Parallel Hash Table Design for NDP Systems

Pranith Kumar
Georgia Institute of Technology
Atlanta, Georgia, USA
pranith@gatech.edu

Hyesoon Kim
Georgia Institute of Technology
Atlanta, Georgia, USA
hyesoon@cc.gatech.edu

ABSTRACT

With the increasing feasibility of die-stacked 3D memory, near-data-processing (NDP) is being widely explored to extract greater performance within a limited power budget. This processing can either be at a fine-grained instruction granularity or at coarse-grained kernel granularity. Allowing both the host processor and processing units in the memory to operate on data concurrently can potentially create coherence and consistency issues. While coherence problems have been solved by including the NDP memory in the coherence domain, porting parallel data structures like hash-table to NDP memory give rise to data structure consistency issues that have not been studied so far, as previous works do not discuss the consistency rules that should be enforced by a NDP memory controller. Instead, there is an implicit assumption that the memory controllers in NDP systems ensure the required order for memory requests.

In this position paper, we propose techniques to adapt a traditional parallel hash-table data structure to a NDP system while ensuring improved performance and data structure consistency.

CCS CONCEPTS

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

KEYWORDS

Processing-in-memory, near memory, hashtable

ACM Reference Format:
Pranith Kumar and Hyesoon Kim. 2020. Parallel Hash Table Design for NDP Systems. In *The International Symposium on Memory Systems (MEMSYS 2020), September 28-October 1, 2020, Washington, DC, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3422575.3422776

1 INTRODUCTION

Recent advances in die-stacked 3D memory technology reignited decades old near-data processing (NDP) research. In these proposed NDP systems, computation logic is placed beneath a 3D-stacked memory and it is accessed using high bandwidth TSVs. There are two prominent programming models for such processing in memory systems: fine-grain instruction-level offloading and coarse-grain kernel-level offloading. As an example of fine-grain instruction offloading, recent studies have evaluated extending traditional DDR memory controllers to include computing capability [4, 8, 11, 15]. Researchers have also proposed several primitives such as atomic add/multiply, hash-insertion [1], and bit operations [18]. These instructions can atomically read-modify-write inside the memory system, allowing both the host processor and the computing logic in the memory to operate on the data concurrently. Such concurrent modifications cause cache coherence and data consistency issues.

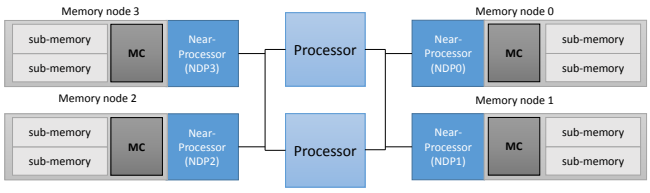

Figure 1: Architecture of a NDP system with 4 memory nodes and 2 processors. Near-Processor is referred as NDP cores in the text and MC is the memory controller.

The solution proposed by researchers to overcome the cache coherence challenge in NDP systems was to bring the memory into the coherence domain of the processor which is traditionally limited to caches and associated structures. This is achieved by either partial or full inclusion of the NDP memory in the coherence protocol of the processor. The other option is to implement the NDP memory regions as non cache-able memory[1, 12, 13]. We summarize previous works on coherence and consistency of NDP systems in Table 1. Please note that when we refer to consistency here, we are referring to the mechanism by which a NDP-mapped hash-table data structure integrity is ensured. In traditional systems, this is ensured using atomic instructions, locks, and memory barrier instructions that are inserted as needed according to the processor consistency model.

In this paper, we first discuss the scenarios in which NDP computing violates parallel data structure consistency. Then, we discuss how a data structure can be re-designed to avoid these consistency violations. We analyze and evaluate a common data structure, the hash-table, and provide a re-design to adapt it to NDP systems.

The contributions of the paper are as follows.

(1) We present a novel method to adapt traditional data structures to NDP fully exploiting the NDP specific characteristics like serialization and sub-memory regions.
(2) We present a programming technique to redesign data structures to avoid consistency problems for hash-tables when adapted to NDP systems.

2 ARCHITECTURE

A NDP system is made up of traditional processing cores that are connected to memory nodes over an unordered network. Figure 1 illustrates the baseline architecture that we consider in our current work. A brief description of this baseline follows.

Offloading	Studies	Coherence	Consistency*	Comments
instruction level	[12],[1],[14],[13]	non-cached	follow host consistency model	use non-temporal instructions
instruction level	[21],[11],[5],[4]	scratch pad	multicast barrier	-
kernel level	[22],[16],[25],[15]	software-assisted	barrier	flush cache on modification

Table 1: Classification of previous NDP proposals on Coherence and Consistency models.

The memory system contains several memory nodes and each memory node contains computational logic in near-data processing (NDP) cores. The system memory is distributed among all the memory nodes. This memory in-turn is divided into sub-memory regions (which can be considered as banks). There is a memory controller (MC) that facilitates communication between the NDP cores and the sub-memory regions. Depending on the 3D stacking technology, the NDP cores might be in the bottom layer of 3D stack, or they might be connected using a 2.5D stack. Each NDP core has a low latency, high-bandwidth access to the local memory node and a high latency, low-bandwidth access to the remote memory nodes.

We assume that the NDP memory controller can perform read-modify-write (RMW), and compare-swap operations. The NDP cores are complex enough to have their own program counters and virtual to physical memory address translations and kernel-level offloading on accelerators is assumed.

3 MOTIVATION

In this section, we discuss the consistency issues that arise on NDP systems when instruction level and kernel level offloading are implemented.

3.1 Instruction Level Offloading

High-performance memory systems that use 3D die stacking technology are in active development. Researchers are aiming not only to increase the memory bandwidth and performance, but also to include computing capabilities. Earlier efforts included a specialized memory named HMC, short for Hybrid Memory Cube, as one example which has introduced the capability to offload certain computations from the host processor to the memory system [9, 17]. Most recently, efforts have focused on adding computational logic to traditional DIMMs so as to utilize regular DDR memory for processing-in-memory systems. This computational logic includes instructions that can atomically read-modify-write the memory without having to fetch the data into the processor's cache. In current systems, to maintain data structure consistency, we utilize the host processor atomic and barrier instructions and the processor ensures the consistency without involving the memory system. However, when memory starts to perform computations directly, it is likely to violate these guarantees if the data structure is not carefully designed. CPU[1] Atomic operations have high overhead to provide processors' consistency semantics[23]. If utilizing NDP's atomic operations has similar overhead, the possible use cases for such instructions will be limited by the overhead.

Since with instruction offloading, loads and stores are all performed in the memory directly (by either operating on a non-cached memory location or invalidating the data in cache), at a first glance, it seems like NDP atomic instructions are free from any consistency violations. However, as we show, this does not always hold.

NDP atomic operations execute in three steps: reading data from DRAM, performing an operation on the data in the logic die, and then writing back the result to the same DRAM location. These steps occur atomically; the corresponding DRAM bank is locked during the atomic request so that no other requests to the same bank can be interleaved. Besides, all NDP instructions access only one memory location (single memory operand). However, memory requests that are operating on different banks can execute in any order to maximize the memory bank parallelism. Furthermore, if memory requests to the same address are issued through different serial I/O links that connect multiple NDPs and processors, the order between these memory requests is also not preserved [9].

3.2 Consistency issues on kernel-level offloading systems

Figure 2 demonstrates an example of potential consistency violations with kernel-level offloading model on NDP systems. In this example we consider operations on a linked-list data structure. In kernel-level offloading, the entire search or delete function is offloaded to NDP system. Nonetheless, each kernel needs to generate multiple memory requests to individual memory nodes. Since, memory requests to the same memory node are serialized, they do not violate the data structure consistency. However, when different NDP cores access the same memory node concurrently, the illustrated inconsistency arises. [2]

In this example, there are two operations (Figure 2(a)) on a linked-list data structure that consists of two elements (with values 5 and 3). These operations are issued concurrently by different nodes. Since the linked-list is distributed across all the nodes, a node can send a request to another memory node while traversing the linked list. The first operation is a *Delete* operation performed by the NDP0 core whereas the second operation is a *Search* operation being performed by the NDP1 core. These operations are composed of three memory requests that access the elements through pointers *ptr* and *curr* and are labeled as *Req*.0, *Req*.1, and *Req*.2 in the illustration. *Req*.0 is checking if the value in the element matches *num* to delete it. If it matches, the element to be deleted is saved in *tmp* and *Req*.1 remaps the next pointer of the element to be deleted to its following element. A concurrent search operation, *Req*.2 is searching for the element containing *num* by comparing the values. The two memory nodes are issuing requests to the same sub-memory region causing an interleaving of issued requests. The correct and incorrect

[1] We consider a CPU to be an x86 processor for this paper.

[2] Due to this reason, in the recent concurrent data structure algorithm for NDP did not allow a link-list spans across multiple memory nodes [19]. However, in this paper, we design a linked-list that overcomes this limitation.

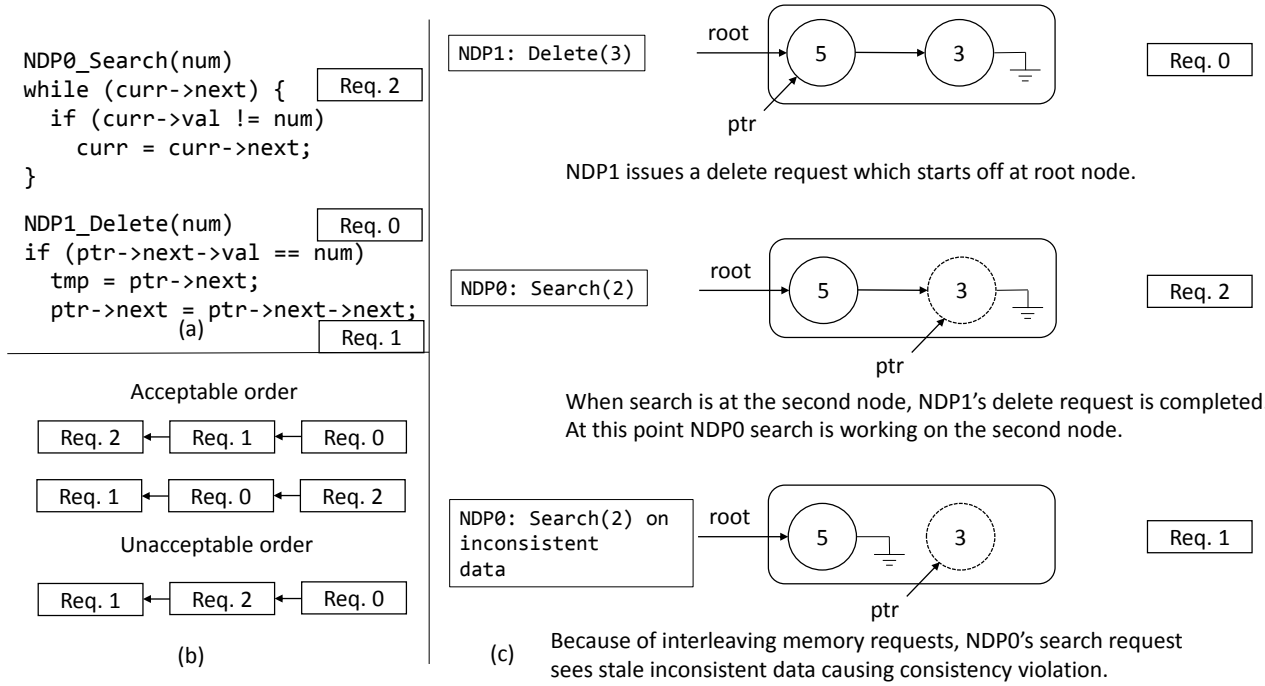

Figure 2: NDP consistency violation example with kernel level offloading

interleaving of these operations is given in Figure 2(b) and an illustration of this incorrect order in action is given in Figure2(c). As shown in the illustration, whenever the requests *Req*.0 and *Req*.1 are interleaved with *Req*.2, an element is incorrectly deleted by one NDP when the node has a pending operation issued by another NDP. Please note that there is no explicit synchronization to avoid this situation because of the assumption that these operations are serialized at the NDP memory controller.

4 HASHTABLE DATA STRUCTURE

In this section we discuss how we redesign a parallel hash-table data structure to efficiently and **correctly** map to the NDP system.

A hash table is a data structure that stores mappings of keys generated using a hash function to values. It is used to quickly search and uniquely identify an object among a collection of similar objects. We describe an in-production version of a hash-table as implemented in QEMU [7] known as the quick hash-table [10]. This hash-table is illustrated in Figure 3. The structure consists of an array of pointers. Each pointer points to a linked-list of buckets. A bucket consists of an array of keys and corresponding values as shown in Listing 1. On a traditional processor, each bucket is sized to fit in a cache line, whereas on a NDP system, we can increase the size of the bucket to a sub-memory region as illustrated in Figure 3.

There are two major characteristics of NDP systems that we exploit for adapting a hash-table. The first characteristic is that NDP operations can be performed on larger granularity memory regions (refer Figure 1). Using sub-memory region granularity allows us to avoid certain consistency issues and to reduce NDP communication

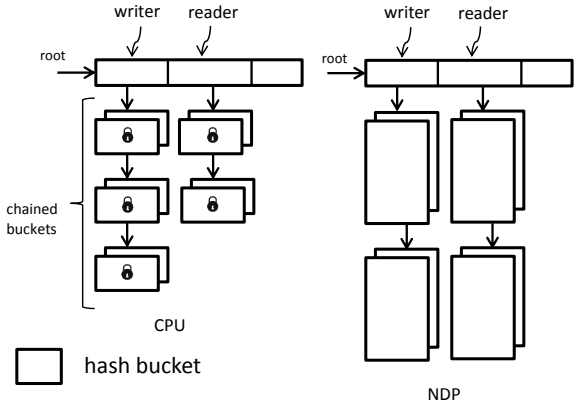

Figure 3: Organization of a Hash Table. Each square in the chain represents a hash bucket. On the left, a conventional hash-table with hash bucket that has locks is sized for a cache line of the CPU, whereas on the right a NDP-mapped hash-table has buckets that are lock free and is sized for a sub-memory region.

overhead when compared to cache line granularity on a traditional processor. The other characteristic is that all operations on a sub-memory region are serialized at the memory controller of that

```
1  struct hash_bucket {
2    lock_t    lock;   // keep writers apart
3    seqlock_t seqlock;// retry reader on update
4    uint32_t  keys[PER_BUCKET];   // keys
5    void      *values[PER_BUCKET];// values
6    struct hash_bucket *next; // next bucket
7  };
8
9  struct hash_table {
10   hash_bucket *buckets;
11   int num_buckets;
12 };
```

Listing 1: Structures in a Hash Table

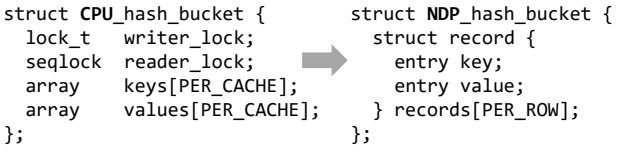

Figure 4: NDP Optimal Structure Reorganization for a Hash Table

node. As we describe later, this allows us to reduce the cost of synchronization when compared to the traditional design.

We consider three main operations in the hash table: (i) Search (ii) Insert and (iii) Delete.

4.1 Hash Table Organization

The hash bucket is the main structure that holds the keys (refer Listing 1 – line 4) and their corresponding hash values (line 5). These data are stored in two arrays, where in traditional processors, each array is sized to fit in a cache line (*PER_BUCKET*) to exploit the spatial locality of the hash-table operations access patterns. Every operation on a bucket acquires a lock. Readers take the *seqlock* whereas writers take the *lock*.

To adapt this to the larger granularity memory region of a NDP system without having to utilize locks, we utilize a well known memory layout transformation known as Structure of Arrays to Array of Structures [2, 3, 24](SoA to AoS [3]) to effectively utilize the organization of the NDP system. This is is illustrated in Figure 4. This reorganization of the fields in the hash bucket allows it to (i) co-locate the key and value pairs in the same sub-memory region instead of them being present in different regions and having to access the two arrays in lock-step without synchronization and (ii) exploit the larger granularity of the sub-memory region (512 B) compared to the cache line size (64 B) in a processor thereby reducing the number of accesses to each bucket by 8x (512/64).

4.2 Hash Table Operations

We describe the main hash-table operations below.

Search is one of the main operations on a hash-table. This is implemented using the *lookup* and *search* functions as shown in Listing 2 and Listing 3. A *key* is used to search for a particular hash value. It returns the corresponding value if such an entry exists.

[3]This transformation is usually applied the other way around i.e., from AoS to SoA. However, in this scenario SoA to AoS is interestingly beneficial.

```
1  /* Conventional lookup */
2  void *lookup(table, hash)
3  {
4    int index = map_hash_to_index(hash);
5    int version;
6    void *result = NULL;
7    bucket *bucket = table->buckets[index];
8
9    do {
10     version = seqlock_read_begin(bucket);
11     // how do you map search to NDP?
12     result = search(bucket, hash, PER_BUCKET);
13   } while (seqlock_read_end(bucket) != verison);
14
15   return result;
16 }
```

Listing 2: A lookup operation in a Hash Table

```
1  /* Conventional Search */
2  void *search(hash_bucket *bucket, uint32_t hash, int
   ↪  num_entries)
3  {
4    for (int i = 0; i < num_entries; i++) {
5      if (bucket->hashes[i] == hash)
6        return bucket->pointers[i];
7    }
8
9    /* hash not found */
10   return NULL;
11 }
```

Listing 3: Search operation in Hash Table

Sequence locks (seqlock [20]) are used to separate the readers and the writers in the traditional version of the hash-table. If the seqlock version changes before the reader is finished reading, there must have been a writer active who updated the structure. In this case, the reader might have read stale data and hence will retry the search operation.

In a conventional search, we iterate over all the hashes in the bucket and check for a matching entry. If such an entry is found, the index of the found hash entry is used to return the corresponding value pointer. If such a hash entry is not found, the entry does not exist and we return NULL.

4.3 Mapping Search Operation to NDP

In an NDP mapped search, we use the NDP-compare operation which searches for a value in a entire sub-memory region. No other operations are allowed to overlap with this operation i.e., it is atomic. This compare instruction returns the offset/index at which a match was found. If it does not find a match, it returns the offset of end of the memory region indicating failure. This operation is shown in Listing 4.

Using such a NDP-mapped search function will allow us to remove the sequence lock since the readers and writers are now serialized at the NDP. The entire search operation of the reader is finished before updating the bucket with the newly inserted pointer by operation of the writer. This allows us keep the reader and writer from interfering with each other, ensuring the hash-table integrity.

```c
/* NDP mapped Search */
void *search(hash_bucket *bucket, uint32_t hash, int
↪ num_entries)
{
  int offset = NDP_search(bucket->hashes, hash, sizeof(hash),
  ↪ num_entries);

  int index = offset / sizeof(hash);
  /* Found the hash */
  if (index != num_entries)
    return bucket->pointers[index];

  /* hash not found */
  return NULL;
}
```

Listing 4: NDP mapped Hash Table Search Operation

```c
/* Conventional Insertion */
void insert(node, hash)
{
  int index = map_hash_to_index(hash);
  hash_bucket *bucket = table->buckets[index];

  // disallow other writers
  lock(bucket.lock);
  // inform readers
  seqlock_write_begin(bucket.seqlock);
  bool success = insert_in_bucket(bucket, hash, node);
  seqlock_write_end(bucket.seqlock);
  unlock(bucket.lock);
}
```

Listing 5: Insert Operation for a Hash Table

```c
bool insert_in_bucket(hash_bucket *bucket, uint32_t hash, void
↪ *ptr)
{
  for (int i = 0; i < PER_BUCKET; i++) {
    if (bucket->hashes[i] == NULL) {
      bucket->pointers[i] = ptr;
      bucket->hashes[i] = hash;
      return true;
    }
  }

  // insertion failed
  return false;
}
```

Listing 6: Inserting an element into a Hash Bucket

4.4 Insertion

In a conventional implementation, an insert into the bucket takes two locks. One lock is to disallow modification by other writers and the other is a sequence lock to inform the readers currently reading to retry their search. The function *insert_in_bucket* in Listing 5 and Listing 6 iterates over the bucket looking for an empty slot to insert the pointer.

4.5 Mapping Insert Operation to NDP

Earlier, we've seen that the sequence lock can be removed since both the readers and writers are serialized at the NDP boundary.

```c
/* NDP Insertion */
bool insert_in_bucket(hash_bucket *bucket, uint32_t hash, void
↪ *ptr)
{
  int offset = NDP_compare_swap(bucket->hashes, NULL, hash,
  ↪ sizeof(hash));

  int index = offset/sizeof(hash);

  if (index != PER_BUCKET) {
    // succesfully swapped
    bucket->pointers[index] = ptr;
    return true;
  }

  // insertion failed
  return false;
}
```

Listing 7: NDP mapped Hash Table Insert Operation

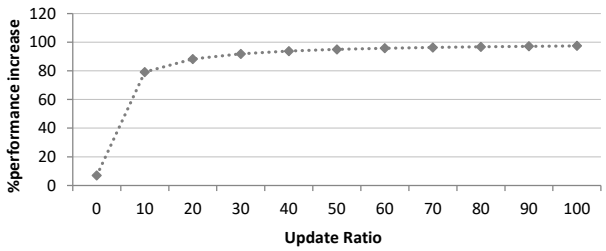

Figure 5: Performance of a NDP-mapped Hash Table

Similarly, we can also remove the writer lock using NDP atomics to prevent data races.

Using the memory region NDP **compare and swap** instruction allows us to perform this action without taking **any** lock. This instruction compares the contents of each location in a memory region at a given granularity and swaps the value with the new given value on a match. If the value does not match, the NDP instruction returns the offset of the end of the memory region indicating failure. This code is given in Listing 7.

4.6 Benefits

To summarize, the major advantages of mapping a hash-table to a NDP system are as follows:

(1) Increasing the bucket size from cache line size to sub-memory region size (64B to 512B) and eliding taking any locks increases throughput of operations
(2) Bandwidth requirement reduction by not having to bring data into the cpu
(3) Improving the search latency, since it is performed on NDP directly

5 NDP-MAPPED HASH TABLE EVALUATION

We evaluate the performance of the NDP-mapped hash-table and compare it to a traditional CPU only hash-table as shown in Figure 5. We model a 4-node NDP-mapped hash-table (8GB, 4 MC, 512B

sub-memory region) and a conventional implementation on a 16-core processor (32KB L1, 256KB L2, 16 MB LLC, 2GHz). The hash-table is populated with 4 GB entries mapped evenly over 512B sub-memory regions. We create a benchmark that spawns threads to generate concurrent traffic to the hash-table from all the CPU cores. This traffic can be a mix of read-only look-ups or write-only update/delete operations.

When only readers operate on the hash-table (i.e., the update ratio is 0), the NDP version of the hash-table is able to sustain ten percent more operations than the corresponding conventional processor only implementation. This performance improvement is due to the elimination of contention on the sequential lock in the hash bucket being accessed by the readers, increasing the granularity of the search in each bucket, and not having to fetch the memory into the processors' cache. However, if all operations are updating the hash-table (the update ratio is 100) the NDP version can sustain double the number of operations as the conventional implementation. The major portion of this performance gain is achieved by eliminating the writer lock contention and increasing the granularity of the search from a cache line in the conventional implementation to a row granularity in the NDP implementation.

6 RELATED WORK ON NDP

The work that is very closely related to the current work is on accelerating linked-list operations using NDP [16]. In their work, to avoid consistency problems, a lock is acquired on the host at a fine granularity before sending any memory updates. This causes serialization on the host side, whereas the design we present avoids acquiring locks altogether by relying on NDP serialization of commands at sub-memory region granularity.

In other proposals, NDP memory is assumed to be non cacheable, because of which a coherence protocol is not needed to be implemented between the core and the NDP. In such proposals, non-temporal instructions are used whenever the core modifies the memory. These instructions update the memory directly instead of fetching the cache line into the processor. This ensures that the updates are seen by the NDP core without changes to the coherence protocol. In other studies, NDP memory is cached in the processor core, but upon an update to the locations, the cached memory is either flushed and updated or write-through caching is employed. Although these techniques satisfy the coherence issue, consistency is not thoroughly handled in the proposals. In [1], a fence instruction is used to enforce consistency without any details of the cost or granularity of such instructions. Research in which the offloading granularity is greater like [4, 6, 25] also focus on bandwidth savings and do not consider consistency. This is because there is an implicit barrier at the end of the offloading granularity. In Epiphany, a many-core processor [21], a multi-cast hardware barrier is used to ensure consistency. In this implementation, a mesh network is used for connecting the NDP cores and each NDP core is responsible to propagate the barrier within its local row. A per-core multi-cast register is set which interrupts the NDP core which waits until the register is cleared before processing any further instructions ensuring consistency.

7 CONCLUSION

In the current work, we detail the challenges faced in adapting parallel data structures to NDP systems. We ported a real-world data structure, hash-table, to NDP and identified the possible consistency issues and proposed multiple ways to avoid such consistency issues, by bundling NDP instructions and modifying the data structure layout.

REFERENCES

[1] Junwhan Ahn, Sungjoo Yoo, Onur Mutlu, and Kiyoung Choi. 2015. PIM-enabled Instructions: A Low-overhead, Locality-aware Processing-in-memory Architecture. In *Proceedings of the 42Nd Annual International Symposium on Computer Architecture* (Portland, Oregon) *(ISCA '15)*. ACM, New York, NY, USA, 336–348. https://doi.org/10.1145/2749469.2750385

[2] Berkin Akin, Franz Franchetti, and James C Hoe. 2016. HAMLeT architecture for parallel data reorganization in memory. *IEEE Micro* 36, 1 (2016), 14–23.

[3] Farhana Aleen and Nathan Clark. 2009. Commutativity Analysis for Software Parallelization: Letting Program Transformations See the Big Picture. In *Proc. of the Int'l. Conf. on Architectural Support for Programming Languages and Operating Systems*.

[4] Mohammad Alian and Nam Sung Kim. 2019. NetDIMM: Low-Latency Near-Memory Network Interface Architecture. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture* (Columbus, OH, USA) *(MICRO '52)*. Association for Computing Machinery, New York, NY, USA, 699–711. https://doi.org/10.1145/3352460.3358278

[5] Mohammad Alian, Seung Won Min, Hadi Asgharimoghaddam, Ashutosh Dhar, Dong Kai Wang, Thomas Roewer, Adam McPadden, Oliver O'Halloran, Deming Chen, Jinjun Xiong, Daehoon Kim, Wen-mei Hwu, and Nam Sung Kim. 2018. Application-Transparent near-Memory Processing Architecture with Memory Channel Network. In *Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture* (Fukuoka, Japan) *(MICRO-51)*. IEEE Press, 802–814. https://doi.org/10.1109/MICRO.2018.00070

[6] M. A. Z. Alves, M. Diener, P. C. Santos, and L. Carro. 2016. Large vector extensions inside the HMC. In *2016 Design, Automation Test in Europe Conference Exhibition (DATE)*. 1249–1254.

[7] Fabrice Bellard. 2005. QEMU, a fast and portable dynamic translator.. In *USENIX Annual Technical Conference, FREENIX Track*, Vol. 41. 46.

[8] Benjamin Y Cho, Yongkee Kwon, Sangkug Lym, and Mattan Erez. 2020. Near Data Acceleration with Concurrent Host Access. In *Proceedings of the 47th Annual International Symposium on Computer Architecture*. ACM.

[9] Hybrid Memory Cube Consortium. 2014. Hybrid Memory Cube Specification 2.0.

[10] Emilio G. Cota. 2017. Quick Hash Table. https://github.com/pranith/quickht

[11] Fabrice Devaux. 2019. The true Processing In Memory accelerator. In *2019 IEEE Hot Chips 31 Symposium (HCS)*. 1–24.

[12] A. Farmahini-Farahani, J. Ho Ahn, K. Morrow, and N. Sung Kim. 2015. DRAMA: An Architecture for Accelerated Processing Near Memory. *IEEE Computer Architecture Letters* 14, 1 (Jan 2015), 26–29. https://doi.org/10.1109/LCA.2014.2333735

[13] A. Farmahini-Farahani, J. H. Ahn, K. Morrow, and N. S. Kim. 2015. NDA: Near-DRAM acceleration architecture leveraging commodity DRAM devices and standard memory modules. In *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*. 283–295. https://doi.org/10.1109/HPCA.2015.7056040

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

[15] Mingyu Gao, Jing Pu, Xuan Yang, Mark Horowitz, and Christos Kozyrakis. 2017. TETRIS: Scalable and Efficient Neural Network Acceleration with 3D Memory. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems* (China) *(ASPLOS '17)*. Association for Computing Machinery, New York, NY, USA, 751–764. https://doi.org/10.1145/3037697.3037702

[16] Byungchul Hong, Gwangsun Kim, Jung Ho Ahn, Yongkee Kwon, Hongsik Kim, and John Kim. 2016. Accelerating Linked-list Traversal Through Near-Data Processing. In *Proceedings of the 2016 International Conference on Parallel Architectures and Compilation* (Haifa, Israel) *(PACT '16)*. ACM, New York, NY, USA, 113–124. https://doi.org/10.1145/2967938.2967958

[17] Gwangsun Kim, J. Kim, Jung Ho Ahn, and Jaeha Kim. 2013. Memory-centric system interconnect design with Hybrid Memory Cubes. In *Parallel Architectures and Compilation Techniques (PACT), 2013 22nd International Conference on*. 145–155. https://doi.org/10.1109/PACT.2013.6618812

[18] Shuangchen Li, Cong Xu, Qiaosha Zou, Jishen Zhao, Yu Lu, and Yuan Xie. 2016. Pinatubo: A Processing-in-memory Architecture for Bulk Bitwise Operations in Emerging Non-volatile Memories. In *Proceedings of the 53rd Annual Design Automation Conference* (Austin, Texas) *(DAC '16)*. ACM, New York, NY, USA, Article 173, 6 pages. https://doi.org/10.1145/2897937.2898064

[19] Zhiyu Liu, Irina Calciu, Maurice Herlihy, and Onur Mutlu. 2017. Concurrent data structures for near-memory computing. In *Proceedings of the 29th ACM Symposium on Parallelism in Algorithms and Architectures*. ACM, 235–245.

[20] Paul E McKenney. 2017. Sequence Locks. In *Is parallel programming hard, and, if so, what can you do about it?* Chapter 9.4.

[21] Andreas Olofsson, Tomas Nordström, and Zain-ul-Abdin. 2014. Kickstarting High-performance Energy-efficient Manycore Architectures with Epiphany. *CoRR* abs/1412.5538 (2014). http://arxiv.org/abs/1412.5538

[22] Ashutosh Pattnaik, Xulong Tang, Adwait Jog, Onur Kayiran, Asit K. Mishra, Mahmut T. Kandemir, Onur Mutlu, and Chita R. Das. 2016. Scheduling Techniques for GPU Architectures with Processing-In-Memory Capabilities. In *Proceedings of the 2016 International Conference on Parallel Architectures and Compilation* (Haifa, Israel) *(PACT '16)*. ACM, New York, NY, USA, 31–44. https://doi.org/10.1145/2967938.2967940

[23] Hermann Schweizer, Maciej Besta, and Torsten Hoefler. 2015. Evaluating the Cost of Atomic Operations on Modern Architectures. In *Proceedings of the 2015 International Conference on Parallel Architecture and Compilation (PACT) (PACT '15)*. IEEE Computer Society, Washington, DC, USA, 445–456. https://doi.org/10.1109/PACT.2015.24

[24] Amanda K Sharp. 2013. Memory Layout Transformations. (2013). https://software.intel.com/content/www/us/en/develop/articles/memory-layout-transformations.html

[25] Dongping Zhang, Nuwan Jayasena, Alexander Lyashevsky, Joseph L. Greathouse, Lifan Xu, and Michael Ignatowski. 2014. TOP-PIM: Throughput-oriented Programmable Processing in Memory. In *Proceedings of the 23rd International Symposium on High-performance Parallel and Distributed Computing* (Vancouver, BC, Canada) *(HPDC '14)*. ACM, New York, NY, USA, 85–98. https://doi.org/10.1145/2600212.2600213