ABNDP: Co-optimizing Data Access and Load Balance in Near-Data Processing

Boyu Tian
tby20@mails.tsinghua.edu.cn
Tsinghua University
Beijing, China

Qihang Chen
chenqh19@mails.tsinghua.edu.cn
Tsinghua University
Beijing, China

Mingyu Gao
gaomy@tsinghua.edu.cn
Tsinghua University
Beijing, China
Shanghai Qi Zhi Institute
Shanghai, China

ABSTRACT

Near-Data Processing (NDP) has been a promising architectural paradigm to address the memory wall challenge for data-intensive applications. Typical NDP systems based on 3D-stacked memories contain massive parallel processing units, each of which can access its local memory as well as other remote memory regions in the system. In such an architecture, minimizing remote data accesses and achieving computation load balance exhibit a fundamental tradeoff, where existing solutions can only improve one but sacrifice the other. We propose ABNDP, which leverages novel hardware-software co-optimizations to simultaneously alleviate these two issues without making tradeoffs. ABNDP uses a novel and efficient distributed DRAM cache to allow additional data caching locations in the system, where the computation load at the original data hotspots can be distributed and balanced, without significantly increasing remote accesses. ABNDP also adopts a hybrid task scheduling policy that considers both the remote access cost and the load imbalance impact, and exploits the flexibility of the multiple data caching locations to decide the best computation place. Our evaluation shows that ABNDP successfully achieves the two goals of minimizing remote access cost and maintaining load balance, and significantly outperforms the baseline systems in terms of both performance (1.7×) and energy consumption (25%).

CCS CONCEPTS

• **Computer systems organization** → **Parallel architectures**; • **Hardware** → *Emerging architectures.*

KEYWORDS

near-data processing, DRAM caches, task scheduling, load balance

ACM Reference Format:
Boyu Tian, Qihang Chen, and Mingyu Gao. 2023. ABNDP: Co-optimizing Data Access and Load Balance in Near-Data Processing. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3 (ASPLOS '23), March 25–29, 2023, Vancouver, BC, Canada.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3582016.3582026

ASPLOS '23, March 25–29, 2023, Vancouver, BC, Canada
© 2023 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-9918-0/23/03.
https://doi.org/10.1145/3582016.3582026

1 INTRODUCTION

In modern computing systems, data accesses from/to the main memory have been an increasingly critical bottleneck in terms of both performance and energy. From the hardware perspective, the efficiency gap between processors and memories has been steadily growing over decades, known as the "memory wall" problem [64]. From the software perspective, data volume is expanding explosively in the big data era, resulting in many emerging data-intensive workloads in various application domains. Therefore, new architectures should focus on optimizing the memory access cost at least as importantly as accelerating the computation, if not more.

Near-Data Processing (NDP) is one of such new architectural paradigms [6]. By moving the computations closer to the data locations in the main memory, the latency and energy overheads due to data movements can be greatly reduced. We can also better utilize the internal data bandwidth and the short access path inside the memory devices. Currently, NDP has been realized using various memory technologies, including 3D-stacked memories [2, 12, 14, 24, 31, 33–35, 52, 67, 84, 91, 95–97], DDR DIMMs [4, 5, 25, 32, 50, 51, 56, 69, 85, 87], and near the DRAM banks inside a memory chip [1, 38, 57]. Many NDP architectures, especially those based on 3D-stacked memories, leverage a large number of parallel processing units that are distributed across the system. Each unit could access its local memory directly, or go through the system interconnect to access other remote memory regions.

Unfortunately, there is a fundamental tradeoff in such NDP systems between minimizing expensive remote data accesses and achieving load balance across all processing units. With skewed real-world datasets, executing computation tasks closer to their data locations would lead to hotspots with imbalanced loads. In contrast, dynamically scheduling tasks to ensure load balance inevitably moves some computations further away from the preferred locations of their data, incurring more remote accesses. Existing techniques [13, 55, 59, 70, 88] must sacrifice one in order to improve the other, restricting the performance and energy benefits of NDP.

In this paper, we propose ABNDP, an optimized NDP architecture with co-designed hardware and software techniques, in order to *simultaneously* alleviate the aforementioned remote access and load imbalance issues without making tradeoffs. ABNDP is implemented on top of a task-based execution model [45, 46, 78] that best supports the data-centric and parallel processing characteristics of typical NDP applications. The proposed techniques in ABNDP only affect the hardware and the scheduling policy, and incur minimal changes to the user programs.

On the hardware side (Section 4), ABNDP uses a novel distributed DRAM cache design, *Traveller Cache*, which reserves a small portion of the local DRAM at each processing unit to cache data from remote memories. Instead of allowing data to be freely cached everywhere, we restrict the caching of each data block to a limited number of *camp locations* spread across the NDP system. Cached data in a camp location are shared by multiple nearby processing units, improving cache space utilization and data reuse times while still ensuring short access distances. We further use *skewed mappings* for different camp locations, so that cache conflicts can be reduced. Traveller Cache only needs a small amount of metadata that can be stored in SRAM with simple and low-cost management.

On the software side (Section 5), ABNDP adopts a *hybrid task scheduling policy* that considers both the remote access cost and the load imbalance impact to decide the best processing unit for executing a task. A key insight is that the scheduling policy is meticulously *co-designed* with the above DRAM cache. The multiple camp locations to cache data offer higher flexibility when scheduling a task besides the original memory location, which can better spread the heavy computations on hot data for more balanced loads. The skewed camp location mapping scheme also simplifies the scheduling of tasks that access multiple data. It becomes more likely that at least one of the diverse mappings can locate the multiple data more closely, reducing the total remote access distances in a task.

We evaluate ABNDP on a set of widely recognized NDP-friendly applications. We show that ABNDP can successfully achieve the two goals of minimizing remote access cost and maintaining load balance, and realize significant performance and energy advantages. ABNDP achieves on average 1.7× and up to 2.2× performance improvements, and on average 25% and up to 40% energy reduction. These results are much better than previous strategies, which only schedule tasks based on data locations with work stealing, and thus only make one of the data access and load balance issues better and harm the other. The combined benefits of ABNDP hardware-software optimizations are also much higher than the two individual designs, demonstrating the necessity of our co-design approach.

We summarize our main contributions in this paper:

- We illustrate a fundamental tradeoff in large-scale NDP architectures between minimizing remote data accesses and achieving computation load balance, where existing solutions can only improve one but sacrifice the other.
- We present ABNDP, which synergistically uses novel hardware and software solutions to simultaneously alleviate the remote access and load imbalance issues in NDP architectures without making tradeoffs.
- We propose a new distributed DRAM cache design, Traveller Cache, to keep the few frequently accessed data locally at each processing unit. Traveller Cache restricts data caching to limited, deterministic, shared, and skewed camp locations. It realizes high data reuse with small metadata cost and enables better task scheduling.
- We propose a hybrid task scheduling policy to co-optimize remote accesses and load imbalance. The policy is aware of the Traveller Cache design, exploiting the multiple camp locations for flexible scheduling, and leveraging the skewed mapping to better schedule tasks involving multiple data.

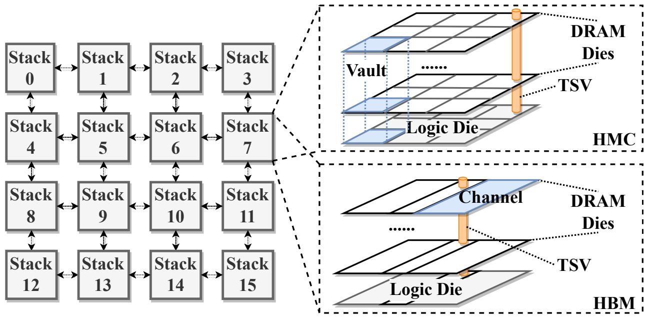


Figure 1: NDP system architecture based on 3D-stacked memory technologies.

- We demonstrate the benefits of ABNDP on widely recognized NDP-friendly applications, with on average 1.7× and up to 2.2× performance improvements, and on average 25% and up to 40% energy reduction.

2 BACKGROUND AND MOTIVATION

2.1 Near-Data Processing (NDP) Architectures

Near-Data Processing (NDP) has been a promising solution to alleviate the memory access bottleneck. The general idea of NDP is to move the computing logic closer to the locations where data are stored, such as near the internal banks of DRAM [1, 38, 57], on the bottom logic die of a 3D-stacked memory [2, 12, 14, 24, 31, 33–35, 52, 67, 84, 91, 95–97], or in the buffer chip on a DDR DIMM [4, 5, 25, 32, 50, 51, 56, 69, 85, 87]. By placing processing logic near data, NDP systems could make efficient use of the abundant bandwidth and the shorter access path internal to the memory structures, and save expensive data movements between the processor and the memory over the narrow and distant memory channels.

In this work, we mainly focus on NDP systems based on 3D-stacked memories like Hybrid Memory Cube (HMC) [65] and High-Bandwidth Memory (HBM) [44, 49], as they offer a balance between technology maturity and performance benefits. As shown in Figure 1, such an NDP system is typically composed of multiple memory *stacks* connected with each other, forming a *memory network* [53, 71, 94]. Each stack consists of several DRAM dies vertically layered on top of a logic die. Through-silicon vias (TSVs) connect different dies with low latency and high bandwidth, e.g., hundreds of GB/s to several TB/s. HMC and HBM use different organizations inside the memory stack. HBM partitions each DRAM die and treats different partitions on different dies as independent memory channels [44, 49]. HMC also divides the DRAM dies, but instead collects the corresponding partitions at the same position on all dies into a vault [65], which is similar to a traditional memory channel and can be independently accessed. The NDP system associates the computing logic to each HBM channel or HMC vault, either by directly putting logic at the bottom die in a true 3D manner, or with 2.5D interposer-based integration. We assume general-purpose, energy-efficient cores are used as the NDP logic [2, 12, 14, 24, 33, 67, 84, 95–97], though other types such as reconfigurable logic [31, 34] and ASIC [35, 52, 91] also fit in our architecture.

We call each memory channel/vault, together with its own computing logic, as an *NDP unit* that operates independently from each other. Each NDP system has multiple stacks, connected through the inter-stack memory network. Each stack contains multiple NDP units, which can also communicate via an intra-stack interconnect. Both levels offer great parallelism. Actually, near-bank and DIMM-based NDP systems also have the similar high-level architecture with many parallel NDP units, but usually lack the direct unit-to-unit communication capability like the memory network in Figure 1. We expect the techniques proposed in this work could also benefit those systems once given more efficient communication support.

2.2 Target Applications of NDP

Previous research has demonstrated the benefits of NDP for a diverse set of parallel data-intensive applications, including MapReduce [33, 73], graph analytics [2, 24, 67, 96, 97], sparse matrix-vector multiplication (SpMV) [91], neural networks (NNs) [35, 52], genome analysis [17, 54], and database [15, 30]. We summarize two key characteristics that make these workloads particularly suitable to NDP.

First, NDP applications usually adopt *data-centric execution*, i.e., certain computation *tasks* are performed on *each individual* data element. For example, in graph analytics, the basic elements are graph vertices and edges. The vertex- and edge-centric paradigms apply operations on every single vertex or edge [63, 76]. In SpMV, each row of the sparse matrix becomes an element, and a task involves the multiplication between each row and the input vector.

Second, NDP applications should also exhibit *high degrees of parallelism*, in order to fully utilize the hundreds of parallel units in an NDP system (Section 2.1). Ideally, each task should mostly process its local data without much communication needed between each other. This requires the tasks on the data to be relatively independent without complex dependencies or fine-grained synchronization. The bulk synchronized execution model is thus widely used in these NDP designs [24, 96, 97].

2.3 Motivation: A Critical Tradeoff in NDP

Nevertheless, despite the more efficient data movements, NDP systems face two critical issues that may limit their overall performance and energy efficiency.

Remote accesses. To map an application to an NDP system, the data of the application need to be partitioned among the NDP units to exploit parallel processing. There are thus two types of memory accesses: *local* accesses to the DRAM in the same NDP unit, and *remote* accesses to other NDP units. The data of a remote access are transferred through the inter/intra-stack interconnects, which incur extra cost depending on the distance. Therefore, the task mapping strategy should maximally co-locate the tasks with their data, i.e., following the "near-data" philosophy. Note that a task may need to access multiple data elements in different units besides the main element it processes; e.g., in Page Rank, a vertex as well as its neighbors is accessed when computing the new rank. We thus must minimize the total distance to all accessed data.

Load imbalance. To fully utilize the massive parallelism in NDP systems, we also need to uniformly assign tasks to all NDP units with balanced loads. This is sometimes quite challenging. At the *algorithm* level, loads of different tasks can be drastically different,

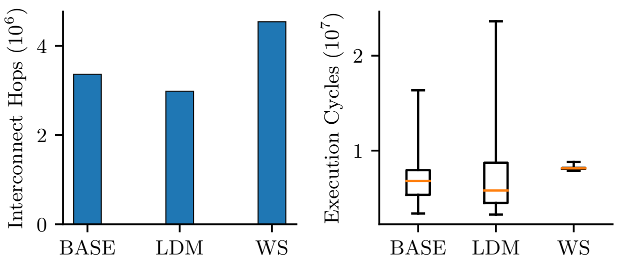


Figure 2: Effects of lowest-distance mapping (LDM) and work-stealing scheduling (WS) on remote accesses (in terms of interconnect hops) and load imbalance (in terms of execution time on all units), when running Page Rank. In the right figure, the box boundaries show the 25%/75% quartiles and the whiskers show the min/max of the execution cycles on all NDP units.

e.g., a Page Rank task for a vertex with more neighbors would involve more memory accesses and more computations. Simply assigning an equal number of tasks to each unit does not work well. At the *hardware* level, the performance of a task also depends on which exact NDP unit it is executing on in the overall interconnect topology. Averagely speaking, the NDP units closer to the center can access most data in the system faster than those units on the corners. Static workload partitioning may analyze the characteristics of the input data, the algorithm, and the system, in order to find the best strategy [26, 55, 59, 70, 88]. A more robust approach is to dynamically schedule the tasks, e.g., with work stealing from the busier units [13, 81], to ensure load balance.

The tradeoff. Unfortunately, though there exist solutions to either of the above two issues, none of previous approaches can address both at the same time. Even worse, the two issues exhibit a fundamental tradeoff; mitigating one problem often results in exacerbating the other, as we show below.

As a reference for comparison, we start with an intuitive and widely used task mapping scheme as the baseline [2, 91, 96]. It evenly distributes all data elements among the NDP units in the system. Each task is assigned to the unit in which its associated element (e.g., the to-be-updated vertex in Page Rank) is stored. We use the NDP system shown in Figure 1, with 16 stacks and 8 NDP units per stack. See Section 6 for the detailed methodology. We count the sum of interconnect hops needed for all data accesses as the metric of remote accesses, and use the execution cycles (not task counts) on different units to reflect load imbalance.

Figure 2 shows the effects of two representative techniques that optimize each issue individually. First, the lowest-distance mapping considers all data accessed in each task and minimizes their total remote distances, therefore the number of interconnect hops reduces compared to the baseline. However, this strategy greatly exacerbates the load imbalance problem, as the busiest NDP unit now slows down by 1.43×, hindering the overall performance. This is due to the different hotness of each data element. In real-world graphs with power-law distributions [37], a small amount of vertices have large numbers of neighbors, and thus are accessed by more tasks than other vertices. If we map tasks solely based on data locations, the NDP units that contain hot vertices would process more tasks than others, creating hotspots and thus imbalance in computation loads. On the other hand, dynamic work-stealing scheduling is able to help with load balancing, and hence the end-to-end performance could improve, which can be seen from the reduction of the maximum execution time in Figure 2. However, scheduling tasks away from their preferred locations to better utilize the otherwise idle NDP units would inevitably introduce more remote accesses to the data, causing higher energy consumption as well as memory access latency overheads.

In summary, there exists a tradeoff between the remote access cost and the load imbalance issue. Existing techniques must sacrifice one aspect in order to optimize the other. In this paper, we aim to propose a combination of hardware and software optimizations to resolve such a conflict, and improve the performance and energy efficiency of NDP systems.

3 SYSTEM OVERVIEW

In this section, we first introduce the task-based programming model and the baseline hardware architecture of our NDP system. Then we provide an overview of our proposed hardware-software co-optimizations in ABNDP to address the conflicting issues of remote accesses and load imbalance.

3.1 Task-Based Model for Programming and Execution

As we have summarized in Section 2.2, NDP-friendly applications usually follow the data-centric execution paradigm with high degrees of parallelism among the computation tasks applied to different data elements. We therefore adopt a *task-based* programming and execution model, which is natural for this data-centric paradigm. For example, in SpMV, each task processes the inner product between one sparse matrix row and the vector, with different rows (tasks) processed in parallel on different NDP units. Using the task abstraction offers several benefits to our NDP system. First, the task-based model allows more flexible scheduling of the computation loads in an application, providing the foundation we need to balance the remote access and load imbalance issues. This is especially important for NDP systems, which consist of hundreds of parallel NDP units. Second, as described below, our task abstraction can easily embed data access information, especially the addresses of all the data elements to be accessed in a task. This information is of particular importance and needs to be made visible to the system runtime when optimizing the remote access cost.

Our task-based model is similar to Swarm [45, 46, 78]. A task has the following attributes: a function pointer, a timestamp to represent dependencies, a hint that encapsulates various information (described later), and any number of additional arguments. Tasks with the same timestamp can run in parallel. Our execution model is bulk synchronized; output data remain unchanged when running tasks with the same timestamp, and all updates are bulk applied at the end after all these tasks complete. A task can dynamically generate any number of child tasks with the `enqueue_task` API:

```
enqueue_task(func_ptr, timestamp, hint, args...)
```

Generated tasks are first scheduled by the scheduler using the hint information, and then dequeued and executed by the corresponding threads running on the NDP units. Algorithm 1 shows an example of a task in the Page Rank algorithm using our task-based model. Each task updates the Page Rank value of a vertex in each iteration. The timestamp represents the current iteration number.

Algorithm 1: Task in Page Rank.

```
1 function TaskPageRank(ts, $v$):
      input: timestamp ts, vertex $v$.
      data: damping factor $\alpha$, total number of vertices $N$,
            convergence threshold $\varepsilon$.
2     var ← 0;
3     for $n$ in $v$.neighbors do
4         var ← var + $n$.currPr/$n$.outDegree;
5     $v$.nextPr ← $\alpha \times$ var + $(1-\alpha)/N$;
6     if $|v$.nextPr − $v$.currPr$| < \varepsilon$ then
7         hint ← /* compose task hint ... */;
8         enqueue_task(TaskPageRank, ts + 1, hint, $v$);
```

Encapsulating data access and computation load information with task hints. To better schedule the tasks in our NDP system, we use *hints* to include the necessary information of a task that will be needed by the scheduler. The first part of a hint is the *data access* information, formatted as a list of single cacheline addresses or address ranges that are accessed by a task. We support two ways to generate such information. First, programmers can insert the accessed addresses manually to a hint as pointers or object references. In the Page Rank example, the addresses to be accessed are the addresses of neighbor vertices of the processing vertex, which can be easily obtained from the vertex neighbor list. Second, we can also enhance the compiler to automatically generate the address list. We provide two pragmas called `#HINT_BEGIN` and `#HINT_END`. The memory accesses in between are treated as the hint addresses of the task.

One thing worth noting is that we limit our hint addresses to the *primary data* of the NDP application, and omit the other *auxiliary data* such as temporary and local variables and data on the thread stack. The primary data are the core part of the application and consistently occupy most of the memory footprint during the execution. They are our main focus for saving data access cost with NDP techniques. In contrast, the auxiliary data are short-lived and only introduce local accesses. It is usually easy to identify the primary data from the program semantics. For example, in graph analytics, the primary data are the graph topology and properties. In SpMV, they are the sparse matrix and the input vector. In recommendation systems, the embedding tables are the primary data.

Another information the hint can optionally cover is the *computation load* of a task, which facilitates estimating the execution time. Programmers can set the value `hint.workload` to reflect the task complexity. If missing, the scheduler will use the total memory access cost of all the primary data addresses in the hint to estimate the load. In either case, the estimation only needs to be approximate (see details in Section 5).

Overheads. There are mainly two aspects of overheads for using a task-based model: the task scheduling cost and the programming burden. Both are acceptable in our case. For scheduling, we only

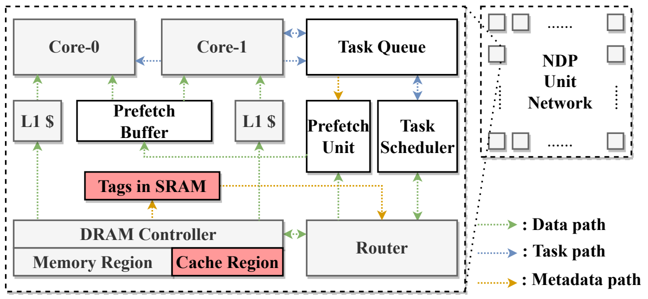


Figure 3: Overall NDP system architecture. The components highlighted in red are newly proposed in ABNDP.

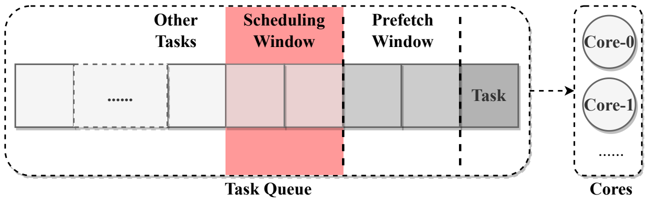


Figure 4: Task management in the task queue. The phase highlighted in red is newly introduced in ABNDP.

add simple logic (a task queue and a scheduler to each NDP unit; see Section 3.2) to do fast scheduling in hardware, in parallel with task computations with minor extra delays. For programming, our model, following Swarm, is simple and widely applicable, especially for the NDP applications mentioned before. Many applications have been implemented using this model in previous work [45, 46, 78].

Another potential concern is the bulk synchronized model. Compared to more fine-grained dataflow task models [22, 23, 86], bulk synchronization will not lead to severe performance loss because load imbalance could be effectively alleviated by our scheduling method (Section 5). On the other hand, it could greatly simplify data coherence in our architecture (Section 4.4).

3.2 Baseline Hardware Architecture

Figure 3 illustrates the hardware architecture in our NDP system. The system is composed of several memory stacks which form a memory network similar to Figure 1. We use a mesh topology for the inter-stack interconnect, with a default scale of 4×4 stacks. Each memory stack consists of 8 NDP units, which are connected by a crossbar-based intra-stack network-on-chip (NoC). There are in total 128 NDP units in the system. We assume 4 GB capacity in each stack, which means 64 GB in total, and 512 MB per NDP unit. Notice that the hardware architecture does not rely on any particular memory technologies and interconnect topologies. Our design can use HMC, HBM, and other 3D memories as long as they exhibit the similar structure. We also focus only on homogeneous NDP units with general-purpose cores. Specialized NDP logic and heterogeneous designs with both cores and accelerators are left as future work, though we believe the key insights would be similar.

We first describe the basic design of the NDP unit in our baseline, and leave the newly proposed components to Section 3.3. As shown in Figure 3, an NDP unit has multiple simple in-order cores (default two [33]), each with its private L1 caches for instructions and data. The cores use the DRAM controller and the router to access data in the local and remote memories. Each core uses a local TLB to translate virtual addresses to physical addresses [33, 41, 68]. To support task-based execution in Section 3.1, the NDP unit also contains the following logic components. A *task queue* maintains all the enqueued tasks. When a core finishes executing its current task, the task queue will dequeue a new task and send it to the core. A *task scheduler* manages the tasks based on the scheduling policy, such as forwarding newly generated tasks to other NDP units, or stealing from others. With the help of the task hints, the scheduler knows the data access addresses and the computation load of each task, facilitating its scheduling decisions. We consider both lowest-distance mapping and work stealing in Section 2.3 in the baseline. We note that there is *no global* task queue or central scheduler; task scheduling is done *locally* at each NDP unit.

Since the accessed data address hint is buffered together with the task in the task queue, we can prefetch these data before the task gets executed on the cores to hide memory access latencies. As shown in Figure 4, we specify a prefetch window in the front of the task queue. A *prefetch unit* issues prefetch requests for the tasks in this window according to their hints, and the prefetched data are stored into an SRAM *prefetch buffer*. We do not prefetch into the small L1 caches to avoid interference and pollution. Hits in the prefetch buffer also bypass the L1 caches. Similar to the message-triggered prefetching in Tesseract [2], our prefetching is exact, but based on task hints instead of remote functions calls.

The baseline architecture is highly optimized to support efficient task-based execution. Nevertheless, it still suffers from the issues discussed in Section 2.3, with the tradeoff between remote accesses and load imbalance.

3.3 ABNDP Summary

We propose ABNDP, which is built on top of the previous baseline design with additional hardware and software optimizations that *simultaneously* alleviate the remote access and load imbalance issues without compromising one for the other. On the hardware side, ABNDP adopts a novel distributed DRAM cache design, *Traveller Cache*, to additionally cache data in other NDP units besides the original memory locations. On the software side, ABNDP uses a *hybrid scheduling policy* that balances the effects of memory accesses and computation loads, in order to decide the best NDP unit to execute a task.

More specifically, as in Figure 3, ABNDP reserves a small portion (e.g., 1/64) of the local DRAM in each NDP unit to be used as a cache space (Section 4). This allows the remote data from other NDP units to be stored closer to where the task will be executed, reducing the remote access overheads. Furthermore, having multiple places across the system for each data element allows for more flexible scheduling, opening up hardware/software co-design opportunities. We further propose a camp location design to optimize the cache efficiency, using simple and low-cost metadata schemes.

In addition, ABNDP uses a more effective task scheduling policy that takes both the data access distances and the computation loads into consideration (Section 5). As shown in Figure 4, we further specify a scheduling window in the task queue. The task scheduler, which operates in parallel with the computing cores, examines the tasks in the scheduling window, and makes decisions about where to execute them. The scheduler considers both the data access and computation load information encapsulated in the task hint (Section 3.1) to balance the two factors. The scheduling policy also works compatibly with the Traveller Cache with combined benefits.

The changes in ABNDP are limited to the hardware and the runtime software, and largely transparent to the user program (Algorithm 1). The user may need to optionally supply more detailed hint information, but this is not required as ABNDP can make its own estimation.

4 TRAVELLER CACHE

At the hardware level of ABNDP, we propose a novel distributed DRAM cache scheme called *Traveller Cache*. Using a small portion of the DRAM in each NDP unit as a cache effectively reduces the remote data access cost, and also allows for more flexibility to schedule tasks near one of the multiple data locations. We choose DRAM-based caching in ABNDP because the stringent area limit of 3D-stacked memory prohibits us from realizing large SRAM caches.

4.1 Design Challenges

There have been numerous studies about DRAM caches in conventional processor-centric architectures, which use faster on-die or in-package DRAM as caches of traditional slower DDR and non-volatile memories [21, 39, 42, 47, 58, 61, 62, 66, 74, 83, 93]. Nevertheless, applying DRAM-based caching to NDP systems is significantly different. Particularly, we use local DRAM to reduce data accesses to other remote memory regions with all memories in the same system level that is 3D stacked, while conventional DRAM caches have two separate tiers of fast and slow technologies.

As a result, we must deal with several unique challenges. Specifically, the capacity of 3D-stacked memory is limited. Each NDP unit only has 512 MB local DRAM, among which the DRAM cache space can only occupy a small portion. On the other hand, typical NDP systems contain hundreds of NDP units (e.g., 128 in our system). Using a small fraction of one unit space to cache data from so many units requires efficient use of the cache space. This is further complicated by the fact that each task may access multiple data elements (e.g., neighbor vertices and edges), all of which prefer to stay in the limited local DRAM cache. However, NDP applications usually exhibit relatively low data locality, where each data element may not be reused as much as in conventional processor-side DRAM caches. Finally, we also need an efficient metadata scheme in such a distributed cache design, both to accurately track the locally cached data and to be aware (or not aware) of the states of other caches.

4.2 Camp Locations

In Traveller Cache, instead of allowing a data block to be freely cached in any NDP unit, we restrict a limited number of locations where a data block can stay. These places are called its *camp locations*, and the original memory location is called its *home*. The number of camp locations of each data block, denoted by C, is a hyperparameter in ABNDP. We spread the camp locations of each data block across the memory network, so every NDP unit can have

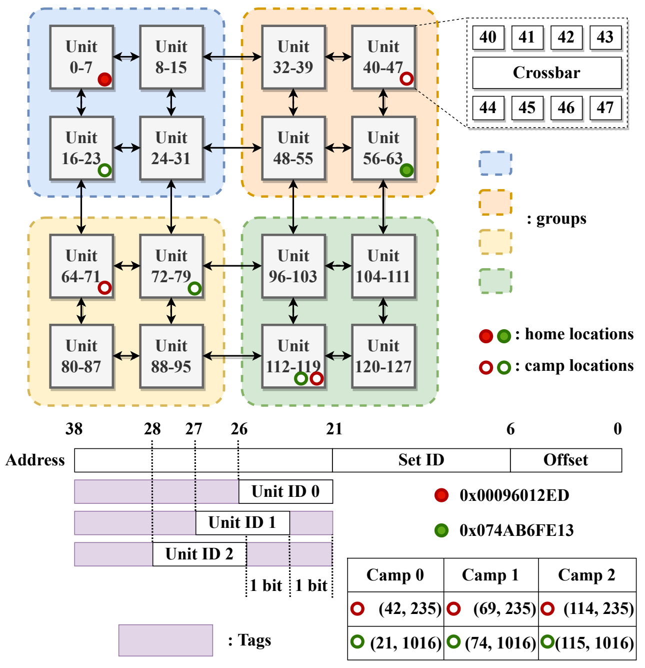


Figure 5: Camp location mapping in the mesh topology. All NDP units are divided into four groups. The camp locations of the two data elements (red and green) are designated in different ways based on their address hashing results.

a nearby camp location. The NDP units around a camp location will all use this single cached data copy instead of independently and repeatedly caching it. Sharing the camp locations in this way improves the cache space utilization as well as the reuse times of the cached data. The access distance only increases slightly compared to local caching, as long as the camp locations are well distributed.

Specifically, we use the cacheline granularity to designate camp locations. Each camp location of a cacheline-sized data block is in the form of {unit ID, set ID}, specifying the NDP unit that is allowed to cache the data, and which cache set in that unit the block is mapped to. Both are computed deterministically from the (physical) address. The cache set mapping follows traditional caches, using the lower bits in the address as the set ID (Figure 5 bottom).

On the other hand, the unit ID mapping scheme is more involving. To uniformly distribute the camp locations across the whole system, we divide all the NDP units into $C + 1$ localized groups (number of camp locations + one home) according to their mutual distances in the interconnect topology. We number the units consecutively, first in each stack, then in each group, and finally across groups. Figure 5 top illustrates an example of the inter-stack mesh with $C = 3$. Other topologies can be similarly processed by grouping and numbering the NDP units in a similar way. In each group, each data block has either one camp location or the home location, from which all the units in the group can get the data.

Rather than treating all the groups identically, we intentionally assign the unit ID of the camp location in each group in different ways, by using different bit slices in the remained address bits (Figure 5 bottom). This is essentially similar to skewed associative caches [79], where each group uses a different mapping for the same data address. Such skewed mapping for the unit ID has the following benefits. First, as in skewed associative caches, making each group use a different mapping reduces cache conflicts. Even though in one group the critical data elements are mapped to the same NDP unit and compete for its limited cache space, in another group they are likely mapped differently and spread to different units without conflicts. Second, for the multiple data elements accessed in one task, the skewed mapping allows them to be cached more closely in at least one of the groups, so we can find a place to execute the task near most of the data. For example in Figure 5, assume a task needs both the red and green data elements. Their home locations are distant. Even with caching, their camp locations in a group may not be close enough (e.g., the bottom left group). But since we have multiple groups, their minimum distance across all the groups is likely to be small (e.g., the bottom right group). In this example, we can use any NDP unit between 112 and 119 to process this task with minimum total remote accesses. Section 5 will discuss how our scheduling policy could make such an optimized decision.

Benefits. We complete this part by summarizing the benefits of the camp location design. First, it makes more efficient use of cached data both locally and in neighbors, reducing duplicated caching. This is important for NDP because of the limited 3D memory capacity. Second, the more determinism in the mapping of unit/set IDs saves metadata lookup and storage overheads (Section 4.3). Otherwise, freely caching requires larger metadata storage in each NDP unit and more times of DRAM cache probing. Finally, the skewed camp location mapping reduces conflicts, and also offers flexible scheduling opportunities to find better places to execute tasks involving accesses to multiple different data.

4.3 Metadata Schemes

Local cache tags. The DRAM cache space in each NDP unit needs an efficient metadata scheme to track the tags for all of its locally cached data blocks. There are several commonly used approaches. Using SRAM to store the cache tags [42] is the simplest and has the best access latency, but the size is strictly limited by the tight area constraint of 2.5D/3D integration. Placing the tags in DRAM with the data [47, 48] avoids this limit, but incurs extra latency and energy overheads, especially for data misses. Techniques to predict hit/miss status and data locations can be used [62, 74, 93]. But in our distributed DRAM cache, we must predict both which of the many NDP units has the data and the data location in that unit. It is difficult to make sufficiently accurate predictions, especially for low-locality NDP applications.

In the Traveller Cache design, we choose to use SRAM to store the cache tags. This method is simple, fully accurate, and has the lowest extra latency and energy cost. Our camp location design further reduces the storage cost to be practical to fit in the limited 3D-stacked die area, since *it restricts the number of data blocks that can be cached at a certain location.*

To show this, we calculate the tag storage size with the default system configuration in Section 3.2, i.e., 128 NDP units with 64 GB capacity in total. We assume $1/R = 1/64$ of the 512 MB in each NDP unit is used as the DRAM cache, whose associativity is $A = 4$ (Section 4.4). So there are $N_{\text{set}} = 512\,\text{MB}/64\,\text{B}/R/A = 32768$ cache sets per unit. First, without the camp location restriction, all the address bits except for the 6-bit block offset (for 64-byte cachelines) and the 15-bit set ID (for 32768 sets) must be stored as the tag, which is $\log(64\,\text{GB}) - 6 - 15 = 15$ bits. When using the camp location mapping, the unit ID bits can be taken out of the tag, as it is known when probing a location in a specific unit. Since we have $128/C = 32$ NDP units in each group, this enables us to save 5 bits and only use 10 bits for each tag, corresponding to a 1.5× reduction. The total tag size for the $N_{\text{set}} \times A = 128\,\text{k}$ cache blocks in one NDP unit is 160 kB, which is an acceptable SRAM size comparable to the total capacity of the two cores' L1 caches.

Our tag scheme has nice scalability. An important observation is that, *when we keep C unchanged and increase the number of NDP units by using more memory stacks, the tag size in each unit keeps constant.* Even if we also proportionally increase the number of groups, the tag size only grows logarithmically. The total tag size does scale linearly with the cache ratio R and the per-unit memory capacity, both of which are unlikely to increase drastically even with future technologies.

Information of remote caches. In Traveller Cache, each NDP unit only tracks its locally cached data, without knowing any information about whether a data block is cached in another unit or not. Tracking the states of the many remote caches not only requires significant metadata storage in each unit, but also incurs metadata traffic among the units to keep the states up-to-date. Both are too expensive. Instead, when an NDP unit wants to access a data block, we make it *always probe only the nearest camp location.* On one hand, compared to the baseline where the access request directly goes to the home memory location, the extra cost in terms of interconnect hops is small, because the camp location is close to the requesting NDP unit. On the other hand, it is usually unnecessary to probe other distant camp locations, since they may not be much closer than the home memory location.

4.4 Cache Organization and Policies

Cache organization. We use a default 4-way set-associative scheme for the DRAM cache in each NDP unit. Higher associativities increase the tag size (by reducing the number of sets). Furthermore, since the memory accesses can be distributed among the many NDP units, each unit does not need to have very high associativities. We empirically evaluate the impact of the associativity in Section 7.2.

Replacement policy. We adopt a probabilistic insertion policy to insert cachelines into the DRAM cache [10]. In another word, each data block has a certain probability of bypassing the DRAM cache, which is empirically set to 40% in our system (Section 7.2). This is motivated by the power-law data distribution in most of our target applications [37], where most cache hits are to a small subset of the overall dataset. Using a probabilistic insertion policy filters out most of the low-reuse data, while the frequently accessed data will eventually be cached after a few trials.

Because we use SRAM-based tags, our DRAM cache can easily adopt any replacement policy without complex metadata management. Nevertheless, we find that there is little performance difference between an LRU and a random policy. The small amount of critical data can be stably cached regardless of the policy choice. Therefore, to avoid extra metadata, we use random replacement.

Cache coherence. As described in Section 3.1, the task-based execution model uses bulk synchronization, where the primary data remain unchanged within each timestamp, and we atomically switch to the newly updated data at the end of each timestamp. Our Traveller Cache only stores these read-only primary data, therefore greatly simplifying the coherence requirement. Data writes bypass the DRAM cache and directly go to the home memory locations. All cached data are bulk invalidated, e.g., by clearing the SRAM-based tags, at the end of each timestamp. No writeback is needed.

Overall access flow. We summarize the end-to-end operation flow of Traveller Cache. When a prefetch request from the prefetch unit is issued (Section 3.2) or an L1 cache miss occurs in an NDP unit, the data access request first traverses to the nearest camp location (a small detour), where it efficiently checks the SRAM tags to see whether the data block is cached. If yes, it is returned to the requesting NDP unit and put into the prefetch buffer or the cache. Otherwise, i.e., a DRAM cache miss, the request continues to the home location to obtain the data. Besides returning the data block to the requesting NDP unit, we also try to insert it to the previously probed camp location, subject to the probabilistic insertion policy.

5 HYBRID TASK SCHEDULING POLICY

To effectively balance the remote access and load imbalance issues, ABNDP uses a *hybrid task scheduling policy*, which incorporates both factors into a score-based scheduling mechanism. As shown in Figure 4, each NDP unit actively makes scheduling decisions for the tasks in the scheduling window of its task queue, by scoring each NDP unit in the system to reflect how efficient it is if using this unit to execute the task.

5.1 Impact of Traveller Cache

The scheduling policy is meticulously co-designed with our hardware Traveller Cache. Essentially, the Traveller Cache design offers several desired features to facilitate better scheduling. First, there are now multiple camp locations where a data block can be cached besides the original home memory location, providing high scheduling flexibility. This is especially important for our hybrid policy. Specifically, for a hot data element that originally causes a task execution hotspot around its home memory location, we can now dispatch some of its tasks to the other camp locations to improve load balance, while still ensuring the task execution is close to the (cached) data. Second, the skewed camp location mapping scheme (Section 4.2) can help those tasks with multiple accessed data. With diverse mappings among all the groups, there will likely be at least a group in which the multiple data elements are sufficiently close to each other, and we can find a proper NDP unit to execute the task at that place.

5.2 Scheduling Algorithm

We now describe the detailed scheduling algorithm. At the high level, given a task t in the scheduling window of the task queue, the scheduler computes a score for each NDP unit u in the system. The unit with the lowest score will be the target for executing this task. The estimated score is a weighted sum of two parts: one for the remote memory access cost_{mem}, and the other for the load imbalance $\text{cost}_{\text{load}}$. Specifically,

$$u_{\text{target}} = \underset{\forall u}{\text{argmin}} \{\text{cost}_{\text{mem}}(t, u) + B \times \text{cost}_{\text{load}}(t, u)\} \tag{1}$$

where B is the hybrid scheduling weight that combines the two cost terms. We discuss it later.

Remote memory access cost. To calculate cost_{mem}, we simply use the average distance from the target NDP unit to all the data elements accessed by the task. Here we heuristically use the camp location (or the home) of the data element nearest to the target unit (not the scheduling unit) as the data location. Recall that we cannot know the contents of remote caches (Section 4.3), so we cannot accurately know whether a data block is cached or not. This estimation is acceptable for our main focus of high-reuse data. If we mispredict and it is a DRAM cache miss, the data will be brought into the cache so in the future our cost calculation becomes correct.

We distinguish the data transfers within the local unit, to a different unit in the same stack, and to another stack, each with cost D_{local}, D_{intra}, D_{inter}. These numbers are directly set to the relative hardware costs. They do not need any hyperparameter tuning. In our specific architecture, because we use a crossbar for the intra-stack NoC, D_{intra} is constant. In contrast, D_{inter} denotes the cost of one hop in our inter-stack mesh.

$$\text{cost}_{\text{mem}}(t, u) = \frac{1}{|t.\texttt{hint.data}|} \times \sum_{a \in t.\texttt{hint.data}} \begin{cases} D_{\text{local}} & a \text{ is local to } u \\ D_{\text{intra}} & a \text{ is intra-stack to } u \\ D_{\text{inter}} \times N_{\text{hops}} & a \text{ is inter-stack to } u \end{cases} \tag{2}$$

Notice that by considering all NDP units in all groups, the above cost model automatically leverages the skewed mappings in different groups, and is able to find the one with the minimum total distance.

Load imbalance cost. To minimize load imbalance, we assign lower (better) scores for more idle NDP units. Specifically, the idleness of a target unit is captured as

$$\text{cost}_{\text{load}}(t, u) = W_u / \overline{W} - 1 \tag{3}$$

where W_u is the sum of the workloads of all *future* tasks in the local task queue of unit u, and $\overline{W}$ is the average over all units. Each NDP unit maintains its own W_u by incrementing it by $t.\texttt{hint.workload}$ when a task is enqueued, and decrementing when dequeued. Because the scheduler also needs the W_u values of other units, we let all units periodically exchange their workload information. The communication happens hierarchically [96], where the units in a stack first communicate to collect all W_u values internal to the stack, and then only one unit sends the collection to other stacks. The workload information exchange only needs to be infrequent, with a large time interval, much longer than the task execution time. For example, we set it to 100,000 cycles, during which each unit can typically execute over thousands of tasks (Section 7.2). The exchange also happens in the background without blocking the normal task execution, since the workload information does not need to be accurate for execution correctness.

Hybrid scheduling weight B**.** Finally, we discuss how to select the hybrid scheduling weight, which is the only hyperparameter in our scheduling algorithm. A larger B weighs more on the load

Table 1: System configurations.

NDP system	4×4 stacks in mesh, 8 NDP units per stack; 64 GB in total, 512 MB per unit
NDP core	2 GHz, 2 cores per NDP unit (256 in total)
L1-D cache	64 kB, 4-way, 64 B cachelines, LRU
L1-I cache	32 kB, 2-way, 64 B cachelines, LRU
Prefetch buffer	4 kB, 64 B blocks, FIFO
DRAM channel	128 bits; $t_{\text{CAS}} = t_{\text{RCD}} = t_{\text{RP}} = 17$ ns; 5.0 pJ/bit RD/WR, 535.8 pJ ACT/PRE
Intra-stack net	128-bit link; 1.5 ns/hop; 0.4 pJ/bit
Inter-stack net	32 GB/s per direction; 10 ns/hop; 4 pJ/bit
Traveller Cache	$1/R = 1/64$ of local mem. capacity, 4-way; $C = 3$ camp loc.; random repl., 40% bypass
Scheduler	100,000-cycle workload exchange interval; hybrid scheduling weight $B = 3D_{\text{inter}}$

balance goal to get higher performance, while a smaller B mainly optimizes the remote memory accesses and saves energy consumption. Intuitively, B means how far the highly idle target unit can be away from the data location. For example, assume a totally idle unit u with $W_u = 0$. Its load imbalance cost $\text{cost}_{\text{load}} = -1$. This results in a reduction of B on the remote access term cost_{mem}. If its distance to the data is within this gap compared to the best unit with the shortest data distance, then its total score will be better and it will be selected as the target NDP unit. Therefore, we set $B = D_{\text{inter}} \times \frac{1}{2}d$ where d is the diameter of our inter-stack network (e.g., 6 for the 4×4 mesh), indicating that we allow an idle unit to overcome half of the maximum hop distance in the system. We also empirically evaluate its impact in Section 7.2.

6 METHODOLOGY

System models. We use zsim [77], a fast and accurate Pin-based simulator to model our NDP system. We extend zsim with the hierarchical interconnection, and support task-based execution similar to Swarm [92]. Table 1 summarizes the key configurations of our system. We model the NDP logic as simple in-order cores at 2 GHz, consuming 163 µW at idle and 371 pJ per instruction [89]. We use CACTI 7 [7] to model the timing and energy of SRAM caches, prefetch buffers, and the tag storage for Traveller Cache. The DRAM channel in each NDP unit follows the HBM characteristics [18]. The intra-stack NoC and intra-stack link models are from [80].

Workloads. We evaluate eight data-intensive parallel applications: BFS (`bfs`), single-source shortest path (`sssp`), Page Rank (`pr`), graph convolutional neural network (`gcn`), A* search (`astar`), K-nearest neighbors (`knn`), sparse matrix-vector multiplication (`spmv`), and K-means (`kmeans`). We port the workloads into our task-based model, similar to existing parallel benchmark suites [23, 63, 82, 91]. We use real-world graphs [60] and datasets [27] for the graph applications and `spmv`, and construct synthetic datasets as the input for `kmeans` and `knn`. We manually add the data access hint in each task, but leave the workload hint unspecified, so our scheduler makes estimation as in Section 3.1.

Baseline designs. We compare ABNDP with five baseline NDP designs, as summarized in Table 2. The baseline **B** assigns a task

Table 2: Evaluated system designs.

Design	Task scheduling	DRAM caches
H	Use host CPU only	
B	Co-locating with one data element	No
Sm	Lowest-distance	No
Sl	Lowest-distance + work-stealing	No
Sh	Hybrid (**ours**)	No
C	Lowest-distance	Yes (**ours**)
O	Hybrid (**ours**)	Yes (**ours**)

to the location of its main data element. **Sm** (**m** for "memory") is the lowest-distance mapping in Section 2.3 considering all data elements in a task, and **Sl** (**l** for "load") further adds work-stealing to balance load. **Sh** uses our hybrid scheduling in Section 5, but without DRAM caching. **C** uses our Traveller Cache in Section 4, but only with basic lowest-distance task mapping. Finally, **O** combines both optimizations in ABNDP. Notice that all the configurations are based on the task-based architecture in Section 3.2 and already support data prefetching to the SRAM buffers. Besides the above NDP architectures, we also include a non-NDP system **H** that only uses the host CPU to execute the same task-based benchmarks. The host CPU has 16 out-of-order cores at 2.6 GHz, a 20 MB last-level cache, and 4 channels of DDR4-2400 memory.

7 EXPERIMENTAL RESULTS

We begin by presenting the overall performance and energy comparison results in Section 7.1, and then show the detailed studies of each design choice in Section 7.2.

7.1 Overall Comparison

Figure 6 shows the overall performance comparison of different system designs. We further illustrate the amount of remote accesses in terms of the total number of inter-stack mesh hops in Figure 8, and the workload distribution across all NDP units in terms of their active processing cycles in Figure 9, for selective representative workloads in our benchmark suite.

By considering all data locations in a task, **Sm** reduces the total remote accesses by 7% over **B** on average (Figure 8), but it actually results in 14% average performance degradation. This is because the load imbalance issue (Figure 9) gets exacerbated due to skewed data hotness, similar to the results in Figure 2. Adding work-stealing in **Sl** helps alleviate load imbalance, and thus achieves an average 14% performance improvement over **B**. But it also causes up to 2× remote accesses (Figure 8), as some tasks are scheduled to remote locations, limiting the benefits. These results of **Sm** and **Sl** clearly illustrate the tradeoff we elaborate before. Our hybrid task scheduling policy **Sh** is able to balance the two problems, with fewer remote accesses than **Sl** and better load balance than **Sm**, and achieves higher speedups than both, 23% on average. This is because its score function is a weighted sum of the two aspects, and hence neither a scheduling choice that incurs too many remote accesses nor the one with significant load imbalance could result in the best score. Nevertheless, it still leads to 45% more remote accesses than the baseline.

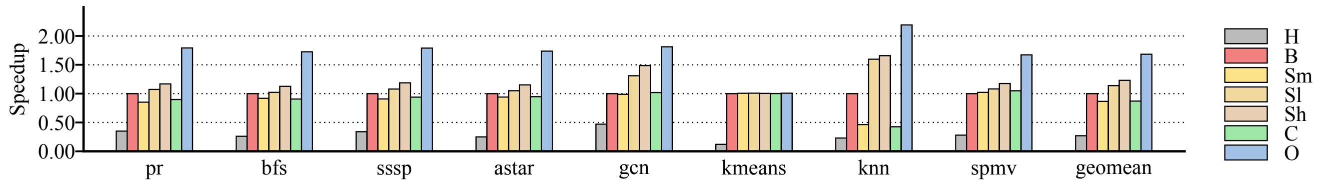


Figure 6: Overall performance comparison between ABNDP and the baselines. Normalized to B.

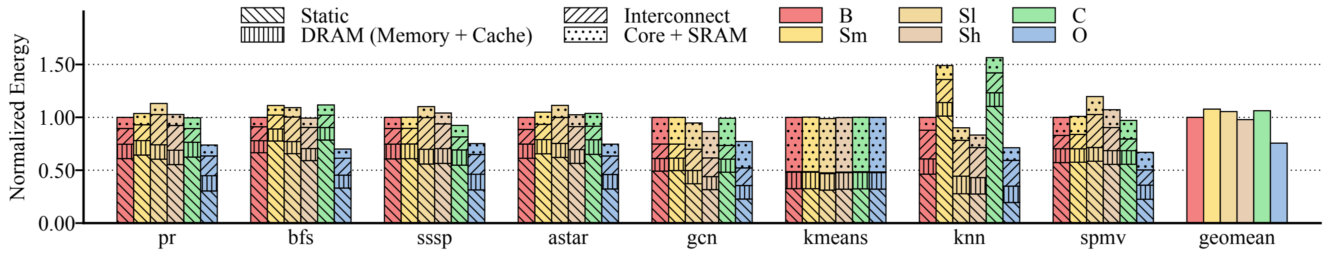


Figure 7: Overall energy comparison between ABNDP and the baselines. Normalized to B.

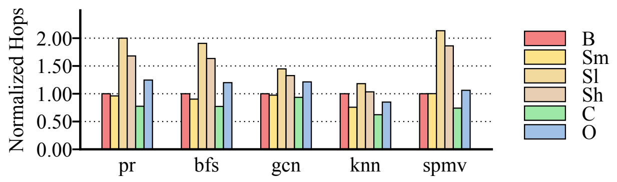


Figure 8: Remote accesses in terms of total inter-stack hops.

On the other hand, if we use Traveller Cache in **C** without any scheduling optimization, the remote accesses can be reduced by 21% over **B**. Traveller Cache could store the critical data elements in the camp locations closer to the computing units, therefore reducing the data transfer distances. However, because the basic scheduling policy does not take the computation loads into consideration, the overall performance still suffers from severe load imbalance and sometimes even degrades.

Finally, ABNDP (**O**) combines the benefits of Traveller Cache and the hybrid task scheduling, and leads to on average 1.68× and up to 2.19× performance improvements. Its load balance behavior is close to the dynamic work-stealing **Sl** design. The amount of remote accesses is also significantly less than both **Sl** and **Sh**, and only slightly higher than **C**. We remark that the combined performance benefit in ABNDP is much higher than the two individual techniques **Sh** and **C**. This is because the two techniques work in tandem with each other as a direct result of our hardware-software co-optimization. Traveller Cache offers more high-quality choices to the hybrid scheduling policy, including the scenarios where the hotspot loads on some data elements are more evenly distributed to the multiple camp locations besides the home, and where the camp locations of the multiple accessed data elements in a task are sufficiently close to each other.

Among the workloads, `kmeans` exhibits little difference across system designs, because its tasks are fully independent and process separate local data. There are no remote access or load imbalance issues to begin with. For `knn`, however, because of the skewed distribution in our synthetic dataset, the workload is highly imbalanced. Designs without load balancing considerations (**Sm** and **C**) perform substantially worse. There are also significant remote accesses from the KD-tree traversal and the linear search in the leaf nodes. So using Traveller Cache has a large impact in Figure 8.

To be complete, we also compare the performance of our NDP systems against a host-only design **H** where the same task-based applications are executing on a typical server-class CPU. The baseline NDP design **B** is 3.70× faster than **H**. These relatively limited performance gains are mostly due to the irregular applications we use, which cannot fully utilize the NDP benefits due to load imbalance and remote access overheads. By effectively alleviating exactly these bottlenecks, ABNDP magnifies the speedup to 6.29× over **H**.

Energy. The energy consumption of each design is shown in Figure 7, which breaks down into four components, for 1) the NDP cores and SRAM caches, 2) the DRAM cache and memory accesses, 3) the interconnect transfers, and 4) the static energy. The interconnect energy highly correlates to the remote access hops in Figure 8. The DRAM energy increases in designs that use Traveller Cache because of extra DRAM cache insertions. But this overhead is well compensated by the reduction in the interconnect energy. The static energy follows the performance trend. Overall, ABNDP consumes the minimum energy over all designs, with on average 24.6% and up to 40.1% reduction compared to the baseline.

Scalability. Figure 10 evaluates the scalability of ABNDP by using 2×2, 4×4 (default), and 8×8 stacks, corresponding to 32, 128, and 512 NDP units (two cores per unit). We keep the number of groups unchanged ($C = 3$). As discussed in Section 4.3, the required tag storage in SRAM keeps the same. As the system scale

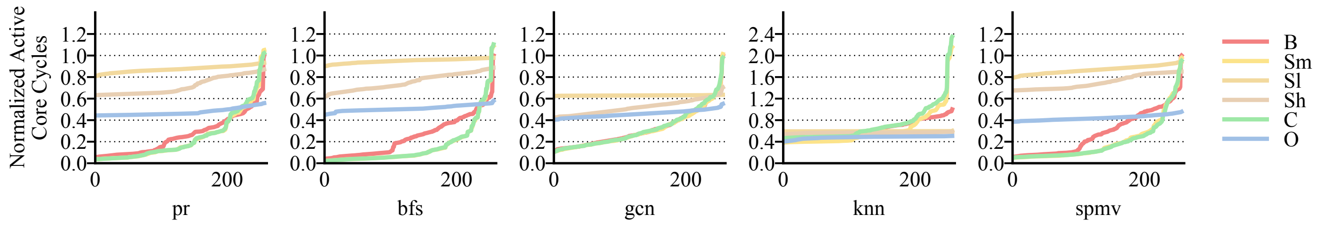
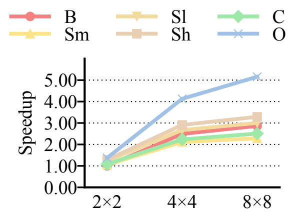
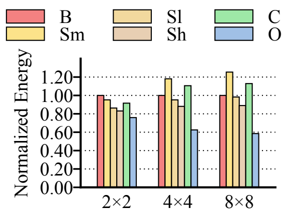


Figure 9: Workload distribution in terms of active cycles on all NDP cores. The horizontal axis is all the NDP cores, sorted by their numbers of active cycles in the ascending order, independently in each design. For gcn, the line of B is almost invisible because it overlaps with the line of Sm.



(a) Speedup.



(b) Energy.

Figure 10: Scalability comparison when running Page Rank.

grows, remote accesses become more expensive due to larger inter-stack networks, and load imbalance also gets more serious due to more processing units. The optimizations in ABNDP therefore become more critical. We can see that both the speedup and energy reduction of **O** compared to **B** get larger when the system scales up. The other NDP baselines scale significantly worse than **O**. **Sm** and **C** reduce the remote access cost, but may further aggravate load imbalance, making their scalability even worse than **B**. **Sl** and **Sh** alleviate the load imbalance issue in the system. Nevertheless, they only show slightly better scalability than **B**. This is because of the increasing remote access cost due to larger interconnect diameters, which restricts more NDP units from working together efficiently.

In addition, the ABNDP caching and scheduling techniques themselves have good scalability. Traveller Cache uses efficient logarithmic-size metadata to locate data in the distributed environment. The scheduling decisions are made within each NDP unit, requiring no centralized control.

We also note that for ABNDP of even larger scales, remote data accesses remain to be a critical bottleneck. As in Figure 10a, using 8×8 stacks only leads to a less than 15% performance gain compared to 4 × 4 stacks. This is also probably why most of the previous designs constrained their NDP systems to 16 stacks [2, 24, 33, 96] or communicated asynchronously [97] to hide long remote access latencies in large systems.

7.2 Design Choice Studies

Camp locations. We first evaluate the benefits of using our skewed camp location mapping across different groups (Section 4.2). The alternative identical mapping uses the same address bits to generate

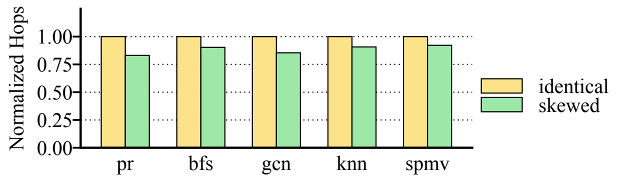


Figure 11: Remote access hop comparison when using skewed vs. identical camp location mappings.

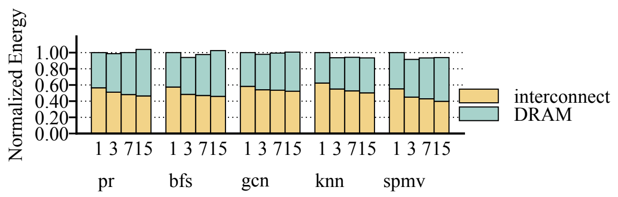
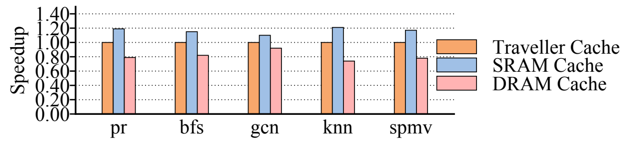


Figure 12: Impact of the camp location number C.

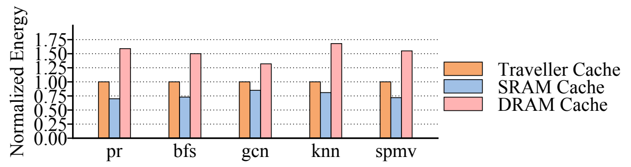
the camp location unit IDs in all groups. Figure 11 shows that the skewed mapping saves 12% remote access hops on average, due to both cache conflict reduction and smaller total distances to multiple data blocks in each task.

Figure 12 shows the DRAM and interconnect energy changes when using different camp location numbers C. Recall that the number of groups is $C + 1$. The impact is minor. With more camp locations, we can cache more data and reduce the interconnect energy. But also there are more DRAM cache insertions that increase the DRAM energy. Overall, the combined effect results in small differences among different C values, and $C = 3$ is a good choice.

Traveller Cache configurations. Traveller Cache is a DRAM data cache while its cache tags are stored in SRAM (Section 4.3). We justify this design by comparing our Traveller Cache with 1) a pure on-chip SRAM data cache, and 2) a pure DRAM cache with tags in DRAM [47, 48]. Figure 13 shows the performance and dynamic energy results when using the three cache schemes, with the same data capacity of 8 MB as our default configuration. The SRAM cache provides a 15% speedup and 23% energy saving than our Traveller Cache on average. However, the 8 MB SRAM requires an unrealistic amount of 16.12 mm^2 logic die area *per unit*. On the other hand,



(a) Speedup.



(b) Dynamic DRAM energy.

Figure 13: Comparison between our Traveller Cache, a pure SRAM data cache, and a DRAM data cache with in-DRAM tags.

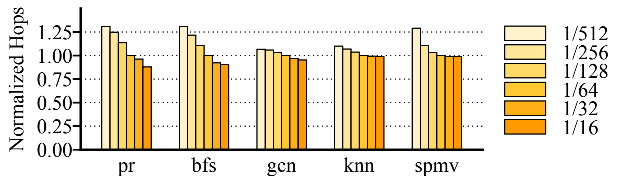


Figure 14: Impact of the Traveller Cache capacity.

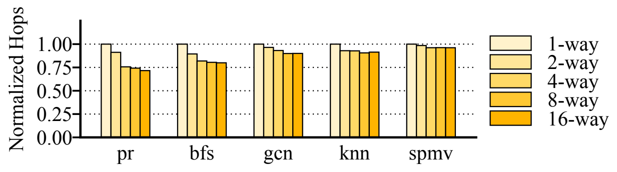


Figure 15: Impact of the Traveller Cache associativity.

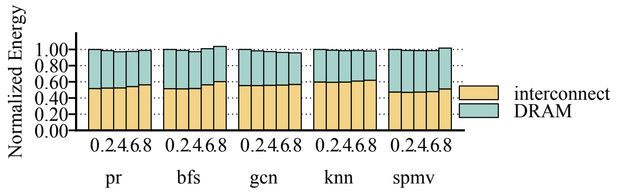


Figure 16: Impact of the Traveller Cache bypass probability.

the tag access overhead of the pure DRAM cache causes a 21% slowdown and 54% more energy on average. The Traveller Cache design only requires 0.32 mm^2 in each NDP unit, and does not suffer from any in-DRAM tag access overhead.

Following Section 4.4, Figure 14 shows the impact on remote accesses when using different DRAM cache capacities, from 1/512 to 1/16 of the local DRAM space. Larger cache sizes keep more data and reduce more remote accesses, but also need more metadata tags. Figure 15 sweeps the associativity, where a 4-way configuration is sufficiently good. Finally, Figure 16 evaluates the bypass probability when inserting into the cache. More bypasses reduce the DRAM energy, but slightly increase the interconnect hops. Overall the design is insensitive to this probability and 40% is a good balance.

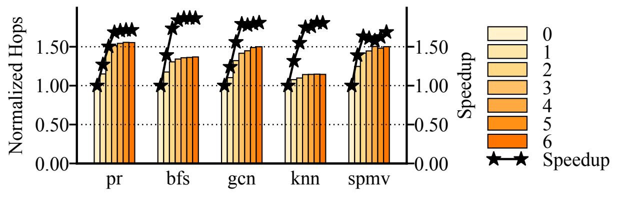


Figure 17: Impact of the hybrid scheduling weight B with different coefficients from 0 to the topology diameter $d = 6$.

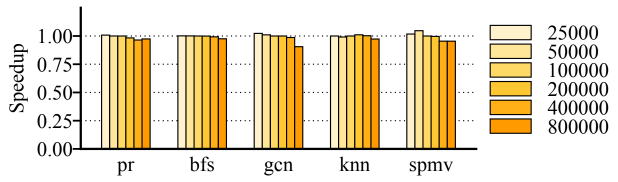


Figure 18: Impact of the workload exchanging interval. Numbers are shown in cycles.

Scheduling policy. The hybrid scheduling policy has a hyper-parameter B as the weight to combine the two factors. Figure 17 evaluates its impact by setting $B = D_{\text{inter}} \times \alpha$, where by default $\alpha = \frac{1}{2}d = 3$ is half of the topology diameter. We see that the number of interconnect hops increases as α gets larger, while the performance saturates around $\alpha = 3$. Finally, Figure 18 shows that the workload exchange interval can be made quite large without affecting performance. Thus there is little extra cost.

8 RELATED WORK

Near-Data Processing. There have been numerous prior studies in the NDP paradigm, which can be categorized into domain-specific accelerators and general-purpose systems. Many prior proposals focused on specific data-intensive applications and designed NDP-based accelerators [2, 17, 24, 35, 52, 54, 91, 96, 97]. Others tried to retain generality and aimed to ease the adoption of NDP, including programming models [3], system support [15, 16, 36], identification of NDP-friendly targets [14, 20, 41, 90], and program compilation [88]. We present our work as a general-purpose system, but it also applies to specialized NDP accelerators.

DRAM caches. 3D-stacked memories can be used as an extra cache level in front of DRAM or non-volatile memories [21, 39, 42, 47, 58, 61, 62, 66, 74, 83, 93]. Previous work mainly focused on using a single 3D memory as the cache to hide memory access latencies for a conventional processor. We instead use a part of each NDP unit's local DRAM as a cache space to reduce remote data accesses in an NDP system.

Data-aware scheduling policies. There have been efforts to incorporate data location considerations into task scheduling decisions. Some designs improved the scheduling schemes for NUMA architectures [11, 19, 28, 29, 72, 81]. Although NUMA-aware scheduling methods also leveraged data locations when doing scheduling, they often relied on the spatial locality in the applications, which might be insufficient in NDP workloads. NDP systems are also much larger and more distributed than traditional NUMA systems with significantly more separate processing units. We must consider data locations more accurately and compute data distances accordingly. NUCA-aware scheduling focused on exploiting the distributed cache bank space by keeping data in the best locations [8, 9, 40, 43, 70, 75]. We also aim at putting data closer to computations. But since our data are cached in DRAM, we need more careful designs to avoid excessive metadata cost.

9 CONCLUSIONS

In this paper, we propose ABNDP, which uses co-designed hardware and software techniques to simultaneously optimize data access and load balance problems in NDP systems. ABNDP adopts a hardware DRAM cache design and a software hybrid scheduling policy. The computation tasks can be flexibly scheduled to multiple data cache locations with both balanced loads and small access distances. The experimental results demonstrate that ABNDP successfully alleviates the two conflicting issues, while previous techniques can only improve one and sacrifice the other.

ACKNOWLEDGMENTS

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] Shaizeen Aga, Nuwan Jayasena, and Mike Ignatowski. 2019. Co-ML: A Case for Collaborative ML Acceleration Using Near-Data Processing. In *International Symposium on Memory Systems (MEMSYS)*.
[2] Junwhan Ahn, Sungpack Hong, Sungjoo Yoo, Onur Mutlu, and Kiyoung Choi. 2015. A Scalable Processing-in-Memory Accelerator for Parallel Graph Processing. In *42nd ACM/IEEE Annual International Symposium on Computer Architecture (ISCA)*.
[3] Junwhan Ahn, Sungjoo Yoo, Onur Mutlu, and Kiyoung Choi. 2015. PIM-Enabled Instructions: A Low-Overhead, Locality-Aware Processing-in-Memory Architecture. In *42nd ACM/IEEE Annual International Symposium on Computer Architecture (ISCA)*.
[4] Bahar Asgari, Ramyad Hadidi, Jiashen Cao, Da Eun Shim, Sung Kyu Lim, and Hyesoon Kim. 2021. FAFNIR: Accelerating Sparse Gathering by Using Efficient Near-Memory Intelligent Reduction. In *27th IEEE International Symposium on High-Performance Computer Architecture (HPCA)*.
[5] Hadi Asghari-Moghaddam, Young Hoon Son, Jung Ho Ahn, and Nam Sung Kim. 2016. Chameleon: Versatile and Practical Near-DRAM Acceleration Architecture for Large Memory Systems. In *49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*.
[6] Rajeev Balasubramonian, Jichuan Chang, Troy Manning, Jaime H Moreno, Richard Murphy, Ravi Nair, and Steven Swanson. 2014. Near-Data Processing: Insights From a MICRO-46 Workshop. *IEEE Micro* 34, 4 (2014).
[7] Rajeev Balasubramonian, Andrew B. Kahng, Naveen Muralimanohar, Ali Shafiee, and Vaishnav Srinivas. 2017. CACTI 7: New Tools for Interconnect Exploration in Innovative Off-Chip Memories. *ACM Transactions on Architecture and Code Optimization (TACO)* 14, 2 (2017).
[8] Nathan Beckmann and Daniel Sanchez. 2013. Jigsaw : Scalable Software-Defined Caches. In *22nd International Conference on Parallel Architectures and Compilation Techniques (PACT)*.
[9] Nathan Beckmann, Po An Tsai, and Daniel Sanchez. 2015. Scaling Distributed Cache Hierarchies through Computation and Data Co-Scheduling. In *21st IEEE International Symposium on High Performance Computer Architecture (HPCA)*.
[10] Payman Behnam and Mahdi Nazm Bojnordi. 2020. RedCache: Reduced DRAM Caching. In *57th ACM/IEEE Design Automation Conference (DAC)*.
[11] Naama Ben-David, Ziv Scully, and Guy E. Blelloch. 2019. Unfair Scheduling Patterns in NUMA Architectures. In *28th International Conference on Parallel Architectures and Compilation Techniques (PACT)*.
[12] Maciej Besta, Raghavendra Kanakagiri, Grzegorz Kwasniewski, Rachata Ausavarungnirun, Jakub Beránek, Konstantinos Kanellopoulos, Kacper Janda, Zur Vonarburg-Shmaria, Lukas Gianinazzi, Ioana Stefan, et al. 2021. SISA: Set-Centric Instruction Set Architecture For Graph Mining on Processing-in-Memory Systems. In *54th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*.
[13] Robert D Blumofe and Charles E Leiserson. 1999. Scheduling Multithreaded Computations by Work Stealing. *Journal of the ACM (JACM)* 46, 5 (1999).
[14] Amirali Boroumand, Saugata Ghose, Youngsok Kim, Rachata Ausavarungnirun, Eric Shiu, Rahul Thakur, Daehyun Kim, Aki Kuusela, Allan Knies, Parthasarathy Ranganathan, and Onur Mutlu. 2018. Google Workloads for Consumer Devices: Mitigating Data Movement Bottlenecks. In *23rd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.
[15] Amirali Boroumand, Saugata Ghose, Minesh Patel, Hasan Hassan, Brandon Lucia, Rachata Ausavarungnirun, Kevin Hsieh, Nastaran Hajinazar, Krishna T. Malladi, Hongzhong Zheng, and Onur Mutlu. 2019. CoNDA: Efficient Cache Coherence Support for Near-Data Accelerators. In *46th ACM/IEEE Annual International Symposium on Computer Architecture (ISCA)*.
[16] Amirali Boroumand, Saugata Ghose, Minesh Patel, Hasan Hassan, Brandon Lucia, Nastaran Hajinazar, Kevin Hsieh, Krishna T Malladi, Hongzhong Zheng, and Onur Mutlu. 2017. LazyPIM: Efficient support for cache coherence in processing-in-memory architectures. In *arXiv preprint arXiv:1706.03162*.
[17] Damla Senol Cali, Gurpreet S. Kalsi, Zülal Bingöl, Can Firtina, Lavanya Subramanian, Jeremie S. Kim, Rachata Ausavarungnirun, Mohammed Alser, Juan Gomez-Luna, Amirali Boroumand, Anant Norion, Allison Scibisz, Sreenivas Subramoneyon, Can Alkan, Saugata Ghose, and Onur Mutlu. 2020. 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)*.
[18] Niladrish Chatterjee, Mike O'Connor, Donghyuk Lee, Daniel R Johnson, Stephen W Keckler, Minsoo Rhu, and William J Dally. 2017. Architecting an Energy-Efficient DRAM System for GPUs. In *23rd IEEE International Symposium on High Performance Computer Architecture (HPCA)*.
[19] Quan Chen, Yawen Chen, Zhiyi Huang, and Minyi Guo. 2012. WATS: Workload-Aware Task Scheduling in Asymmetric Multi-Core Architectures. In *IEEE 26th International Parallel and Distributed Processing Symposium (IPDPS)*.
[20] Shuang Chen, Yi Jiang, Christina Delimitrou, and Jose F. Martinez. 2022. PIMCloud: QoS-Aware Resource Management of Latency-Critical Applications in Clouds with Processing-in-Memory. In *28th IEEE International Symposium on High Performance Computer Architecture (HPCA)*.
[21] Chiachen Chou, Aamer Jaleel, and Moinuddin K. Qureshi. 2016. CANDY: Enabling Coherent DRAM Caches for Multi-Node Systems. In *49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*.
[22] OpenMP Committee. 2013. OpenMP 4.0 Complete Specifications. https://openmp.org/wp-content/uploads/OpenMP4.0.0.pdf.
[23] Vidushi Dadu and Tony Nowatzki. 2022. TaskStream: Accelerating Task-Parallel Workloads by Recovering Program Structure. In *27th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.
[24] Guohao Dai, Tianhao Huang, Yuze Chi, Jishen Zhao, Guangyu Sun, Yongpan Liu, Yu Wang, Yuan Xie, and Huazhong Yang. 2019. GraphH: A Processing-in-Memory Architecture for Large-Scale Graph Processing. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)* 38, 4 (2019).
[25] Guohao Dai, Zhenhua Zhu, Tianyu Fu, Chiyue Wei, Bangyan Wang, Xiangyu Li, Yuan Xie, Huazhong Yang, and Yu Wang. 2022. DIMMining: Pruning-Efficient and Parallel Graph Mining on Near-Memory-Computing. In *49th ACM/IEEE Annual International Symposium on Computer Architecture (ISCA)*.
[26] Reetuparna Das, Rachata Ausavarungnirun, Onur Mutlu, Akhilesh Kumar, and Mani Azimi. 2013. Application-to-Core Mapping Policies to Reduce Memory System Interference in Multi-Core Systems. In *19th IEEE International Symposium on High Performance Computer Architecture (HPCA)*.
[27] Timothy A Davis and Yifan Hu. 2011. The University of Florida Sparse Matrix Collection. *ACM Transactions on Mathematical Software (TOMS)* 38, 1 (2011).
[28] Andi Drebes, Antoniu Pop, Karine Heydemann, Albert Cohen, and Nathalie Drach. 2016. 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)*.
[29] Andi Drebes, Antoniu Pop, Karine Heydemann, Nathalie Drach, and Albert Cohen. 2016. 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)*.

[30] Mario Drumond, Alexandros Daglis, Nooshin Mirzadeh, Dmitrii Ustiugov, Javier Picorel, Babak Falsafi, Boris Grot, and Dionisios Pnevmatikatos. 2017. The Mondrian Data Engine. In *44nd ACM/IEEE Annual International Symposium on Computer Architecture (ISCA)*.

[31] Amin Farmahini-Farahani, Jung Ho Ahn, Katherine Morrow, and Nam Sung Kim. 2015. NDA: Near-DRAM Acceleration Architecture Leveraging Commodity DRAM Devices and Standard Memory Modules. In *21st IEEE International Symposium on High Performance Computer Architecture (HPCA)*.

[32] Siying Feng, Xin He, Kuan-Yu Chen, Liu Ke, Xuan Zhang, David Blaauw, Trevor Mudge, and Ronald Dreslinski. 2022. MeNDA: A Near-Memory Multi-Way Merge Solution for Sparse Transposition and Dataflows. In *49th ACM/IEEE Annual International Symposium on Computer Architecture (ISCA)*.

[33] Mingyu Gao, Grant Ayers, and Christos Kozyrakis. 2015. Practical Near-Data Processing for In-Memory Analytics Frameworks. In *24th International Conference on Parallel Architectures and Compilation Techniques (PACT)*.

[34] Mingyu Gao and Christos Kozyrakis. 2016. HRL: Efficient and Flexible Reconfigurable Logic for Near-Data Processing. In *22nd IEEE International Symposium on High Performance Computer Architecture (HPCA)*.

[35] Mingyu Gao, Jing Pu, Xuan Yang, Mark Horowitz, and Christos Kozyrakis. 2017. Tetris: Scalable and Efficient Neural Network Acceleration With 3D Memory. In *20nd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[36] Christina Giannoula, Nandita Vijaykumar, Nikela Papadopoulou, Vasileios Karakostas, Ivan Fernandez, Juan Gómez-Luna, Lois Orosa, Nectarios Koziris, Georgios Goumas, and Onur Mutlu. 2021. SynCron: Efficient Synchronization Support for Near-Data-Processing Architectures. In *27th IEEE International Symposium on High Performance Computer Architecture (HPCA)*.

[37] Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. 2012. PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs. In *10th USENIX Conference on Operating Systems Design and Implementation (OSDI)*.

[38] Peng Gu, Xinfeng Xie, Yufei Ding, Guoyang Chen, Weifeng Zhang, Dimin Niu, and Yuan Xie. 2020. iPIM: Programmable In-Memory Image Processing Accelerator Using Near-Bank Architecture. In *47nd ACM/IEEE Annual International Symposium on Computer Architecture (ISCA)*.

[39] Nagendra Gulur, Mahesh Mehendale, R. Manikantan, and R. Govindarajan. 2015. Bi-Modal DRAM Cache: Improving Hit Rate, Hit Latency and Bandwidth. In *48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*.

[40] Nikos Hardavellas, Michael Ferdman, Babak Falsafi, and Anastasia Ailamaki. 2009. Reactive NUCA: Near-Optimal Block Placement and Replication in Distributed Caches. In *36th ACM/IEEE Annual International Symposium on Computer Architecture (ISCA)*.

[41] Kevin Hsieh, Eiman Ebrahimi, Gwangsun Kim, Niladrish Chatterjee, Mike O'Connor, Nandita Vijaykumar, Onur Mutlu, and Stephen W Keckler. 2016. Transparent Offloading and Mapping (TOM) Enabling Programmer-Transparent Near-Data Processing in GPU Systems. In *43rd ACM/IEEE Annual International Symposium on Computer Architecture (ISCA)*.

[42] Cheng-Chieh Huang and Vijay Nagarajan. 2014. ATCache: Reducing DRAM Cache Latency via a Small SRAM Tag Cache. In *23rd International Conference on Parallel Architectures and Compilation (PACT)*.

[43] Aamer Jaleel, Hashem H. Najaf-abadi, Samantika Subramaniam, Simon C. Steely, and Joel Emer. 2012. CRUISE: Cache Replacement and Utility-Aware Scheduling. In *17th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

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

[45] Mark C. Jeffrey, Suvinay Subramanian, Maleen Abeydeera, Joel Emer, and Daniel Sanchez. 2016. Data-Centric Execution of Speculative Parallel Programs. In *49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*.

[46] Mark C. Jeffrey, Suvinay Subramanian, Cong Yan, Joel Emer, and Daniel Sanchez. 2015. A Scalable Architecture for Ordered Parallelism. In *48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*.

[47] Djordje Jevdjic, Gabriel H. Loh, Cansu Kaynak, and Babak Falsafi. 2015. Unison Cache: A Scalable and Effective Die-Stacked DRAM Cache. In *48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*.

[48] Djordje Jevdjic, Stavros Volos, and Babak Falsafi. 2013. Die-Stacked DRAM Caches for Servers Hit Ratio, Latency, or Bandwidth? Have It All with Footprint Cache. In *40th ACM/IEEE Annual International Symposium on Computer Architecture (ISCA)*.

[49] Hongshin Jun, Jinhee Cho, Kangseol Lee, Ho-Young Son, Kwiwook Kim, Hanho Jin, and Keith Kim. 2017. HBM (High Bandwidth Memory) DRAM Technology and Architecture. In *2017 IEEE International Memory Workshop (IMW)*.

[50] Hongju Kal, Seokmin Lee, Gun Ko, and Won Woo Ro. 2021. SPACE : Locality-Aware Processing in Heterogeneous Memory for Personalized Recommendations. In *48th ACM/IEEE Annual International Symposium on Computer Architecture (ISCA)*.

[51] Liu Ke, Udit Gupta, Benjamin Youngjae Cho, David Brooks, Vikas Chandra, Utku Diril, Amin Firoozshahian, Kim Hazelwood, Bill Jia, Hsien-Hsin S. Lee, Meng Li, Bert Maher, Dheevatsa Mudigere, Maxim Naumov, Martin Schatz, Mikhail Smelyanskiy, Xiaodong Wang, Brandon Reagen, Carole-Jean Wu, Mark Hempstead, and Xuan Zhang. 2020. RecNMP: Accelerating Personalized Recommendation with Near-Memory Processing. In *47th ACM/IEEE Annual International Symposium on Computer Architecture (ISCA)*.

[52] Duckhwan Kim, Jaeha Kung, Sek Chai, Sudhakar Yalamanchili, and Saibal Mukhopadhyay. 2016. NeuroCube: A Programmable Digital Neuromorphic Architecture with High-Density 3D Memory. In *43rd ACM/IEEE Annual International Symposium on Computer Architecture (ISCA)*.

[53] Gwangsun Kim, John Kim, Jung Ho Ahn, and Jaeha Kim. 2013. Memory-Centric System Interconnect Design with Hybrid Memory Cubes. In *22nd International Conference on Parallel Architectures and Compilation Techniques (PACT)*.

[54] Jeremie S. Kim, Damla Senol Cali, Hongyi Xin, Donghyuk Lee, Saugata Ghose, Mohammed Alser, Hasan Hassan, Oguz Ergin, Can Alkan, and Onur Mutlu. 2018. GRIM-Filter: Fast Seed Location Filtering in DNA Read Mapping Using Processing-in-Memory Technologies. *BMC Genomics* 19, 2 (2018).

[55] Orhan Kislal, Jagadish Kotra, Xulong Tang, Mahmut Taylan Kandemir, and Myoungsoo Jung. 2018. Enhancing Computation-to-Core Assignment with Physical Location Information. In *39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*.

[56] Youngeun Kwon, Yunjae Lee, and Minsoo Rhu. 2019. 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)*.

[57] Young-Cheon Kwon, Suk Han Lee, Jaehoon Lee, Sang-Hyuk Kwon, Je Min Ryu, Jong-Pil Son, O Seongil, Hak-Soo Yu, Haesuk Lee, Soo Young Kim, Youngmin Cho, Jin Guk Kim, Jongyoon Choi, Hyun-Sung Shin, Jin Kim, BengSeng Phuah, HyoungMin Kim, Myeong Jun Song, Ahn Choi, Daeho Kim, SooYoung Kim, Eun-Bong Kim, David Wang, Shinhaeng Kang, Yuhwan Ro, Seungwoo Seo, JoonHo Song, Jaeyoun Youn, Kyomin Sohn, and Nam Sung Kim. 2021. 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 *2021 IEEE International Solid- State Circuits Conference (ISSCC)*.

[58] Yongjun Lee, Jongwon Kim, Hakbeom Jang, Hyunggyun Yang, Jangwoo Kim, Jinkyu Jeong, and Jae W. Lee. 2015. A Fully Associative, Tagless DRAM Cache. In *42nd ACM/IEEE Annual International Symposium on Computer Architecture (ISCA)*.

[59] Baptiste Lepers, Vivien Quéma, and Alexandra Fedorova. 2015. Thread and Memory Placement on NUMA Systems: Asymmetry Matters. In *2015 USENIX Annual Technical Conference (USENIX ATC)*.

[60] Jure Leskovec and Rok Sosič. 2016. SNAP: A General-Purpose Network Analysis and Graph-Mining Library. *ACM Transactions on Intelligent Systems and Technology (TIST)* 8, 1 (2016).

[61] Gabriel H Loh. 2009. Extending the Effectiveness of 3D-Stacked DRAM Caches With an Adaptive Multi-Queue Policy. In *42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*.

[62] Gabriel H. Loh and Mark D. Hill. 2011. Efficiently Enabling Conventional Block Sizes for Very Large Die-Stacked DRAM Caches. In *44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*.

[63] Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. 2010. Pregel: A System for Large-Scale Graph Processing. In *2010 ACM SIGMOD International Conference on Management of Data*.

[64] Sally A McKee. 2004. Reflections on the Memory Wall. In *Proceedings of the 1st Conference on Computing Frontiers*.

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

[66] Onur Mutlu and Srinivas Devadas. 2017. Banshee: Bandwidth-Efficient DRAM Caching via Software/Hardware Cooperation High-Bandwidth In-Package DRAM. In *50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*.

[67] Lifeng Nai, Ramyad Hadidi, Jaewoong Sim, Hyojong Kim, Pranith Kumar, and Hyesoon Kim. 2017. GraphPIM: Enabling Instruction-Level PIM Offloading in Graph Computing Frameworks. In *23rd IEEE Symposium on High Performance Computer Architecture (HPCA)*.

[68] Ravi Nair, Samuel F Antao, Carlo Bertolli, Pradip Bose, Jose R Brunheroto, Tong Chen, C-Y Cher, Carlos HA Costa, Jun Doi, Constantinos Evangelinos, et al. 2015. Active Memory Cube: A Processing-In-Memory Architecture for Exascale Systems. *IBM Journal of Research and Development* 59, 2/3 (2015).

[69] Jaehyun Park, Byeongho Kim, Sungmin Yun, Eojin Lee, Minsoo Rhu, and Jung Ho Ahn. 2021. TRiM: Enhancing Processor-Memory Interfaces with Scalable Tensor Reduction in Memory. In *54th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*.

[70] Anuj Pathania. 2018. Task Scheduling for Many-Cores with S-NUCA Caches. In *Design, Automation & Test in Europe Conference & Exhibition (DATE)*.

[71] Matthew Poremba, Itir Akgun, Jieming Yin, Onur Kayiran, Yuan Xie, and Gabriel H. Loh. 2017. There and Back Again: Optimizing the Interconnect in Networks of Memory Cubes. In *44th ACM/IEEE Annual International Symposium*

on Computer Architecture (ISCA).

[72] Iraklis Psaroudakis, Tobias Scheuer, Norman May, Abdelkader Sellami, and Anastasia Ailamaki. 2016. 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).*

[73] Seth H Pugsley, Jeffrey Jestes, Huihui Zhang, Rajeev Balasubramonian, Vijayalakshmi Srinivasan, Alper Buyuktosunoglu, Al Davis, and Feifei Li. 2014. NDC: Analyzing the Impact of 3D-Stacked Memory+Logic Devices on MapReduce Workloads. In *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS).*

[74] Moinuddin K Qureshi and Gabe H Loh. 2012. Fundamental Latency Trade-off in Architecting DRAM Caches: Outperforming Impractical SRAM-tags with a Simple and Practical Design. In *45th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO).*

[75] Martin Rapp, Anuj Pathania, Tulika Mitra, and Jörg Henkel. 2019. Prediction-Based Task Migration on S-NUCA Many-Cores. In *Design, Automation & Test in Europe Conference & Exhibition (DATE).*

[76] Amitabha Roy, Ivo Mihailovic, and Willy Zwaenepoel. 2013. X-Stream: Edge-Centric Graph Processing Using Streaming Partitions. In *24th ACM Symposium on Operating Systems Principles (SOSP).*

[77] Daniel Sanchez and Christos Kozyrakis. 2013. ZSim: Fast and Accurate Microarchitectural Simulation of Thousand-Core Systems. In *40th ACM/IEEE Annual International Symposium on Computer Architecture (ISCA).*

[78] Daniel Sanchez, David Lo, Richard M. Yoo, Jeremy Sugerman, and Christos Kozyrakis. 2011. Dynamic Fine-Grain Scheduling of Pipeline Parallelism. In *20th International Conference on Parallel Architectures and Compilation Techniques (PACT).*

[79] André Seznec and Francois Bodin. 1993. Skewed-Associative Caches. In *International Conference on Parallel Architectures and Languages Europe (PARLE).*

[80] Yakun Sophia Shao, Jason Clemons, Rangharajan Venkatesan, Brian Zimmer, Matthew Fojtik, Nan Jiang, Ben Keller, Alicia Klinefelter, Nathaniel Pinckney, Priyanka Raina, et al. 2019. Simba: Scaling Deep-Learning Inference with Multi-Chip-Module-Based Architecture. In *52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO).*

[81] Shumpei Shiina and Kenjiro Taura. 2019. Almost Deterministic Work Stealing. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC).*

[82] Julian Shun and Guy E. Blelloch. 2013. Ligra: A Lightweight Graph Processing Framework for Shared Memory. In *18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP).*

[83] Jaewoong Sim, Gabriel H. Loh, Hyesoon Kim, Mike Oconnor, and Mithuna Thottethodi. 2012. A Mostly-Clean DRAM Cache for Effective Hit Speculation and Self-Balancing Dispatch. In *45th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO).*

[84] Gagandeep Singh, Juan Gómez-Luna, Giovanni Mariani, Geraldo F Oliveira, Stefano Corda, Sander Stuijk, Onur Mutlu, and Henk Corporaal. 2019. Napel: Near-Memory Computing Application Performance Prediction Via Ensemble Learning. In *56th ACM/IEEE Design Automation Conference (DAC).*

[85] Weiyi Sun, Zhaoshi Li, Shouyi Yin, Shaojun Wei, and Leibo Liu. 2021. ABC-DIMM: Alleviating the Bottleneck of Communication in DIMM-based Near-Memory Processing with Inter-DIMM Broadcast. In *48th ACM/IEEE Annual International Symposium on Computer Architecture (ISCA).*

[86] Isaac Sánchez Barrera, Miquel Moretó, Eduard Ayguadé, Jesús Labarta, Mateo Valero, and Marc Casas. 2018. Reducing Data Movement on Large Shared Memory Systems by Exploiting Computation Dependencies. In *Proceedings of the International Conference on Supercomputing (ICS).*

[87] Nishil Talati, Haojie Ye, Yichen Yang, Leul Belayneh, Kuan-Yu Chen, David Blaauw, Trevor Mudge, and Ronald Dreslinski. 2022. NDMiner: Accelerating Graph Pattern Mining Using Near Data Processing. In *49th ACM/IEEE Annual International Symposium on Computer Architecture (ISCA).*

[88] Xulong Tang, Orhan Kislal, Mahmut Kandemir, and Mustafa Karakoy. 2017. Data Movement Aware Computation Partitioning. In *50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO).*

[89] Po-An Tsai, Nathan Beckmann, and Daniel Sanchez. 2017. Jenga: Software-Defined Cache Hierarchies. In *47th ACM/IEEE Annual International Symposium on Computer Architecture (ISCA).*

[90] Po An Tsai, Changping Chen, and Daniel Sanchez. 2018. Adaptive Scheduling for Systems with Asymmetric Memory Hierarchies. In *51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO).*

[91] Xinfeng Xie, Zheng Liang, Peng Gu, Abanti Basak, Lei Deng, Ling Liang, Xing Hu, and Yuan Xie. 2021. SpaceA: Sparse Matrix Vector Multiplication on Processing-in-Memory Accelerator. In *27th IEEE International Symposium on High-Performance Computer Architecture (HPCA).*

[92] Victor A. Ying, Mark C. Jeffrey, and Daniel Sanchez. 2020. T4: Compiling Sequential Code for Effective Speculative Parallelization in Hardware. In *47th ACM/IEEE Annual International Symposium on Computer Architecture (ISCA).*

[93] Vinson Young, Chiachen Chou, Aamer Jaleel, and Moinuddin Qureshi. 2018. ACCORD: Enabling Associativity for Gigascale DRAM Caches by Coordinating Way-Install and Way-Prediction. In *45th ACM/IEEE Annual International Symposium on Computer Architecture (ISCA).*

[94] Jia Zhan, Itir Akgun, Jishen Zhao, Al Davis, Paolo Faraboschi, Yuangang Wang, and Yuan Xie. 2016. A Unified Memory Network Architecture for In-Memory Computing in Commodity Servers. In *49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO).*

[95] Dong Ping Zhang, Nuwan Jayasena, Alexander Lyashevsky, Joseph L. Greathouse, Lifan Xu, and Michael Ignatowski. 2014. TOP-PIM: Throughput-Oriented Programmable Processing in Memory. In *23rd International Symposium on High-Performance Parallel and Distributed Computing (HPDC).*

[96] Mingxing Zhang, Youwei Zhuo, Chao Wang, Mingyu Gao, Yongwei Wu, Kang Chen, Christos Kozyrakis, and Xuehai Qian. 2018. GraphP: Reducing Communication for PIM-Based Graph Processing with Efficient Data Partition. In *24th IEEE International Symposium on High Performance Computer Architecture (HPCA).*

[97] Youwei Zhuo, Chao Wang, Mingxing Zhang, Rui Wang, Dimin Niu, Yanzhi Wang, and Xuehai Qian. 2019. GraphQ: Scalable PIM-Based Graph Processing. In *52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO).*

Received 2022-10-20; accepted 2023-01-19