Accelerating Linked-list Traversal Through Near-Data Processing

Byungchul Hong[†], Gwangsun Kim[†], Jung Ho Ahn[‡], Yongkee Kwon[*], Hongsik Kim[*], John Kim[†]
[†]KAIST [‡]Seoul National University [*]SK Hynix
{kaisthong,gskim,jjk12}@kaist.ac.kr gajh@snu.ac.kr {yongkee.kwon,hongsik1.kim}@sk.com

ABSTRACT

Recent technology advances in memory system design, along with 3D stacking, have made near-data processing (NDP) more feasible to accelerate different workloads. In this work, we explore near-data processing for a fundamental operation – linked-list traversal (LLT). We propose a new NDP architecture that does not change the existing sequential programming model and does not require any modification to the processor microarchitecture. Instead, we exploit the packetized interface between the core and the memory modules to off-load LLT for NDP. We leverage a system with multiple memory modules (e.g., hybrid memory cube (HMC) modules) interconnected with a memory network and our initial evaluation shows that simply off-loading LLT computation to near-memory can actually *reduce* performance because of the additional off-chip memory network channel traversals. Thus, we first propose NDP-aware data localization to exploit locality – including locality within a single memory module and memory vault – to minimize latency and improve energy efficiency. In order to improve overall throughput and maximize parallelism, we propose batching multiple LLT operations together to amortize the cost of NDP by utilizing the highly parallel execution of NDP processing units and the high bandwidth of 3D stacked DRAM. The combination of NDP-aware data localization and batching can provide significant improvement in performance and energy efficiency compared to host-processing.

Keywords

Near-data processing; Processing-in-memory; Big-memory workload; Linked-list traversal

1. INTRODUCTION

With the recent emergence of 3D memory stacking, 3D stacked memory combined with a logic layer has been proposed to create a hybrid memory cube (HMC) [24]. Micron has developed the first generation HMC [43] and Intel has announced a Xeon Phi system that leverages a variation of the HMC [47]. In this work, we exploit the availability of such memory modules; in particular, the logic layer within an HMC provides the opportunity for offloading computation and near-data processing (NDP) to accelerate different workloads. Linked-list traversal (LLT) is a simple, fundamental operation used to search an element in a linked-list, but conventional CPUs are not efficient in executing LLT because its access patterns are mostly sequential and random [33]. Thus, to achieve efficient execution of linked-list traversal, we explore offloading or near-data processing of the linked-list traversal to the HMC logic layer.

A linked-list is a basic data structure used to store data by chaining multiple elements through pointer chasing. An array data structure has several benefits over the linked-list, including a more compact storage, better performance in retrieving data and better locality. However, an array also has some limitations which include difficulty in adding new elements to the middle of the array and difficulty in storing elements with different sizes. In comparison, linked-list has the opposite properties such as faster update but slower retrieval and is more appropriate for dynamic applications which require frequent updates. To combine the advantages of both data structures, different data structures, such as a linked-list of arrays is used in modern applications. In this work, we evaluate the impact of NDP on different linked-list implementations, including the linked-list of arrays.

Accessing the elements stored in the linked-list requires pointer-chasing and is also referred to as linked-list traversal (LLT). LLT is intrinsically a *sequential* operation because of the data dependency to the next address, and also often accesses *random* addresses. Thus, architectural advances in modern processors (e.g., multiple cache hierarchies, MLP, and prefetching[1]) provide limited performance improvement for LLT. In addition, while the computation for LLT is relatively simple, the main-memory latency is critical to the overall performance.

Emerging big memory applications that use linked-list data structures, such as hash table and adjacency list of graphs, use LLT for data retrieval where LLT represents a significant amount of the total execution time [33]. A breakdown of execution time for different workloads is shown in Figure 1, which are measured on Dell PowerEdge R910 server with Intel Xeon E7540 quad-sockets, and 512GB of

PACT '16, September 11-15, 2016, Haifa, Israel
© 2016 ACM. ISBN 978-1-4503-4121-9/16/09…$15.00
DOI: http://dx.doi.org/10.1145/2967938.2967958

[1]Address correlation between the linked items can be stored in on-chip buffer to help prefetching during the LLT [15]. However, the huge size of linked-lists and random access pattern can limit the benefit of such modern prefetching techniques for big-memory workloads.

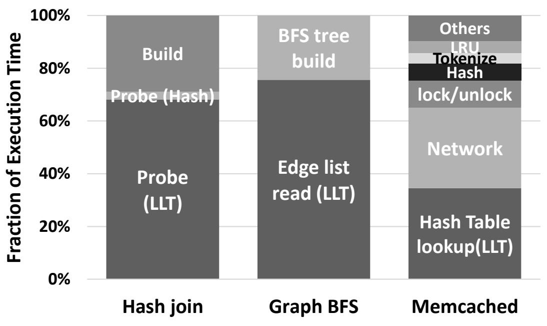


Figure 1: Execution time breakdown of Hash join [8], Graph BFS [39], and Memcached [17] that use LLT.

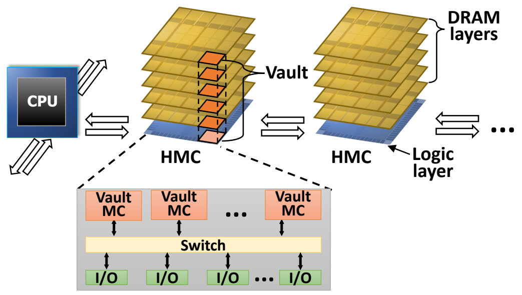


Figure 2: High-level block diagram of hybrid memory cubes (HMCs) and the logic layer, interconnected through memory network.

main memory. Approximately 70-75% of the execution time for Hash join (used by DBMS) [8] and Graph500 BFS [39] and approximately 35% for Memcached [17] is from LLT. In this work, we explore how LLT for these workloads can be accelerated using NDP.

We propose an NDP architecture with multiple memory modules interconnected through a *memory network* [31] to accelerate LLT and exploit the parallelism available across the multiple memory modules. Our approach does not require any modification to the existing sequential programming model and CPU cores. Instead, we exploit the packetized interface between the core and the memory modules to off-load LLT for near-data processing using load/store instructions and simplify the CPU-memory interface. However, we show that simply off-loading LLT does not necessarily improve performance but can actually *decrease* performance because of the additional off-chip channel traversals through the memory network. Thus, we propose NDP-aware data localization to localize data near the NDP execution units to minimize off-chip accesses and reduce LLT latency. In addition, to fully exploit the parallelism available with NDP and improve the LLT throughput, we propose batching of multiple LLT operations within a single packet (and across multiple packets) to improve overall performance and exploit the request-level (or inter-LLT) parallelism.

In particular, the contributions of this work includes the following:

- We propose a near-data processing (NDP) architecture that off-loads linked-list traversal (LLT) to logic near memory using load/store instructions without modifying the existing programming model or core microarchitecture.
- We propose NDP-aware data localization to minimize off-chip accesses and reduce the execution time of LLT.
- We leverage *batching* multiple LLT operations that exploits the inter-LLT parallelism and the parallelism available through the memory network.
- Our evaluation shows that our proposed LLT offloading for NDP can increase performance by 5.9× and energy efficiency by 2.8× compared with host-processing.

2. BACKGROUND AND RELATED WORK

2.1 Hybrid Memory Cube (HMC)

3D integration technology allows the dies fabricated by two or more types of processes to be stacked within a package by TSVs (Through-Silicon Vias). A representative DRAM architecture that exploits the 3D integration technology is HMC (Hybrid Memory Cube) [24], which stacks a logic die and DRAM dies within a package to improve the energy efficiency, bandwidth, and scalability in capacity of main memory systems. Multiple DRAM dies are stacked and divided into vertically aligned partitions that are referred to as *vault* in HMC, as shown in Figure 2. Each vault is connected to the corresponding memory controller located at the logic layer through TSVs. The HMC logic die also has high-speed links through which the memory controllers communicate with other memory modules and the CPU sockets and a switch or an on-chip interconnection network connects these components (the high-speed links and the memory controllers) together. Even with the existence of the links, the controllers, and the interconnect, the logic die has additional area to hold processing elements for acceleration [37, 44]. HMC modules, each becoming a router, can be interconnected together to scale the system and create a memory network [31].

2.2 Linked List

Four different types of linked-list implementation that we consider in this work are shown in Figure 3 and summarized in Table 1. We define an *item* as the data element that is being stored in the linked-list and a *node* as an element that contains the item(s) and a pointer to the next node. The most conventional type of linked-list is shown in Figure 3(a). A single item is stored within each node with a pointer to the next node. Memory efficiency of this type of linked-list can be poor and it also results in a lot of linked-list traversals. However, if an array structure is inappropriate and if neither the size of the items nor the number of items is not known, this data structure can be used. Key-value stores, such as Memcached [17], use this data structure to store various types of data objects.

Figure 3(b) provides a block diagram of linked-list of arrays where each node contains an array of items. Since an array structure is used, the item size needs to be fixed and the items within the node are stored contiguously in memory. In addition, a single pointer is only needed for each array – thus, this implementation reduces the memory requirement and cache locality can be exploited. Hash join [11] is an example that uses this linked-list implementation.

If the total number of items is fixed in addition to the item size, linked-list of arrays can be implemented without a bucket array (Figure 3(c)). Since the pointer reference

Table 1: Different types of linked-list

Type	Item & next ptr.	Item size	# of items	Workloads
1	Unified	Variable	Variable	Memcached [17] LLU [48]
2	Unified	Fixed	Variable	Hash join [11]
3	Unified	Fixed	Fixed	Hash join [8]
4	Separated	-	-	Graph500 [39]

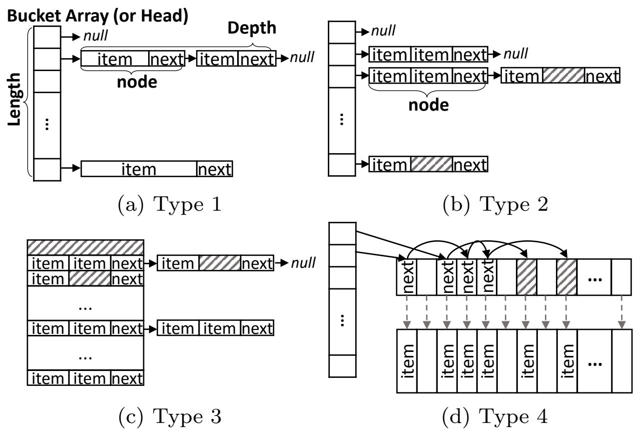


Figure 3: Block diagram of the different linked-list data structures summarized in Table 1.

from the bucket array is removed, reading an item becomes faster; however, unless the index values are uniformly distributed across all of the entries, this approach can waste the memory storage. A different implementation of Hash Join [8] uses this particular data structure type. Linked-list within an array approach is shown in Figure 3(d), where pointers to the next node and items are stored separately. The *List* implementation of Graph500 [39] uses this linked-list type for the adjacency list of graph. While this approach is area-efficient and additional edges can be easily added, significant amount of LLT is required to read the entire adjacency list. In this work, we evaluate our proposed NDP across the different linked-list implementations described in this section.

2.3 Related Work

Near-data processing (NDP) or processing-in-memory (PIM), which tightly integrates functional units with memory units for high performance and energy efficiency, is not new, and has been proposed since the 90s, including FlexRAM [30], IRAM [42], EXECUBE [34], and various smart memory controllers [12, 16, 1]. However, these were not commercialized because DRAM technology did not provide the high performance necessary for the functional units. Recently, NDP has regained interest because of the evolving technology including 3D stacking and big-data applications becoming popular as well as device scaling starting to slow down [6]. As a result, there has been active interest in NDP from the industry and examples include the IBM Active memory cube [41] and the Micron automata processor [14] that can accelerate deep packet inspection and graph analytics. Samsung recently announced a plan to support near data processing of a "search" function in in-memory database [45] by adding FPGA under the stacked DRAM.

NDP with programmable cores: NDC [44] applied NDP to in-memory MapReduce by incorporating energy efficient in-order cores to the logic layer of 3D-stacked memory. Mercury architecture [21] leverages a 3D-stacked memory architecture to improve the physical density and efficiency of key-value store (Memcached) servers. Tesseract [3] accelerates graph workloads by applying processing-in-memory, where in-order core, prefetcher, and message queue are added to each vault, and memory network is used to transfer the data between memory modules. Gao et al. [18] developed an in-memory analytics framework for NDP with a scalable hardware support and run-time system. Kim et al. [32] showed that misses per kilo instructions (MPKI) is one of the most important application characteristics to determine applying near data processing.

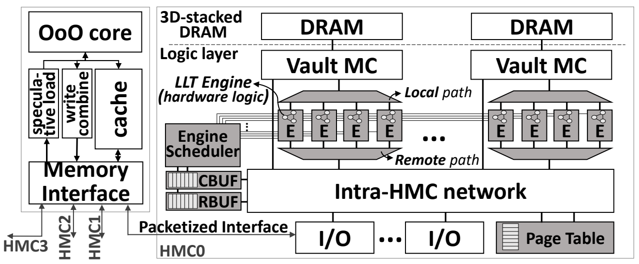
NDP with fixed-function/configurable logics: Guo et al. [20] integrated configurable array of hardware logics to an accelerator layer of 3D-stacked memory to accelerate FFT and reshape of data layout while software controls the execution paths of the accelerators. Akin et al. [4] proposed near data processing of data reorganization operations, such as shuffle, pack/unpack, and swap with mathematical framework. Gao et al. [19] proposed heterogeneous reconfigurable logic as near data processing units to increase power and area efficiency. BSSync [35] proposed a hardware support in near memory logic layer to minimize the overhead of atomic operations for machine learning workloads while Nai et al. [40] showed the feasibility of instruction-level offloading for the HMC 2.0 [24] with graph traversal workload.

This work: Our work can be classified as "Fixed-function, Compound PIM operations" based on the taxonomy from Loh et al. [38] as the number of memory accesses is decided by the depth of linked-list. We implement near data processing unit as a fixed-function hardware logic for energy-efficient parallel execution, but with control determined from the off-loading command packet. Compared to prior work on NDP, we provide fine-grained offloading of LLT to NDP through efficient communication interface and exploit the parallelism available through the memory network. Prior work [33] also proposed to accelerate LLT; however, they focused on accelerating LLT for host-processing while this work accelerates LLT with NDP and thus, enables memory access latency to be significantly reduced.

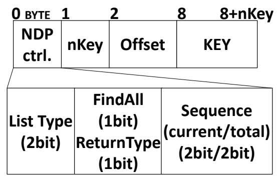
3. NEAR-DATA PROCESSING ARCHITECTURE

Since LLT often has random memory access patterns, on-chip resources in modern CPUs do not necessarily benefit LLT. For example, the probability that the next item of the hash chain in Memcached is read from the CPU on-chip LLC is only 0.56% to 1.2%.[2] Request-level locality (e.g., hot keys) did not increase on-chip cache locality because of the large data set. Thus, we propose to move the execution of LLT closer to the data and reduce LLT latency by minimizing off-chip accesses and improve parallel execution and energy efficiency through dedicated hardware logic or *LLT engine.* The LLT engine that we propose supports all four types of linked-list described earlier in Section 2.2. Throughout this work, we use the term *storage memory* to refer to the memory allocated for the large data set such as a linked-list or a hash table.

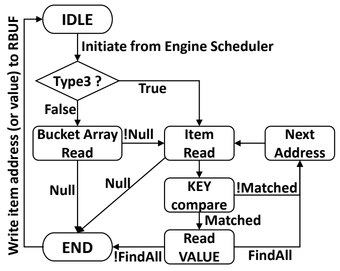
[2]Simulation setup used is described in Section 5.2.



(a) HMC logic layer for near data processing of LLT



(b) command packet format



(c) Simplified LLT engine FSM

Figure 4: (a) HMC logic diagram with additional logics for LLT NDP shaded with grey color, (b) offloading command packet format, and (c) simplified finite state machine of the LLT engine.

3.1 NDP Offloading Interface

Without modifying the core architecture, we exploit the packetized interface between the core and the memory modules such that the host CPU communicates with near-data logic through normal load/store instructions. NDP commands/results are communicated through the payload of memory packets. High-level overview of the proposed NDP architecture is shown in Figure 4(a) with the highlighted blocks representing additional logic added to the logic layer, including the command and return buffers (CBUF, RBUF), page table, LLT engines, and LLT engine scheduler. To take advantage of the parallelism with NDP, we assume at least one LLT engine per vault but also later evaluate the impact of increasing the number of LLT engines to further increase parallelism in Section 5.3. To offload LLT operations, the host CPU writes an off-loading command to the command buffer (CBUF) in the target HMC module by transmitting a write packet to the destination memory module. The result or the output of the command is stored in the return buffer (RBUF) and is returned back to the core through a read response packet when host reads the corresponding return buffer. CBUF and RBUF, referred to as *communication* buffers in this work, are divided into 64-Byte entries, same as the memory packet size. Each CBUF entry is mapped to a single RBUF entry, and thus, the result of LLT operation off-loaded to CBUF[n] can be obtained from the RBUF[n], where n is the communication buffer entry ID.

The offloading command packet (Figure 4(b)) consists of control bits (`NDP ctrl`), size of the key (`nKey`), offset of LLT start address from the base address of the storage memory (`Offset`), and the input key to be searched (`Key`). The `Key` represents the identification of an item to be compared when retrieving a data (or the item) from the linked-list. The `Key` can have variable data types such as a variable-length string type in Memcached or a fixed-length integer type in Hash Join. If `Key` is not used during the LLT (e.g., Graph BFS), `nKey` field in the command packet is set to 0 and all of the values in the linked-list are read.

We assume that packet payload size is fixed to 64B, and if the size of a key is small, multiple offloading commands can be combined into a single command packet through *batching* we describe later in Section 4.2. However, for large keys (e.g., Memcached can have up to 250B size key), the large key is sent across multiple packets. The `sequence` information within the control bits indicates packet ordering, and is used to re-order the packets if they arrive out-of-order. NDP control bits also include information on the linked-list type (`List Type`), and NDP execution control information (`FindAll`, `ReturnType`). `FindAll` determines whether to find all matches in the linked-list or a single first match, and `ReturnType` determines whether to return the address of the value (used for large values) or the value itself.

When an off-loading command is written to CBUF, engine scheduler decodes the command and triggers one of the available LLT engines connected to the target vault that is determined by the `offset` field. LLT engine calculates the start address of the linked-list to be accessed based on the `offset` and the base address[3], and translates it to physical address through the page table (described in Section 3.2). LLT engine then starts pointer-chasing, item format decoding, and key comparison as shown in Figure 4(c), and retrieved results are stored to the return buffer (RBUF). After host CPU writes the off-loading command to CBUF, host CPU reads the corresponding RBUF address by sending a read packet to obtain the NDP result when available. Host usually waits in idle state until the RBUF response arrives as the instructions after LLT may have data dependency on the LLT result and can cause performance degradation. In Section 4.2, we describe how this performance bottleneck can be removed by applying batching.

For safe offloading communication with multiple host CPU threads, it must be guaranteed that multiple threads do not offload to the same entry of the communication buffers. If a fine-grain bucket lock is acquired before LLT operation (e.g., Memcached), the communication buffer entry can be determined based on the bucket lock index and the lock mechanism will protect concurrent access from multiple threads. For workloads that do not acquire the bucket lock, communication buffer is divided by the number of host threads and each thread uses its own entries.

Table 2: Memory type of different memory regions

Region	Memory type	Used for
Storage memory	1GB huge page, Locked, Write-through cache	Linked-lists
Communi -cation	Memory mapped I/O, Uncacheable & write-combine	CBUF/RBUF Page table
Others	4KB page, Write-back cache	Stack, Heap

3.2 Address translation and Memory types

In our proposed NDP system, one of the objective is to enable a system that can work with existing CPU architectures with minimal modifications. For LLT, one challenge is address translation since linked-list requires accessing mul-

[3]Static information, such as base address of the storage memory and data structure formats (number of items in a node, byte position of item/next) are written to the configuration registers during initialization phase of workload to minimize the size of the off-loading command.

tiple memory addresses that is not known in advance. The next address stored in the linked-list is a virtual address (VA) since the host CPU builds the linked-list and both NDP and host CPU access the linked-list. The next address can be stored as physical address (PA) to simplify the LLT access, but this would require non-trivial changes to the memory management of the host CPU. As a result, we take advantage of two features that are commonly available in modern systems, huge pages and memory locking, to simplify address translation in our proposed NDP architecture.

LLT engines translate address through the page table that is duplicated within each memory module in the logic layer (Figure 4). To reduce the cost of the duplicated page table, we exploit huge pages to keep the page table size small.[4] For in-memory applications, storage memory is rarely swapped out to the lower-level storage once it is allocated [9]. Therefore, storage memory region can be locked (i.e., `mlock`) to maintain consistent mapping from VA to PA.[5] The page table in near-data logic is managed by the host, and host updates the page table upon the (re)allocation of the storage memory.

Communication buffers use memory mapped I/O and uncacheable region. We assume uncacheable memory similar to the x86 architecture which has a special memory type, called uncacheable speculative write combine (USWC), to enable fast read/write to the uncacheable region [25]. Modern OS and x86 support direct access to memory mapped I/O space in USWC region from the user-level application [25] without OS intervention at run-time. Applications access the communication buffers through load/store instructions, such as `_mm_stream(_load)_si128` in x86 with SIMD registers. If multiple stores to the USWC region have consecutive addresses, they are combined in a write combining buffer until the buffer becomes full or the next store does not have a consecutive address. The write combining buffer sends out multiple stores to memory as a single request packet. In case of read, load instruction (with hint) to the USWC region fills 64 bytes into a speculative load buffer, and subsequent loads to the consecutive addresses are returned from the speculative load buffer. To utilize those architectural optimizations, we change the order of communication buffer accesses as the subsequent accesses have consecutive addresses when sending multiple commands concurrently.

3.3 Cache coherence and memory consistency

Table 2 summarizes the different memory type of each memory region. Storage memory is the only region accessed by NDP, and thus we need to consider cache coherence and memory consistency for this region. Host does both *read* and *write* to the storage memory region, but NDP only does *read* since NDP writes the LLT results to the return buffer and host updates the storage memory with the results if necessary. Memory model should guarantee that the updated data (modified by host) should be visible to NDP units before NDP starts LLT.

[4]Huge-page allocation may have issues if memory is fragmented. However, storage memory is allocated at the application start-up [9] and big memory workloads are usually long running programs (e.g., web server, database), making the huge-page allocation overhead at start-up ignorable.

[5]Since `mlock` does not guarantee consistent mapping to the same physical address if OS migrates physical pages, pages can be pinned (e.g., mm_mpin()) to ensure the consistent mapping. However, to the best of our knowledge, Linux kernels try to avoid migrating huge pages because of its overhead.

To maintain memory consistency, a write-through (WT) cache is used for the storage memory region to ensure that the host writes back the updated data to the main memory without adding hardware cache coherence protocol to NDP. WT cache can cause significant memory traffic, but WT cache in our work is different as only the storage memory uses the write-through policy. In comparison, normal stack/heap regions (which may have high temporal locality) still use write-back (WB) caching. In the workloads that we evaluate, storage memory region is fairly read-intensive as write traffic represents less than 10% of the total memory access – e.g., GET dominant Memcached operations, longer Probe phase than Build phase for Hashjoin/Graph. NDP units can be made as cache coherent devices (or coherent I/O devices [27]) with WB cache for storage memory, and NDP units can send coherent read requests to host. However, this creates significant amount of coherent requests to the host chip, and even with a best case scenario (i.e., data fetched from host on-chip cache), host on-chip cache is not necessarily closer to NDP units compared to the local 3D-stacked memory.

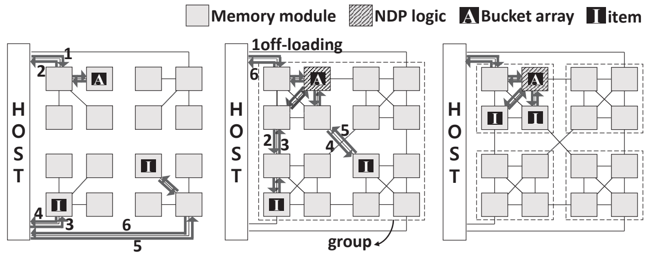
To guarantee the write-back of modified data to main memory before NDP starts LLT, proper synchronization is necessary between the data update and the LLT execution. For Graph search and Hash Join, there is a barrier synchronization between the write phase and read phase that protects the concurrent read/write accesses to the whole storage memory (e.g., Build phase and Probe phase of Hash join). Updated data in write-through region are written-back to main memory at the barrier. For Memcached, concurrent read/write accesses are protected by fine-grain (per bucket) lock – e.g., for two threads (T0, T1) accessing the same bucket,

- T0 : Acquire lock[bucket] => Linked-list update (write by host) => Release lock[bucket]
- T1 : Acquire lock[bucket] => Offloading to NDP => NDP reads linked-list => Release lock[bucket]

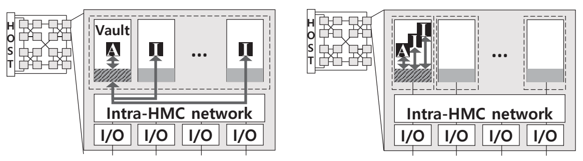
Through acquiring the lock that incorporates memory barrier, the memory consistency model is not impacted as offloading cannot precede the lock acquisition in T1. In addition, the updated value by T0 needs to be written back to the memory before NDP reads the linked-list. Thus, the write-back to memory needs to be done on the lock release for the write-through memory region, which is done for the write-combined (WC) and write-through (WT) memory types in x86 [26].

4. NEAR-DATA PROCESSING OF LINKED-LIST TRAVERSAL (LLT)

An example of LLT with host-processing (HSP) and LLT with NDP is shown in Figure 5(a),(b). We assume a star-topology for HSP to minimize diameter while we assume a distributed dragonfly topology [31] for the memory network with NDP. In this example, HSP requires 10 hop count (or network channel traversal) to complete a single LLT with two items shown in the example while NDP requires 14 hop count. As a result, simply off-loading LLT does not necessarily improve LLT performance. In this section, we first describe NDP-aware data localization to increase locality and



(a) host-processing (b) NDP ($M = 1$) (c) NDP ($M = 4$)



(d) NDP($M = 16$) (e) NDP ($M = 256$)

Figure 5: Data access in host-processing and NDP with different localization degrees.

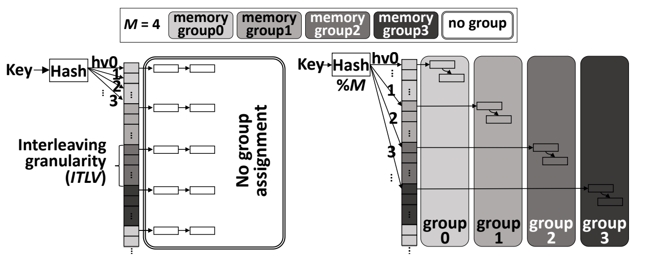
minimize the memory network hop count. We then propose *batching* multiple LLT operations together to exploit the NDP parallelism available within the logic layer of the memory module.

4.1 NDP-aware data localization

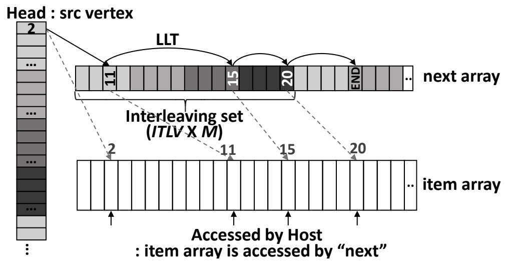
In this section, the number of *memory groups* (M) is defined as the number of main memory partitions and represents the degree of locality in the NDP system. Our baseline NDP for LLT is shown in Figure 5(b) without any localization ($M = 1$). As M increases, the degree of localization increases – for example, with $M = 16$ (and 16 memory modules in Figure 5(d)), the size of a memory group is a single memory module and off-chip memory network traversal is minimized; however, there can be accesses to different vaults within the memory module. If M is further increased to 256 (assuming 16 vaults within each memory module), the locality is further confined to each vault and minimize movement within a memory module as well. While increasing M improves locality, it reduces the amount of memory capacity per memory group and trade-off is discussed later in Section 5.3.

In our proposed NDP system, the unit of off-loading is a single linked-list traversal. For NDP-aware data localization, we store the linked-list to "physically neighboring" memory.[6] To achieve this locality, we modify the application and memory allocation function. We first design a memory manager that allocates *storage memory* that is divided into *partitions* with each partition assigned to one of the memory groups based on the physical address of the partition. In a fine-grain interleaved multi-channel memory system that we assume, the partitioning size should be smaller than the address interleaving granularity ($ITLV$) to make sure each partition is not interleaved across non-neighboring physical memory. We achieve data localization by mapping each memory group to physically neighboring memory (managed by OS and the memory manager) similar to NUMA-aware memory allocation, and by storing a linked-list to a single memory group (managed by application).

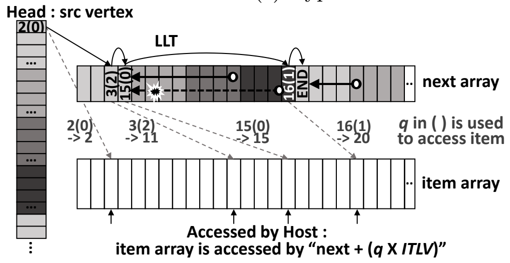
[6]Depending on the value of M, the linked-list can be stored in the same memory vault, memory module, or neighboring memory modules.



(a) Type 1,2,3 (b) Localization of Type 1,2,3


Head : src vertex
LLT
next array
Interleaving set (ITLV X M)
item array
Accessed by Host : item array is accessed by "next"

(c) Type 4

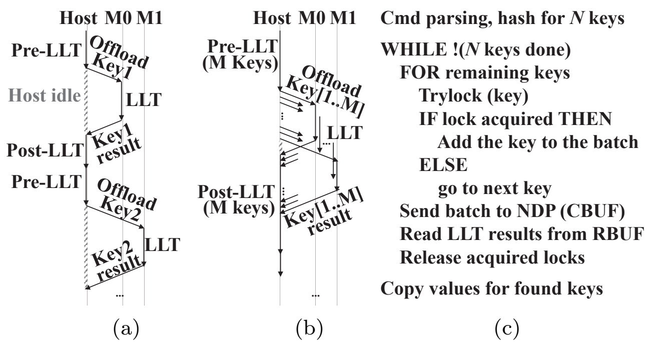

Head : src vertex
LLT
next array
q in ( ) is used to access item
item array
Accessed by Host : item array is accessed by "next + (q X ITLV)"

(d) Localization of Type 4

Figure 6: NDP-aware data localization with 4 memory groups for the four different linked-list types.

Figure 6(a,b) show how to localize the linked-list Type 1, 2, and 3. The location of linked-list (i.e., index) is decided by index calculation, such as hashing. Items having hash value (hv) n are normally stored to the nth linked-list as shown in Figure 6(a). However, we change this direct mapping as items having index n are stored to a memory group m using a modulo operation $m = n \ mod \ M$, as shown in Figure 6(b). When application requests memory allocation to store an item, the index value n is given together as a parameter to the memory manager. The memory manager returns a memory address from memory group m, and the item is stored in that particular memory group.

Figure 6(c,d) show how to localize the linked-list Type 4, which stores next pointer and item in different arrays – next array and item array. In Graph500 [39], adjacency list is implemented as a linked-list within the next array, and item array stores the edge information. To localize the linked-list on the next array for NDP, we relocate the location (i.e., index) of the next pointers *within* an interleaving set, which is defined as '$ITLV \times M$'. Next pointer connected to source vertex n, where $n \ mod \ M = m$, is attempted to be relocated to a specific index allocated in the memory group m. For example, when edge x is connected to the source vertex n, where x has an index distance '$q \times ITLV + r$' from the boundary of the interleaving set, x is stored in the next array and next[x] again stores the next connected edge, as shown in Figure 6(c). To localize for NDP, we try to move next[x] to next[x'], where x' has an index distance



```
Cmd parsing, hash for N keys
WHILE !(N keys done)
    FOR remaining keys
        Trylock (key)
        IF lock acquired THEN
            Add the key to the batch
        ELSE
            go to next key
    Send batch to NDP (CBUF)
    Read LLT results from RBUF
    Release acquired locks
Copy values for found keys
```

Figure 7: NDP (a) without batching and (b) with batching, and (c) support for Memcached batching for GET sequence with trylock.

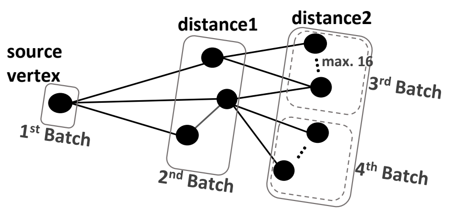


Figure 8: Batching in graph: vertexes connected at the same distance from the source vertex.

'$m \times ITLV + r$' from the boundary of the interleaving set and next[x'] resides in the memory group m. However, item[x] needs to be accessed by the host to get the edge information, and thus the information that can restore the original index x needs to be stored in somewhere. To minimize the required bits to restore the original index, we keep r consistent but only change q in the relocation sequence and q is stored in the MSB of the next pointer.[7] By limiting the range of relocation to inside an interleaving set (i.e., $q < M$), we need $\log_2 M$ bits to store q. Since there are several unused bits in the next pointer (to align the address of array elements with multiple bytes), storing q does not require additional memory capacity.

Relocation can fail or result in a *conflict* if the new index is already occupied by another next pointer that has been relocated. In Figure 6(d), next[15] is attempted to be relocated to next[3], but fails since next[11] is already relocated to next[3]. Note that index 11 and 15 have the same remainder(3) when divided by the $ITLV(4)$. Since conflicts can cause access to remote memory module in NDP, the neighboring memory groups that are mapped physically closer (e.g., remote vaults within the same HMC module when the memory group is a single vault) are used to relocate to minimize off-chip access in LLT. In our experiments with 16 HMC modules and 16 vaults per HMC module, 50.2%/38.8%/11.0% of next pointers are stored to local vault/local HMC/remote HMC modules, respectively.

4.2 Batching multiple LLT operations

While NDP-aware data localization reduces LLT latency (and reduce off-chip memory bandwidth) and improves overall performance compared to host-processing, the parallelism available within NDP is not fully exploited unless the throughput of LLT offloading is increased. As shown in Figure 7(a), the host and NDP work sequentially; however, by exploiting the request-level parallelism (i.e., independent LLTs), overall throughput (and performance) can be significantly improved. Thus, we propose *batching* of multiple or independent LLTs to be off-loaded together to enable highly parallel execution in NDP. Prior work [22] also utilized batching of Memcached GET requests for GPGPU offloading. However, because of the thousands of threads in GPGPUs, the offloading had high overhead as it required batching more than thousands of requests to hide the communication overhead. In comparison, since the overhead for NDP-offloading is relatively low (normal load/store operations to memory modules), tens of requests are sufficient to maximize performance (in our evaluation, the optimal batch size was approximately 32 to 64).

One challenge with batching is to identify the independent LLTs that can be off-loaded together and executed in parallel. Data dependency exists among the accesses within an LLT, but there is usually no dependency among the LLTs (i.e., inter-LLT). For example, probing of a single tuple in Hash join is translated to a single LLT operation, but probing a tuple is independent of probing of other tuples – thus, multiple tuples can be probed in parallel. In Memcached, a GET request initiates a single LLT operation but is independent of other GET requests. In graph search, the neighboring list of multiple vertexes connected at the same distance from the source vertex can be searched independently and in parallel (Figure 8). Based on these parallelism, multiple LLT operations can be offloaded to NDP as a batch (Figure 7(b)). If the key size is small, multiple LLT operations off-loaded to the same memory module can be merged into a single off-loading command packet to further reduce the off-chip accesses (e.g., with 8B keys, 4 LLTs can be sent within a single 64B command packet). Otherwise, the LLT operations within a batch are off-loaded as multiple packets to the memory modules. As we show later in Section 5.3, to fully realize the benefit of batching, multiple LLT engines are needed within each vault of the HMC to maximize the parallelism.

Another challenge of batching is handling lock acquisitions. For the applications acquiring bucket lock before starting an LLT, such as Memcached, batching can result in significant overhead in terms of lock contention. To reduce lock contention with batching, we use *trylock*, or a non-blocking attempt to acquire a lock, as shown in Figure 7(c). To batch multiple keys, trylock is executed once per key and proceeds to the next key even if the lock is not acquired. After trying to obtain the lock for all of the keys remaining in the batch once, only the keys for which bucket locks are acquired are off-loaded to NDP in a batch, and this sequence is iterated for the remaining keys.

5. EVALUATION

5.1 Workloads

The proposed NDP to accelerate LLT is evaluated with the four workloads summarized in Table 3.[8] Except for Memcached, the batch sizes were empirically determined

[7]When host accesses item array, the location (or index) in the item array is decided by 'x' & $MASK + q \times ITLV$'.

[8]In our evaluation, we did not evaluate Type 2 linked-list (Hash Join [11]) since the Hash Join [8] with Type3 was shown to be more optimized for host-processing.

Table 3: Workload description

Workload	Configuration	Batch size
LLU [48]	LLU-d2: Length 32M Depth 2, Item 16B (Value 4B)	64(8)
	LLU-d4: Length 32M Depth 4, Item 16B (Value 4B)	64(4)
Graph500 [39]	R-MAT, 8M vertex, 128M edges Adjacency list: List implement	16(1)
Hash Join [8]	Probe phase of optimized no-partitioning Hash Join. Tables with 16M,128M tuples Key 8B, Payload 8B	64(4)
Memcached [17]	MC(ETC): 16GB hash table Key 16-120B, Value 1B-1MB, Zipf	24(1)
	MC(FIX): 16GB hash table Key 32B, Value 32B, Zipf	24(1)

and the batch sizes with the best performance were used. The number inside the parenthesis represents the maximum number of LLT operations that can be sent together in a single off-loading command packet, which is decided by the size of off-loading command (or return value size), assuming 64B packets.

LLU benchmark [48] is a synthetic workload which consists of only linked-list traversals (LLTs) with parameterized list length and depth. We use LLU benchmark to understand the impact of LLT exclusively with NDP since other workloads have other operations that are executed by the host. We evaluate LLU with two different depths (2 and 4). In LLU benchmark, only offset (i.e., index of the linked-list to be searched) is sent as NDP command (nKey=0) and the values of the linked items are returned to the host as the NDP output.

Hash Join is a join method used by database management systems (DBMS) and consists of two phases. The *Build* phase consists of creating an intermediate hash table based on the smaller table that is being combined. The *Probe* phase consists of hashing and LLT based on each tuple in the larger table. We implement the *Probe* phase with NDP by batching 64 tuples and 4 off-loading commands can be sent in a single command packet with 8B key size. Payloads (or values) of the matched keys are returned as NDP results.

Breadth-first search (BFS) is the main kernel of Graph 500 [39] benchmark and the *list* implementation uses linked-list within an array (Type 4) to implement the adjacency list of the graph. Searching the adjacent vertexes in BFS is composed of LLT within a next pointer array to retrieve the connected edges and then accesses to the item array as shown in Figure 6(c),(d). Traversing the adjacency list array is off-loaded to NDP, and offset of the first entry in the adjacency list array is sent as offset in the off-loading command. The connected next pointers (or the adjacency list of the source vertex) are returned to host as the NDP result, and host accesses to the item array based on the returned next pointers. Batching for graph BFS described in Section 4.2 is used.

Memcached [17] is an in-memory key-value store used as a cache layer in data centers and consists of GET/SET commands. Memcached uses a hash table as a storage memory and uses Type 1 linked-list to store various sizes of items. GET operation is composed of hashing (for indexing) and LLT in the hash table. Offset in the bucket array (decided by hash value) and input key to be searched are sent as

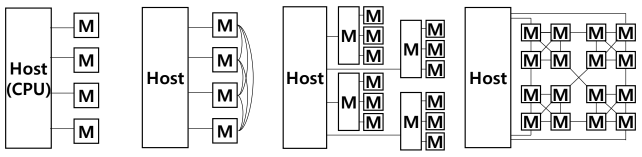


Figure 9: Different topologies with a single host and (a,b) 4 and (c,d) 16 memory modules.

Table 4: Simulator configuration

	Parameter	Configuration
CPU	Core	32 OoO cores @ 3.2GHz Issue width: 4, ROB size: 64
	L1 I/D cache	32KB, 4way, 1cycle latency
	L2 cache	Shared 16MB, 16way, 10cycle
	Cache line size	64B
Memory	HMC org.	8 layers, 16 vaults 16 banks per vault
	HMC capacity	4 GB
	scheduler	FR-FCFS

	DRAM timings	tCK=1.25ns, tWR=12, tCCD=4,
		tRCD=CL=tRP=11, tRAS=22

Network	Router clock	1.0GHz
	Inter-chip link	16 lanes per direction 12.5 Gbps per lane
	Topology	host-processing: Star NDP: FBFLY, DFLY
	Routing	Minimal

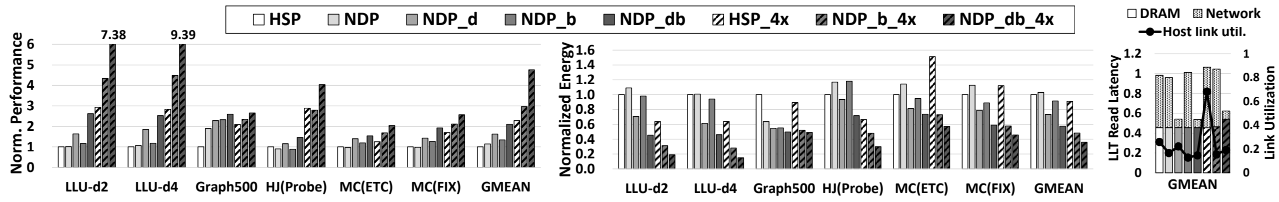
off-loading command, and address of the Value (in case of 'Hit') or Null (in case of 'Miss') is returned as the result from NDP because value size can be larger than available return buffer size. Two different client models (ETC and FIX) [5] are used. Since standard Memcached has limited scalability from global locks, we optimize Memcached by removing all global locks and implementing fine-grain locks [23].[9]

5.2 Methodology

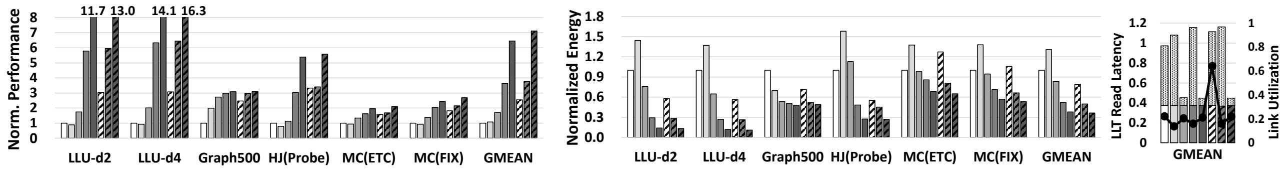
We evaluated the impact of NDP using a detailed cycle-accurate simulator with the configurations summarized in Table 4. To simulate the big-memory workloads with reasonable simulation speed, we used the gem5 [10] simulator but replaced the front end (e.g., the core model) with the trace-based McSimA+ [2] simulator. A cycle-accurate network simulator, Booksim [29], was used to model both the inter-chip network between the CPU and the HMC modules and the intra-chip network within the HMC modules.

We evaluated NDP systems with 4 and 16 memory modules or HMCs – connected to a single multi-core host CPU, and we assumed that each chip has four inter-chip channels. In host-processing, we configured the inter-chip network (or memory network) with Star topology (Figure 9(a),(c)) since only the host accesses the memory modules and no connection between HMC memory modules is needed. For NDP, we configured the memory network as flattened butterfly (FBFLY) for the 4HMC system and 4-way distributed dragonfly (DFLY) topology for the 16HMC system [31] to reduce the NDP traffic latency (Figure 9(b),(d)). Each link or channel is composed of 16 lanes and 12.5 Gbps per lane, and SerDes latency assumed to be 5ns per hop (2.5ns for serialization and 2.5ns for deserialization). For intra-HMC net-

[9]The optimized baseline for host-processing (HSP) used is 7.2× faster than the standard Memcached 1.4.20 version.



(a) Normalized performance, energy and LLT latency at 4 HMC



(b) Normalized performance, energy and LLT latency at 16 HMC

Figure 10: Normalized performance, energy consumption, average LLT data read latency, and CPU-memory link utilization of host-processing (HSP) and NDP by applying proposed optimizations one by one.

Table 5: Evaluated processing configurations

Abbr.	System configuration
HSP	Baseline host-processing
NDP	Near-data processing with LLT offloading
NDP_d	NDP with data locality
NDP_b	NDP with batching
NDP_db	NDP with data locality and batching
HSP_4×	HSP with 4× processing (i.e., 128 threads)
NDP_b_4×	NDP_b with 4 engines per vault
NDP_db_4×	NDP_db with 4 engines per vault

work, we assumed 5-way concentrated mesh [7], where four vault controllers and a single I/O are connected to a router in each concentration with 1ns router delay and 1ns wire delay for inter-concentration transfer. Each vault within the logic layer has its own memory scheduler, and address map is composed as (Row: Vault: Cube: Column: Layer: Offset). DRAM timing parameters are based on Micron DDR3-1600 1Gbit DRAM specification. The logic layer of HMC (with all additional logics for NDP) is implemented at the Booksim network interface, and NDP logic operates at the same frequency as the network router (1GHz).

For energy estimation, we used McPAT [36] to measure CPU power and CACTI-3DD [13] for 3D stacked DRAM. For link energy, we used the model described in [31], but the per bit energy value is adjusted based on the Micron study [28, 46] – using $4.47pJ/bit$ for real packet and $3.35pJ/bit$ for idle packet. To estimate the power consumption of the additional logics for NDP, CACTI was used for SRAM power and we synthesized RTL and estimated the hardware logic power based on the synthesized area (1.7mW per engine).

5.3 Results

The different configurations under evaluation are summarized in Table 5. The results comparing performance (1/execution time) and energy consumption are shown in Figure 10, with the results normalized to host-processing (HSP). Both the HSP and NDP assumed 32 host threads as the configuration was the most energy-efficient for HSP. Except for Graph500, simply offloading LLT (NDP) does not result in significant improvement in performance and for some workloads, the performance actually degrades slightly. For the 16-HMC system, the LLT read latency *increases* by 10.7%, as the average hop count between memory modules through memory network (2.1) is larger than the average hop count between the host and memory modules with a host-centric star topology (1.75). Off-loading overhead (offload command building, communication buffer access) also can increase overall LLT execution time, and thus, there is minimal benefit from LLT offloading. Energy consumption also increases in NDP from the higher hop-count and the additional static energy consumed in the high-speed interchip links. Among the workloads evaluated, only Graph500 resulted in performance improvement from the baseline offloading (NDP). In host-processing, host accesses the next pointer (i.e., edge index) and item (i.e., edge information) sequentially. However, because of the particular linked-list type (i.e., Type 4), we modified the algorithm such that multiple next pointers are retrieved by a single off-loading with NDP, and the host can access multiple items in parallel since the locations are already identified and results in performance improvement.

Impact of data localization: With data localization (NDP_d in Figure 10) to each vault ($M = 16$ for 4-HMC systems and $M = 256$ for 16-HMC systems), performance is increased by 63% (72%) for the 4 (16) HMC systems since the LLT read latency is reduced by minimizing the off-chip accesses and intra-HMC network traversals. For LLU synthetic workloads, NDP_d improves performance by ~2× but for other workloads such as Hash Join with small list depth, the performance benefit from localization is limited. As the depth of linked-list decreases, the impact of LLT latency is minimized. Type2 and Type3 linked-lists are better suited for host-processing since multiple items are stored in the contiguous memory (i.e., array), and multiple items can be fetched to host with a single memory access. To increase performance and energy efficiency for these workloads, increasing throughput (parallelism) is necessary and can be achieved through batching. However, note that the benefit of batching is not fully realized without data localization.

Impact of batching: The addition of batching (NDP_db in Figure 10) increases overall throughput and provides significant performance improvement especially for larger network size. For the 16-HMC system, NDP_db increases performance by 6.7× compared with NDP_d for LLU-d2 and by 4.8× for Hash Join. The latency for individual LLT does not change but the overall throughput increases significantly and results in performance improvement. For the small system configuration (4-HMC), the benefit is still limited in `NDP_db` because of the small number of LLT engines in the overall system. For example, with the 4-HMC system with 16 vaults in each HMC, there is a total of only 64 LLT engines and it is not necessarily sufficient to exploit the parallelism. NDP reduces the host link bandwidth utilization (or CPU-memory data transfer) by moving the computation to near-data logic; batching further reduces host link utilization by sending/receiving multiple NDP commands/results in a single memory packet.

Number of LLT engines: With the parallelism available through the increased number of LLT engines from 1 to 4 (`NDP_db_4`×), there is an additional 2.3× performance improvement over `NDP_db` in a 4-HMC system. `NDP_db_4`× increases the cost slightly at the logic layer but more importantly there is significant energy savings.[10] For the 16-HMC system, the benefit from increased number of LLT engines is smaller (on average 11%) since even with 1 LLT engine per vault, there is already 256 LLT engines in `NDP_db` and provides sufficient parallelism. The benefit of batching is not fully realized without localization as `NDP_b` and `NDP_b_4`× shows the results of using only batching. The impact of localization at higher throughput (`NDP_b_4`× vs. `NDP_db_4`×) is slightly reduced compared to the impact at lower throughput (`NDP_b` vs. `NDP_db`), but localization still contributes 46% and 37% performance benefits of `NDP_db` and `NDP_db_4`×. Overall, Graph500 and Memcached showed smaller overall performance improvement since LLT represented relatively smaller fraction of the overall workload execution time.

To provide a comparison against a prior work with *host*-side LLT acceleration [33], we compare against `HSP_4`× where we assume 4× acceleration is done on the host, similar to what was proposed by the prior work [33]. Instead of modeling dedicated logic, we assumed additional cores were available and increased the number of CPU threads to 128.[11] As shown in Figure 10, `HSP_4`× improves performance by 2.3× over the HSP and by 2.1× over the baseline NDP. However, `NDP_db_4`× exceeds the performance of `HSP_4`× by 2.1× and 2.8× for the 4-HMC and 16-HMC system since `HSP_4`× does not reduce the off-chip memory access latency and bandwidth.

Probe phase of Hash Join and LLU benchmark can be parallelized without any synchronizations among the threads. However, the performance scaling of `HSP_4`× is limited also for LLU and Hash Join. Even though Hash Join has a data structure well suited for host-processing, there is little locality in inter-LLT data accesses because hash function usually decides the address of the linked-list and the addresses are randomly distributed to the giant hash table. Thus, most linked-list traversal creates off-chip accesses, which increases bandwidth requirement as the number of processing units increases.

[10]Our RTL implementation of LLT engine at TSMC 40nm showed 0.0049mm^2 area. Total area of LLT engines with 16 vaults and 4 engines per vault has 0.31mm^2 or 0.31% area of the HMC logic die (assumed 100mm^2), which has negligible area overhead.

[11]Performance of `HSP_4`× is measured with 128 OoO cores, but power consumption was calculated with 32 OoO cores + 96 in-order cores to mimic the specialized logics added to host side, similar to the prior work [33].

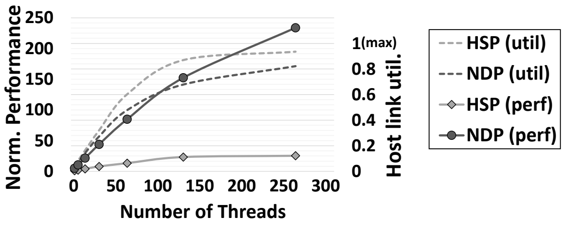
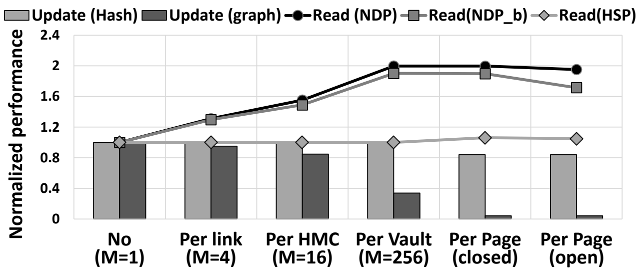
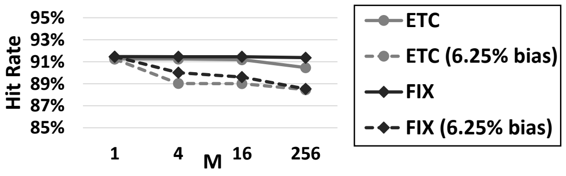


Figure 11: Scalability of HSP and NDP for Hash Join (Probe) workload – Performance normalized to HSP 4 threads.



(a) Linked-list read and write performance



(b) Memcached Hit-rate

Figure 12: Trade-off with different data localization.

Scalability of HSP and NDP: The scalability of HSP and NDP is shown in Figure 11 as the number of host threads increases, illustrating both performance and host-link bandwidth utilization. In this work, we increase the number of processor cores to scale the number of host threads. As the number of threads increases, the host link bandwidth becomes the bottleneck and beyond 128 threads, the performance of HSP saturates; however, NDP continues to scale up to 256 threads. With NDP, the increase in the number of host threads better exploits the parallelism in NDP while the data localization reduces the memory network bandwidth utilization to improve scalability.

Trade-off with localization: Higher degree of data locality (i.e., higher M) can improve performance but it also decreases the *capacity* per memory group, and update performance can be degraded by the non-uniform utilization of the capacity across the memory groups. Figure 12 shows the trade-off of localization including DRAM page level locality (assumed 2KB page size). Vault controller is configured to closed-page mode except 'per page(open)' configuration. `NDP` read performance increases as the degree of localization increases by the reduced inter-chip and intra-chip network traversals. With the addition of batching (`NDP_b`), read performance is still largely affected by the degree of localization, but the impact slightly decreases since the increased processing throughput by batching reduces the sensitivity to the data access latency. Open-page policy with page level

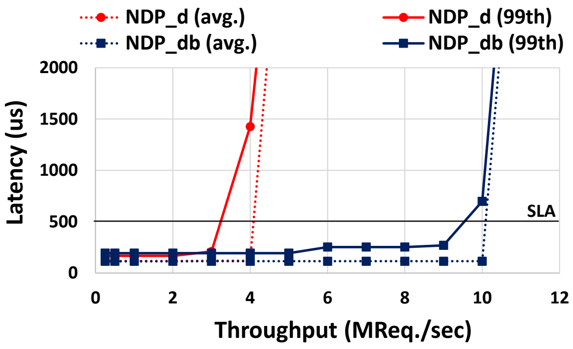


Figure 13: Average and Tail latency of Memcached(FIX) workload.

locality is not translated to page hit in vault controller because of the sequential access pattern of LLT and concurrent accesses to different linked-lists from the multiple processing units. Open-page policy rather increases average DRAM latency by unhidden pre-charge time. NDP-aware localization has minimal impact to the host processing (HSP) read performance, since the capacity of each group is still much larger than on-chip caches except the page level locality and the localization does not result in on-chip cache locality. Page level locality slightly increases host read performance by 4.3%.

With reduced capacity per memory group (larger M), some memory groups become full earlier or conflict occurs more frequently if indexing results are not distributed to all memory groups uniformly, and searching a slot to be stored can take longer execution time. For linked-lists in a hash table, hashing function distributes index values well and the size of vault (256MB) is large enough to amortize the locally skewed distribution. However, linked-list for adjacency list of graph can have severe imbalance in the item distribution because a few vertexes can have much more connected edges. By the severe imbalance in graph, update performance decreases by far from the vault level localization.[12]

For applications such as a key-value store (Memcached), when new elements are added beyond the current memory group size, older elements are dropped *within* each memory group to maintain locality. Smaller capacity per memory group can create more frequent eviction and lower the quality of replacement policy. Thus, item hit rate can decrease with higher degree of localization as shown in Figure 12(b), and biased index distribution can worsen the impact. The degree of localization should be selected properly based on the trade-off of each workload.

Latency sensitive workload: For all workloads except Memcached, completion of the whole LLT operations impacts to the workload completion time, but in Memcached, each LLT operation is mapped to a user request and the latency of each LLT can impact to the latency of the user-level request. Figure 13 shows the average and 99% tail latency of Memcached(FIX). Batching slightly increases the tail latency at low offered throughput. With the proposed trylock scheme, CPU proceeds to next request if lock contention occurs, and the waiting time to acquire the contended lock can be slightly longer, since the lock is released after processing the whole batch. But the tail latency at lower offered throughput is still below the system level agreement (SLA), and the increased throughput enabled by batching gives 2.4× higher request processing throughput under the SLA. Network part is modeled with an assumption to a polling-based light user-level network stack. Client model inserts TCP/IP packets to an outside buffer at a fixed rate (or throughput), and a network model picks the packets from the outside buffer by polling periodically. After parsing the TCP/IP packet, payload (i.e., Memcached commands) is transferred to an inside buffer, the communication buffer between the network model and Memcached.

[12]As described in Section 4.1, relocation sequence checks neighboring memory groups also if conflict occurs at the local memory group. However, if workload has frequent update operation, we can limit the number of trials to small to minimize this update performance degradation.

6. CONCLUSION

In this work, we propose and evaluate near data processing (NDP) of linked-list traversal (LLT). We show how simply offloading LLT for NDP does not necessarily improve performance but can actually degrade energy efficiency because of the additional off-chip channel traversals. As a result, we propose NDP-aware data localization and batching to fully realize the benefits of off-loading to near-memory. NDP-aware data localization minimizes off-chip accesses and reduces LLT latency while batching multiple LLT operations improves overall throughput by exploiting the parallelism available within NDP. We evaluate different big-memory workloads with LLT on our proposed NDP architecture and show up to 5.9× increase in performance and 2.8× reduction in energy compared with baseline host-processing. While this work focused on LLT, the contributions of this work, including exploiting the packetized interface for NDP and using load/store instructions, locality-aware data placement, or batching to maximize throughput, can be possibly extended for other computations that involve significant amount of memory accesses by providing additional hardware logic near the memory.

7. ACKNOWLEDGMENTS

This research was supported in part by the Mid-career Researcher Program (NRF-2013R1A2A2A01069132), in part by the MSIP under the ITRC support program (IITP-2016-H8501-16-1005), and in part by Next-Generation Information Computing Development Program through NRF funded by the Ministry of Science, ICT & Future Planning (2015M3C4A7065647). This research was also funded by SK Hynix.

8. REFERENCES

[1] J. Ahn *et al.*, "Scatter-add in data parallel architectures," in *HPCA*, 2005, pp. 132–142.

[2] J. Ahn *et al.*, "McSimA+: A manycore simulator with application-level+ simulation and detailed microarchitecture modeling," in *ISPASS*, 2013, pp. 74–85.

[3] J. Ahn *et al.*, "A scalable processing-in-memory accelerator for parallel graph processing," in *ISCA*, 2015, pp. 105–117.

[4] B. Akin *et al.*, "Data reorganization in memory using 3D-stacked DRAM," in *ISCA*, 2015, pp. 131–143.

[5] B. Atikoglu *et al.*, "Workload Analysis of a Large-scale Key-value Store," in *ACM SIGMETRICS*, 2012, pp. 53–64.

[6] R. Balasubramonian *et al.*, "Near-data processing: Insights from a MICRO-46 workshop," *IEEE Micro*, vol. 34, no. 4, pp. 36–42, 2014.

[7] J. Balfour *et al.*, "Design tradeoffs for tiled CMP on-chip networks," in *ICS*, 2006, pp. 187–198.

[8] C. Balkesen *et al.*, "Main-memory hash joins on multi-core CPUs: Tuning to the underlying hardware," in *ICDE*, 2013, pp. 362–373.

[9] A. Basu *et al.*, "Efficient Virtual Memory for Big Memory Servers," in *ISCA*, 2013, pp. 237–248.

[10] N. Binkert *et al.*, "The gem5 simulator," *SIGARCH Comput. Archit. News*, vol. 39, no. 2, pp. 1–7, 2011.

[11] S. Blanas *et al.*, "Design and evaluation of main memory hash join algorithms for multi-core CPUs," in *International Conference on Management of data (SIGMOD)*. ACM, 2011, pp. 37–48.

[12] J. Carter *et al.*, "Impulse: Building a smarter memory controller," in *HPCA*, 1999, pp. 70–79.

[13] K. Chen *et al.*, "CACTI-3DD: Architecture-level modeling for 3D die-stacked DRAM main memory," in *DATE*, 2012, pp. 33–38.

[14] P. Dlugosch *et al.*, "An efficient and scalable semiconductor architecture for parallel automata processing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, pp. 3088–3098, 2014.

[15] B. Falsafi *et al.*, "A primer on hardware prefetching," *Synthesis Lectures on Computer Architecture*, vol. 9, no. 1, pp. 1–67, 2014.

[16] Z. Fang *et al.*, "Active memory operations," in *ICS*, 2007, pp. 232–241.

[17] B. Fitzpatrick and A. Vorobey, "Memcached: a distributed memory object caching system," 2011.

[18] M. Gao *et al.*, "Practical Near-Data Processing for In-memory Analytics Frameworks," in *PACT*, 2015, pp. 113–124.

[19] M. Gao *et al.*, "HRL: Efficient and Flexible Reconfigurable Logic for Near-Data Processing," in *HPCA*, 2016, pp. 126–137.

[20] Q. Guo *et al.*, "3d-stacked memory-side acceleration: Accelerator and system design," in *In the Workshop on Near-Data Processing (WoNDP)*, 2014.

[21] A. Gutierrez *et al.*, "Integrated 3D-stacked Server Designs for Increasing Physical Density of Key-value Stores," in *ASPLOS*, 2014, pp. 485–498.

[22] T. H. Hetherington *et al.*, "Characterizing and Evaluating a Key-value Store Application on Heterogeneous CPU-GPU Systems," in *ISPASS*, 2012, pp. 88–98.

[23] B. Hong *et al.*, "Adaptive and Flexible Key-Value Stores Through Soft Data Partitioning," in *ICCD*, 2016.

[24] Hybrid Memory Cube Consortium, "Hybrid Memory Cube Specification 2.0," 2014.

[25] Intel, "Increasing Memory Throughput With Intel Streaming SIMD Extensions 4 (Intel SSE4) Streaming Load," in *White Paper*, 2008.

[26] Intel, "Intel 64 and IA-32 Architectures Software Developer's Manual," 2014.

[27] Intel, "Intel Virtualization Technology for Directed I/O," *Architecture Specification*, 2014.

[28] J. Jeddeloh *et al.*, "Hybrid memory cube new DRAM architecture increases density and performance," in *Symposium on VLSI Technology*, 2012.

[29] N. Jiang *et al.*, "A detailed and flexible cycle-accurate Network-on-Chip simulator," in *ISPASS*, 2013, pp. 86–96.

[30] Y. Kang *et al.*, "FlexRAM: Toward an advanced intelligent memory system," in *ICCD*, 2012, pp. 5–14.

[31] G. Kim *et al.*, "Memory-centric system interconnect design with hybrid memory cubes," in *PACT*, 2013, pp. 145–156.

[32] H. Kim *et al.*, "Understanding Energy Aspects of Processing-near-Memory for HPC Workloads," in *Proceedings of the 2015 International Symposium on Memory Systems*. ACM, 2015, pp. 276–282.

[33] O. Kocberber *et al.*, "Meet the walkers: Accelerating index traversals for in-memory databases," in *MICRO*, 2013, pp. 468–479.

[34] P. M. Kogge, "EXECUBE-A new architecture for scaleable MPPs," in *ICPP*, 1994, pp. 77–84.

[35] J. H. Lee *et al.*, "BSSync: Processing near memory for machine learning workloads with bounded staleness consistency models," in *PACT*, 2015, pp. 241–252.

[36] S. Li *et al.*, "McPAT: an Integrated Power, Area, and Timing Modeling Framework for Multicore and Manycore Architectures," in *MICRO*, 2009, pp. 469–480.

[37] G. H. Loh, "A register-file approach for row buffer caches in die-stacked DRAMs," in *MICRO*, 2011, pp. 351–361.

[38] G. Loh *et al.*, "A processing in memory taxonomy and a case for studying fixed-function pim," in *Workshop on Near-Data Processing (WoNDP)*, 2013.

[39] R. C. Murphy *et al.*, "Introducing to the graph 500," *Cray User's Group (CUG)*, 2010.

[40] L. Nai *et al.*, "Instruction Offloading with HMC 2.0 Standard: A Case Study for Graph Traversals," in *Proceedings of the 2015 International Symposium on Memory Systems*. ACM, 2015, pp. 258–261.

[41] R. Nair *et al.*, "Active Memory Cube: A processing-in-memory architecture for exascale systems," *IBM Journal of Research and Development*, vol. 59, no. 2/3, pp. 17–1, 2015.

[42] D. Patterson *et al.*, "A case for intelligent RAM," *IEEE Micro*, vol. 17, no. 2, pp. 34–44, 1997.

[43] J. T. Pawlowski, "Hybrid Memory Cube (HMC)," in *Hot Chips*, 2011.

[44] S. H. Pugsley *et al.*, "NDC: Analyzing the impact of 3D-stacked memory+ logic devices on MapReduce workloads," in *ISPASS*, 2014, pp. 190–200.

[45] Samsung, "Samsung announces IMDB memory." [Online]. Available: http://www.techeye.net/business/samsung-announces-imdb-memory-with-ndp-hbm-too

[46] G. Sandhu, "DRAM scaling and bandwidth challenges," in *NSF Workshop on Emerging Technologies for Interconnects (WETI)*, 2012.

[47] A. Sodani *et al.*, "Knights Landing: Second-Generation Intel Xeon Phi Product," *IEEE Micro*, vol. 36, no. 2, pp. 34–46, 2016.

[48] C. B. Zilles, "Benchmark health considered harmful," *ACM SIGARCH Computer Architecture News*, vol. 29, no. 3, pp. 4–5, 2001.