation Partitioning," in *50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2017.

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

[77] X. Xie, P. Gu, Y. Ding, D. Niu, H. Zheng, and Y. Xie, "MPU: Towards Bandwidth-Abundant SIMT Processor via Near-Bank Computing," *arXiv preprint arXiv:2103.06653*, 2021.

[78] X. Xie, Z. Liang, P. Gu, A. Basak, L. Deng, L. Liang, X. Hu, and Y. Xie, "SpaceA: Sparse Matrix Vector Multiplication on Processing-in-Memory Accelerator," in *27th IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2021.

[79] T. Yang, J. Gong, H. Zhang, L. Zou, L. Shi, and X. Li, "HeavyGuardian: Separate and Guard Hot Items in Data Streams," in *24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, 2018.

[80] T. Yang, J. Jiang, P. Liu, Q. Huang, J. Gong, Y. Zhou, R. Miao, X. Li, and S. Uhlig, "Elastic Sketch: Adaptive and Fast Network-Wide Measurements," in *Conference of the ACM Special Interest Group on Data Communication (SIGCOMM)*, 2018.

[81] T. Yang, H. Zhang, J. Li, J. Gong, S. Uhlig, S. Chen, and X. Li, "HeavyKeeper: An Accurate Algorithm for Finding Top-K Elephant Flows," *IEEE/ACM Transactions on Networking*, vol. 27, no. 5, 2019.

[82] V. A. Ying, M. C. Jeffrey, and D. Sanchez, "T4: Compiling Sequential Code for Effective Speculative Parallelization in Hardware," in *47th ACM/IEEE Annual International Symposium on Computer Architecture (ISCA)*, 2020.

[83] 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 Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2016.

[84] D. P. Zhang, N. Jayasena, A. Lyashevsky, J. L. Greathouse, L. Xu, and M. Ignatowski, "TOP-PIM: Throughput-Oriented Programmable Processing in Memory," in *23rd International Symposium on High-Performance Parallel and Distributed Computing (HPDC)*, 2014.

[85] 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 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2018.

[86] B. Zhao, X. Li, B. Tian, Z. Mei, and W. Wu, "DHS: Adaptive Memory Layout Organization of Sketch Slots for Fast and Accurate Data Stream Processing," in *27th ACM SIGKDD Conference on Knowledge Discovery & Data Mining*, 2021.

[87] M. Zhou, W. Xu, J. Kang, and T. Rosing, "TransPIM: A Memory-Based Acceleration via Software-Hardware Co-Design for Transformer," in *IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2022.

[88] Z. Zhou, C. Li, X. Wei, X. Wang, and G. Sun, "GNNear: Accelerating Full-Batch Training of Graph Neural Networks with Near-Memory Processing," in *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2023.

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

[90] Y. Zhuo, C. Wang, M. Zhang, R. Wang, D. Niu, Y. Wang, and X. Qian, "GraphQ: Scalable PIM-Based Graph Processing," in *52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2019.

[91] G. K. Zipf, *Human Behavior and the Principle of Least Effort: An Introduction to Human Ecology*. Ravenio Books, 2016.