Improving In-Memory Database Operations with Acceleration DIMM (AxDIMM)

Donghun Lee
Minseon Ahn
Jungmin Kim
dong.hun.lee@sap.com
minseon.ahn@sap.com
jungmin.kim@sap.com
SAP Labs Korea

Oliver Rebholz
oliver.rebholz@sap.com
SAP SE

Jinin So
Jong-Geon Lee
Jeonghyeon Cho
Vishnu Charan Thummala
jinin.so@samsung.com
jg1021.lee@samsung.com
caleb1@samsung.com
vishnu.c.t@samsung.com
Samsung Electronics

Ravi Shankar JV
Sachin Suresh Upadhya
Mohammed Ibrahim Khan
Jin Hyun Kim
venkata.ravi@samsung.com
sachin1.s@samsung.com
ibrahim.khan@samsung.com
kjh5555@samsung.com
Samsung Electronics

ABSTRACT

The significant overhead needed to transfer the data between CPUs and memory devices is one of the hottest issues in many areas of computing, such as database management systems. Disaggregated computing on the memory devices is being highlighted as one promising approach. In this work, we introduce a new near-memory acceleration scheme for in-memory database operations, called Acceleration DIMM (AxDIMM). It behaves like a normal DIMM through the standard DIMM-compatible interface, but has embedded computing units for data-intensive operations. With the minimized data transfer overhead, it reduces CPU resource consumption, relieves the memory bandwidth bottleneck, and boosts energy efficiency. We implement scan operations, one of the most data-intensive database operations, within AxDIMM and compare its performance with SIMD (Single Instruction Multiple Data) implementation on CPU. Our investigation shows that the acceleration achieves 6.8x more throughput than the SIMD implementation.

CCS CONCEPTS

• **Hardware** → **Emerging interfaces**; • **Information systems** → **Database management system engines**.

KEYWORDS

Near-Memory Processing, AxDIMM, Acceleration DIMM, In-Memory Database, DBMS, Database Management Systems

ACM Reference Format:
Donghun Lee, Minseon Ahn, Jungmin Kim, Oliver Rebholz, Jinin So, Jong-Geon Lee, Jeonghyeon Cho, Vishnu Charan Thummala, Ravi Shankar JV, Sachin Suresh Upadhya, Mohammed Ibrahim Khan, and Jin Hyun Kim. 2022. Improving In-Memory Database Operations with Acceleration DIMM (AxDIMM). In *Data Management on New Hardware (DaMoN'22), June 13, 2022, Philadelphia, PA, USA.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3533737.3535093

1 INTRODUCTION

As data volumes continue to increase, memory becomes the main bottleneck in many data-intensive applications like in-memory database management systems (IMDBMS). Previous research has successfully improved the performance of data-intensive operations by using additional accelerators, such as FPGA [11, 19, 23, 26, 34] and GPU [15, 25, 32]. However, it is inevitable to copy the data from host memory devices to the local memory within the accelerators, thus consuming a lot of energy to transfer the data in-between. To alleviate this data movement overhead between processing units and host memory devices, processing-in-memory (PIM) [9, 22] and near-memory processing technologies [18] have been recently highlighted.

Prior work on in-memory processing, such as UPMEM [10] or HBM-PIM [22], involves an advanced concept of near-memory processing. However, these techniques need additional data copying or internal data reformatting before the embedded processing capability is invoked. This extra overhead diminishes the benefit of in-memory processing.

DaMoN'22, June 13, 2022, Philadelphia, PA, USA
© 2022 Association for Computing Machinery.
ACM ISBN 978-1-4503-9378-2/22/06…$15.00
https://doi.org/10.1145/3533737.3535093

Our previous research [21] introduced a simple proof-of-concept system on the PCIe-based FPGA to show that near-memory acceleration optimizes data movement between processing units and memory devices without copying the data out of the memory devices or reformatting the original data for embedded processing.

In this paper, we introduce Acceleration DIMM (AxDIMM), an acceleration platform for database operations implemented on a custom FPGA board with the standard DIMM-compatible interface. It behaves like a normal DIMM that is compliant with the standard DDR protocol, but has embedded computing units for the data-intensive operations. Since there is no additional overhead from data copying or internal reformatting, AxDIMM minimizes data movement and reduces memory bandwidth usage and CPU usage by the offloaded database operations, thus improving energy efficiency. In addition, multiple engines within AxDIMM perform parallel processing on multiple memory ranks (sets of DRAM chips). Since AxDIMM supports the traditional DIMM form factor, it can easily replace normal DIMMs without system modification, and its memory space is recognized as a part of the normal memory space in the operating system.

We implement scan operations as basic data-intensive operations and demonstrate the feasibility of AxDIMM in IMDBMSs. Our experiments show up to 6.8x better performance in both latency and throughput than the SIMD (Single Instruction Multiple Data) implementation on CPU.

Here are the main contributions of our research using AxDIMM:

- We apply a near-memory acceleration scheme for database operations to minimize the data transfer overhead between CPUs and memory devices.
- We implement AxDIMM, supporting the standard DDR protocol and accelerating the scan operations.
- We prove the performance benefit of AxDIMM.
- Finally, we discuss the limitations of the current prototype and possible approaches to rectify them.

The remainder of this paper is organized as follows: Section 2 introduces the scan operations in IMDBMS. Section 3 shows the architecture and internal configuration of the AxDIMM system. The experimental setup is described in Section 4, and the evaluation results are addressed in Section 5. Section 6 discusses the technical issues. Section 7 represents the related work and Section 8 concludes the paper.

2 SCAN OPERATIONS IN IMDBMS

This section describes the behavior of scan operations as one of the basic data-intensive database operations and introduces its performance characteristics.

2.1 Behavior

Scan operations are data-intensive in IMDBMSs, filtering out the rows that satisfy the predicates. Recent IMDBMSs widely use the columnar storage for fast read accesses to support hybrid transactional and analytical processing (HTAP) [30, 31]. We use the columnar storage consisting of the read-optimized main storage and the separate write-optimized delta storage [29, 36]. To reduce the memory footprint, the main storage uses dictionary encoding, where all distinct values are stored in a dictionary in lexicographical order. The individual value in each row is replaced with the corresponding value ID of the dictionary and stored in a separate value ID array. The value ID arrays are bit-packed compressed such that the number of bits is determined by the number of distinct values. If the dictionary contains N distinct values, the number of bits per value is $ceil(log_2(N))$ [12]. Scan operations read the bit-packed value ID arrays to apply the predicates. Two common types of predicates are (1) range predicate, having from/to values, and (2) inlist predicate, having a list of filtered values. The scanned results are marked in a bit vector (BV), where the number of bits in the bit vector is same as the number of the rows in the table. The value (0 or 1) at a certain index in the BV indicates whether or not the row at the same index of the value ID array satisfies the filter conditions.

2.2 Performance Characterization

Scan operations access the columns sequentially. Most modern processors boost the performance of sequential accesses through prefetching. Once sequential accesses are detected, the prefetcher in L2 caches issues the next cacheline ahead of the actual memory request to reduce the long latency of memory accesses. Another performance improvement to the scan operations can be achieved with SIMD instructions like SSE4, AVX2, and AVX512 [38]. However, they are bottlenecked by the limited memory bandwidth due to the huge memory traffic within a short period of time. Fig. 8 in Section 5 shows that SIMD implementations have high memory bandwidth bound, which means a large portion of CPU cycles is stalled while the application is running due to the memory bandwidth limit. Previous research [21] shows that massive data-intensive operations like scan operations can block the other critical transactions due to the memory bandwidth bound.

3 AXDIMM DESIGN FOR SCAN OPERATIONS

This section presents the overall architecture and main components of AxDIMM that support the embedded scan operations. The bit-packed target data for the scan operations resides within AxDIMM, as explained in section 2.1. The result buffer for each scan is allocated within AxDIMM before invoking the scan. Finally, the scan reads the compressed data and writes the filtered results to the pre-allocated result buffer.

3.1 Hardware Architecture

Fig. 1 shows the overall architecture and working frequency of the AxDIMM prototype. It includes the system configuration register space, database acceleration (DBA) engines, and memory subsystem. The AxDIMM behaves in the same way as the normal memory DIMM interfaced via DDR protocol, but supports the additional near-memory processing feature within it. Since the main storage of IMDBMS is stored in AxDIMM and the scan operations are performed by embedded DBA engines, there is no need to copy the data to processing units out of AxDIMM nor to transform the data format.

Fig. 2 shows the picture of our AxDIMM prototype. To make AxDIMM compatible with the existing platform, we customize the BIOS. The DDR4 host interface in the AxDIMM works at 400 MHz to support FPGA I/O speed limitation, while the baseline host DRAM

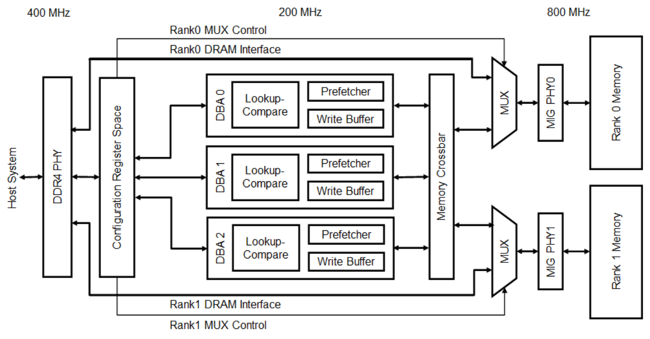

Figure 1: AxDIMM hardware architecture

runs at 800 MHz. AxDIMM is implemented on an on-board Xilinx Zynq Ultrascale+ FPGA running at 200 MHz with 32 GB on-board DRAM running at 800 MHz. Its internal memory bandwidth is twice as large as the baseline host DRAM because of two ranks and two memory interface generators (MIG) operating at the same frequency as the baseline host DRAM. The embedded data-intensive operations thus utilize the increased internal memory bandwidth, resulting in better throughput. The CPU usage is reduced by the offloaded database operations in AxDIMM. As a result, the conserved computing resources are consumed by the other operations improving overall database performance.

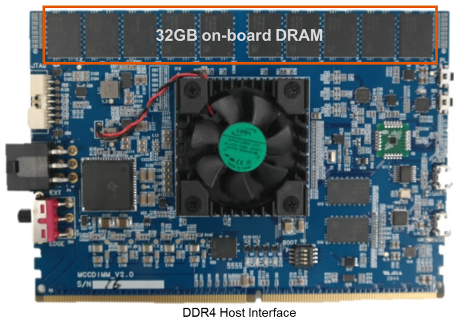

Figure 2: AxDIMM prototype

3.1.1 System Configuration Register Space. The configuration register space is a portion of memory that consists of a set of registers for AxDIMM itself and three DBA engines, such as control registers and status registers. In addition, all the parameters in offloading APIs are stored in the configuration registers of each DBA engine, including the addresses of the input data and the result buffer, the number of bits per value in the value ID array, the starting index, and the predicates for range and inlist scans. To facilitate the accesses by the host CPU, 1GB is reserved at the beginning of Rank0.

3.1.2 DBA Engines. The AxDIMM has 3 DBA engines performing the offloaded database operations. Each DBA engine consists of a lookup-compare module, a prefetcher, and a write buffer. The lookup-compare module reads the data and compares it with the predicate. The prefetcher loads the data from the end-point memory in AxDIMM to the SRAM buffer in advance until the requested scan is completed. The write buffer temporarily stores the scan result in the SRAM buffer before writing it to the AxDIMM memory.

Within DBA engines, all the virtual addresses from the applications should be translated to physical addresses. AxDIMM memory is directly mapped to the host address space. Since the memory interleaving is disabled in the host DRAM, AxDIMM memory is in one single contiguous region of the physical address space. Therefore, we can simply translate the virtual addresses from the application by using the process page table in the Linux kernel [5].

3.1.3 Memory Crossbar. There is a memory crossbar between the DBA engines and the AxDIMM memory. This memory crossbar arbitrates memory accesses from 3 DBA engines and sends them to the corresponding rank, thus allowing all 3 DBA engines to access all the ranks in AxDIMM. Each DBA engine can perform the offloaded scan operations regardless of the input data location within AxDIMM.

3.1.4 Concurrent Accesses between the Host CPU and DBA Engines. Basically, all the accesses to memory devices should be arbitrated by the memory controller in the host CPU to maintain the deterministic behavior of the standard DDR protocol. All the memory requests must go through the memory controller to support the concurrent memory accesses by both the host CPU and DBA engines. However, memory accesses from DBA engines cannot go through the memory controller because AxDIMM is located on the DIMM side. Therefore, it is regarded that implementing a memory controller within AxDIMM instead of the memory controller in the host CPU is out of scope and this work focuses on the acceleration within a DIMM.

As a workaround, we have two separate modes in AxDIMM, to enable the DBA engines to access the AxDIMM memory in a time-sharing manner. In non-acceleration mode, the host CPU can directly access the ranks as a normal DIMM while the DBA engines are idle. The end-point memory in the AxDIMM behaves in the same way as the normal memory. In Fig. 1, when the mode is set to non-acceleration, all the DRAM interfaces from DDR4 PHY are directly connected to MIGs when the links from the memory crossbar are disconnected. As such, AxDIMM behaves like a normal DIMM. In acceleration mode, the DBA engines can exclusively access the ranks and perform the offloaded scan operations, while the regular DRAM accesses from the host CPU are not accepted. All the links from the memory crossbar are directly connected to MIGs when the DRAM interfaces are disconnected.

3.1.5 Write Combine Cache Policy. When the result buffer is allocated and initialized by the host CPU, it is also copied to the CPU caches. As such, the copy in the caches should be invalidated when a DBA engine modifies the result buffer. To avoid the overhead of cache coherency in the AxDIMM memory, we use a write combine cache policy [1]. The whole AxDIMM memory region is defined as uncached for reads. Since any data read from AxDIMM cannot be stored in caches, it bypasses the CPU caches, reading the AxDIMM memory directly. Therefore, there is no need to invalidate the caches when reading the result buffers in AxDIMM after the DBA engine performs offloaded scan operations. Since the cache size is generally smaller than the result buffer and the host CPU consumes the results sequentially, the impact of this cache policy is negligible.

3.2 Software Stack

This section explains the software stack used to invoke the designed hardware. Fig. 3 shows the diagram for the AxDIMM software stack. The AxDIMM memory is exposed to the host as a DAX device [2] in the system. This allows applications to use the mmap interface to access this memory region directly, which means no additional driver is needed. AxDIMM library provides offloading APIs to the application and manages the DBA registers in AxDIMM to invoke the embedded scan operations. Once the offloading APIs are called, the offloading requests are stored in the offloading queue. If one of the DBA engines is idle, the scheduler assigns the offloading request to the DBA engine by manipulating the registers.

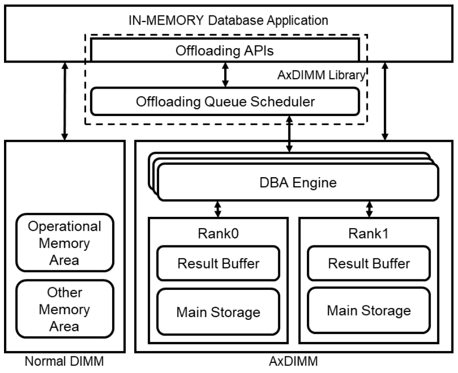

Figure 3: AxDIMM software stack

3.2.1 Offloading APIs. Table 1 shows the two scan APIs implemented in the system: 1) Range scan with bit-vector output and 2) Inlist scan with bit-vector output. Each API has the following input parameters as seen in Table 2: the address pointer to the target data, the starting index of the data to be scanned, the number of the records to be scanned, the address pointer to the result buffer, bit case (the number of bits per value) used in the bit-packed compression, and filter conditions. In case of inlist scans, the predicate consists of a bit vector where each bit indicates whether or not the value at the index is included in the predicate. Before calling the offloading APIs, the result buffer should be allocated in the AxDIMM memory by the application, and the starting address of the result buffer is sent to AxDIMM as the input parameter of the offloading APIs.

3.2.2 Offloading Queue Scheduler. It is quite common to generate several worker threads to speed up query processing in a DBMS. The column for a scan operation is divided into several chunks and multiple scan requests are created concurrently. Since the number of scan requests can be larger than the number of DBA engines, an offloading queue is implemented in the AxDIMM library to handle the multiple requests. The queue handles up to 64 concurrent requests in the system. The scheduler assigns a single request to an idle DBA engine in a rank-aware first-in-first-out (FIFO) manner. Once the DBA engine completes its offloaded scan operation, it informs the scheduler. The scheduler then sends the notification back to the requester and removes the request from the queue.

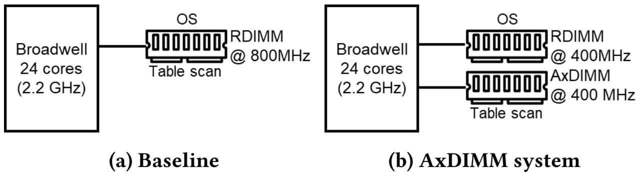

Figure 4: System setup

4 EXPERIMENTS

This section describes the experimental setup to evaluate the performance of AxDIMM and the micro-benchmark to generate the scan operations against the bit-packed data.

4.1 System Setup

To evaluate the performance benefit in AxDIMM, we built a test system on a Broadwell server with Intel Xeon CPU E5-2650 v4 @ 2.2 GHz. As shown in Fig. 4, the baseline system has one single 32 GB RDIMM working at 800 MHz, while the AxDIMM system has one 32 GB RDIMM and one 32 GB AxDIMM working at 400 MHz. RDIMM and AxDIMM are located in different memory channels in the AxDIMM system. If the channel interleaving is turned on, data fragments can happen across DIMMs. As AxDIMM does not support communication between multiple DIMMs and cannot handle the data fragments, we disable memory interleaving.

To evaluate the performance of the scan operations, we use AVX2 implementation [38] in the baseline, since the Intel® Broadwell CPU does not support AVX512. Our in-house multi-threaded micro-benchmark adopts the mmap vector implementation [4] to allocate memory in AxDIMM for the input data and the result buffer.

4.2 Configurations

For the test data in the micro-benchmark, we generate a value ID array with the 2 billion records of input data using a uniform random generator and adopt bit-packed compression for each bit-case. During the scan operations, the dictionary is not used because the filter conditions are expressed only with value IDs and scan operations read only value ID array. In this work, we analyze the performance of three bit-cases — 4, 10, and 17 — with two selectivities per bit-case to evaluate the performance effect according to the result size. The test configurations for the scan operations are summarized in Table 3.

4.3 Procedure

The micro-benchmark performs by (1) preparing the data and the result buffer for each worker thread in the AxDIMM memory or RDIMM at the baseline, (2) calling offloading APIs for the offloaded scan operations or AVX2-based scan operations, (3) measuring the performance after completing the scan operations, and (4) optionally retrieving the result to confirm the correctness of the scan operations. We create up to 8 threads in the micro-benchmark to see the performance scalability for both AxDIMM acceleration and AVX2 implementation. When preparing the data and the result

Table 1: Scan offloading APIs

Range Scan	int scan_range_to_bv(begin_index, *data, data_size, *result_buffer, bit_case, min, max);
Inlist Scan	int scan_inlist_to_bv(begin_index, *data, data_size, *result_buffer, bit_case, *predicates);

Table 2: Parameters for scan APIs

Type	Name	Description
uint64_t	begin_index	the starting index of the data to be scanned
uint64_t*	data	the address pointer to the target data
uint64_t	data_size	the number of the records to be scanned
uint64_t*	result_buffer	the address pointer to the result buffer
uint32_t	bit_case	bit case used for the bit compression
uint32_t	min	the minimum value ID of the range predicate
uint32_t	max	the maximum value ID of the range predicate
uint64_t*	predicates	the address pointer to the predicate vector of the inlist predicate

Table 3: Predicate configuration for scan operations

Bit-case	Selectivity (1)		Selectivity (2)	
4	0.0625	range: 1 to 2	0.125	range: 1 to 3
10	0.001	range: 1 to 2	0.1	range: 1 to 103
17	0.001	range: 1 to 133	0.1	range: 1 to 13109

buffer in the AxDIMM memory, AxDIMM is set to non-acceleration mode. While performing the scan operations in AxDIMM, it is set to acceleration mode. When retrieving the result from the AxDIMM memory, AxDIMM is set back to non-acceleration mode.

5 EVALUATION

This section shares our evaluation results on the performance of DBA engines of AxDIMM in comparison to AVX2 implementation against the data on the RDIMM as the baseline.

First, we share the latency test results by comparing the elapsed time in seconds for a single scan between AVX2 implementation and AxDIMM acceleration. The elapsed time includes the time for reading the data of 2 billion records, performing the scan operation, and writing the results in both implementations. Second, we show the throughput scalability as the number of scan operations on RDIMM (AVX2) and AxDIMM increases. Finally, we compare the CPU usage and the memory bandwidth bound between AVX2 and AxDIMM while running multiple scan operations, which is gathered via Intel® VTune® [3]. The two scans (range and inlist) are evaluated on three bit-cases with two selectivities for each bit-case.

5.1 Latency Test Results

Fig. 5 shows the single scan latency on 2 billion records. AxDIMM acceleration has a noticeable gain in latency compared to the AVX2 implementation on RDIMM. AxDIMM speeds up the scan operation from 1.3x to 6.8x because of the dedicated hardware implementation within the memory device. AxDIMM has a bigger gain from the inlist scan than the range scan for all bit-cases, regardless of their selectivity conditions. AVX2 implementation shows much longer latency in the inlist scan compared to the range scan, due to the increased number of comparisons based on the number of arguments in the inlist predicates. Intrinsically, the inlist scan has multiple comparisons so that the inlist predicate is a bit vector composed of multiple bits where the number of bits is same as the number of value IDs in the dictionary. Rather, the range scan has only two comparisons with its predicate.

Fig. 5 shows that the inlist scan has better performance improvement than the range scan because AxDIMM performs the concurrent comparisons of the inlist scan more efficiently than AVX2 implementation. AxDIMM acceleration achieves similar performance between the inlist scan and the range scan in bit-cases 4 and 10 because it performs parallel comparisons per cycle. However, bit-case 17 has much longer latency in inlist scan acceleration because the number comparison per clock within FPGA is reduced by the increased size of the inlist predicate.

5.2 Throughput Scalability

Fig. 6 shows the average scan throughput as the number of worker threads increases. AxDIMM acceleration shows a much higher throughput, up to 6.8x more than AVX2 implementation. The AxDIMM throughput becomes saturated with 3 threads in the bit-case 4 due to the limited number of DBA engines, while it is saturated earlier in higher bit-cases due to the internal bandwidth limitation resulting from the increased data size. Even with more than 3 threads, AxDIMM shows much better performance than AXV2 implementation. The throughput becomes smaller in higher bit-cases in both AxDIMM and AVX2 implementations due to the larger data size. The AXDIMM throughput of inlist scan in bit-case 17 is reduced due to the increased predicate size, like the latency results.

5.3 CPU Usage and Memory Bandwidth Bound

We measure CPU usage and memory bandwidth bound in the bit-case 10 and selectivity 0.1 to show that AxDIMM reduces CPU resource usage and the memory bottleneck. First, we measure the average CPU usage in the high load phase in the micro-benchmark. Fig. 7 shows that most of the CPU usage resulting from the scan operations is eliminated by offloading. The saved computing resources are used by other database operations and improve overall database performance. Second, we measure the memory bandwidth

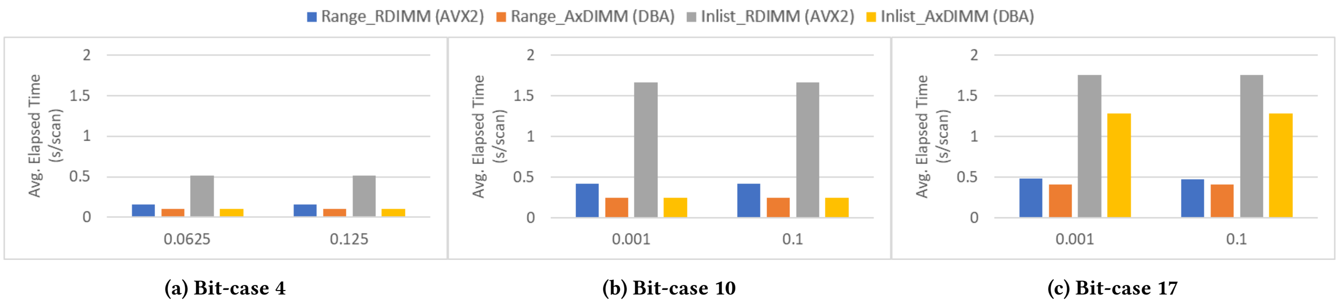

Figure 5: Latency (x-axis: selectivity)

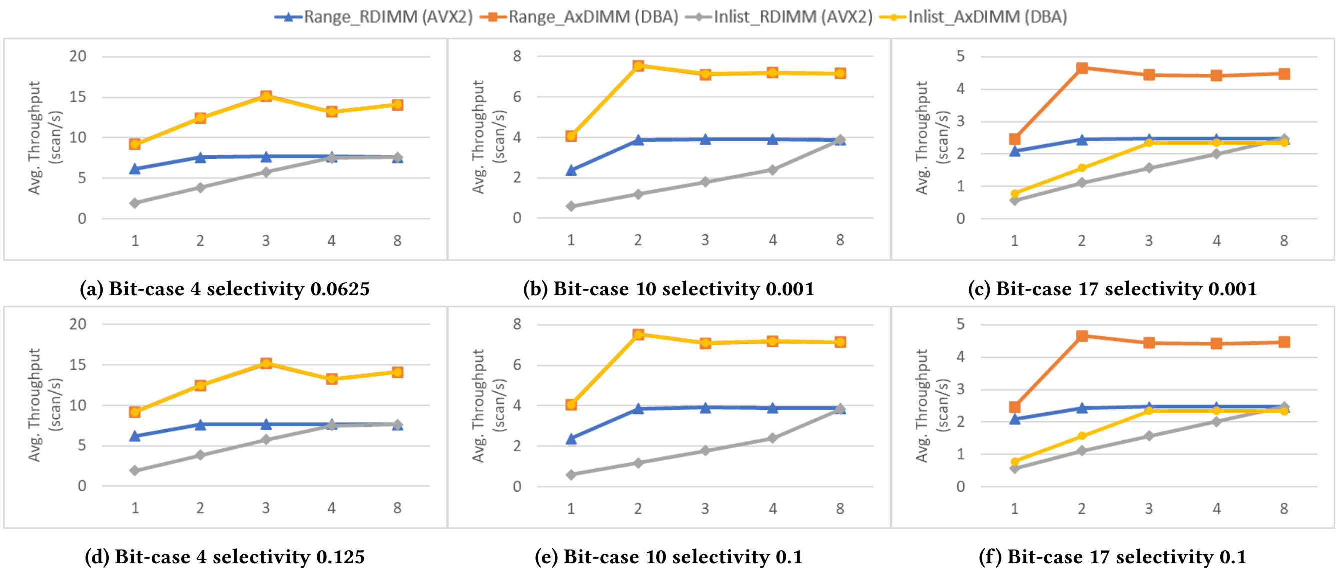

Figure 6: Throughput (x-axis: the number of worker threads)

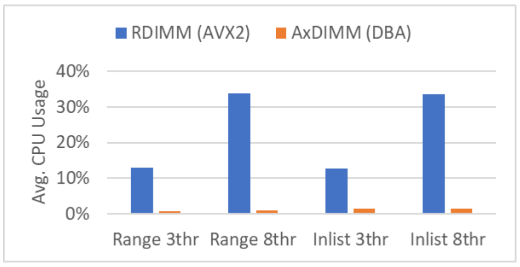

Figure 7: Average CPU usage

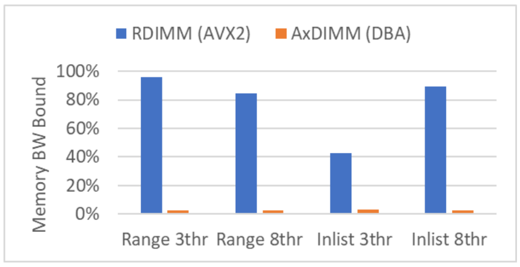

Figure 8: Memory bandwidth bound

bound by profiling the micro-benchmark using Intel® VTune® [3]. Fig. 8 shows that most of the memory bandwidth bound is relieved by offloading the scan operations. When performing AVX2 implementation on CPU, all the data in the value ID array should be read and the results are written back. However, AxDIMM acceleration reduces the data transfer, relieving the memory bandwidth bound.

5.4 Overall Analysis

Through our experiments above, we confirm that the data-intensive operations can be optimized via near-memory processing. The performance improvement comes from (1) the dedicated hardware implementation of DBA engines and (2) full exploitation of the internal bandwidth of the memory subsystem larger than the memory bandwidth in the host CPU. The results show that AxDIMM acceleration can save most of all CPU usage and memory bandwidth usage for the scan operations. Thus, the relieved host CPU power and memory bandwidth usage will be utilized for other database workloads resulting in overall performance improvement in IMDBMSs.

6 DISCUSSION

This section addresses our insights from this work, and describes the limitations of the current implementation of AxDIMM and how we can rectify them in the future.

6.1 Feasibility of Near-Memory Acceleration

This work implements the scan operations within AxDIMM to show that the offloading of expensive database operations to AxDIMM can relieve the memory bandwidth bound between host CPUs and memory devices, and save the computing power of host CPUs. We believe offloading to AxDIMM can be applied to any data-intensive operation which causes the memory bandwidth bound and consumes much computing power, such as (de)compression of the bit-packed data, or building a hash table. Besides the data-intensive operations, compute-intensive operations accessing a large volume of data are also good candidates. For example, operations with advanced compression schemes, matrix multiplications, and other ML operations [18] can be accelerated in AxDIMM.

6.2 Limited Gate Count and Frequency in FPGA

The current implementation of AxDIMM is built on a custom FPGA board. Because of the limited gate count and the limited frequency, only three DBA engines working at 200 MHz are implemented, consuming about 79% of FPGA resources. It is these limitations that restrict the performance of acceleration. If we could resolve these issues by applying ASIC technology in the next phase, much higher performance improvement is expected, thanks to more parallelism available from the increased number of DBA engines and the reduced latency from the higher working frequency.

6.3 Cache Coherency Limitation

As another limitation of near-memory processing in AxDIMM, Section 3.1.5 explains the cache coherency issue with regard to the output result buffer. When AxDIMM writes the results into the output results buffer, some data in the cache can be incoherent. Thus, we use the write combine cache policy to avoid it. In addition to the result buffer, we also need to take care of the input data of the scan operations. When the host CPU writes data, some portion of the data may reside in the cache but not in the AxDIMM memory. Therefore, it must be flushed by the application before AxDIMM acceleration is invoked. In this work, cache flush and memory barrier instructions are used when data is written to the AxDIMM memory.

Recently, new memory interfaces like CXL are proposed to support these cache coherency issues. CXL type 2 uses the CXL.cache protocol, supporting cache coherency between host and CXL devices as well as CXL.memory and CXL.io. If DBA engines are implemented with CXL type 2, these cache coherency issues are resolved. CXL also enables concurrent accesses by the host CPU and the DBA engines because the memory controllers are located within the CXL memory devices, resulting in memory access scheduling capability within the CXL devices.

When the data is updated by the host CPU, AxDIMM does not care data modification while one processing unit accesses the specific memory area because data consistency is assured in the same way of the existing DBMS such as a locking mechanism.

7 RELATED WORK

GPU can perform database operations exploiting its massive parallelism. The group-by and aggregation operations can be offloaded to GPU in a hash-based manner [17]. However, this approach cannot avoid data movement from the host DRAM to GPU memory. Adaptive work placement [16] proposes a way to select a right computing unit to improve the overall performance in a heterogeneous computing environment consisting of CPUs and GPUs. Shanbhag et. al. [33] analyze the performance gain on GPU and show that operations like selection, projection, and sort, have a good speedup that is nearly equal to the bandwidth ratio. To overcome performance, a tiled-based execution model is proposed using GPU's memory to store workload set directly instead of using GPU as a coprocessor.

Offloading expensive operations to FPGAs is widely accepted in the database area as well. Lasch et. al. [20] implement a computationally expensive re-pair compression algorithm in an FPGAs using OpenCL. Mohsen et. al. [27] show that FPGA implementation of binary packing can efficiently saturate PCIe bus bandwidth and achieve a better compression ratio than CPU execution. Memory-accessible FPGAs like an FPGA-embedded hybrid system [24] and Xeon+FPGA heterogeneous [13, 14, 35] architecture can offload the database operations to FPGAs attached in the same interconnect with CPUs. In these approaches, there is no explicit data copy to FPGAs because the coherency in the interconnect can move the data in the memory to the offloading device in the on-demand manner, but the data movement is still required. Recently, Alonso et. al. [7] develop a near-memory processing using an ARM-based server and FPGAs with the cache coherence protocol integrated with an open-source database, but it is in its early stage to measure the performance of data-intensive analysis queries.

Near-memory processing uses computing power in the near-memory devices to mitigate bandwidth bottleneck between CPUs and memory devices, improving performance and energy consumption. Boromand et. al. [8] analyze the impact of processing-in-memory (PIM) performing part of the computation close to memory. Recently, MCN (Memory Channel Network) architecture [6] has been proposed to develop a DIMM-based memory channel network with MCN DIMMs, buffered DIMM with a small processor, to give the host computer the illusion that MCN DIMMs are connected through an Ethernet interface.

SIMD instructions for vector processing unit are widely used in the database operations to accelerate performance. First, Willhalm et. al. [39, 40] uses SIMD instructions when scanning column vectors with various predicates, such as range predicate and in-list predicate. Second, Sitaridi et. al. [37] uses SIMD instructions for optimized string matching against regular expressions. Third, Mula et. al. [28] proposes an improved vectorized approach for the population count using AVX2.

8 CONCLUSION

As one of the disaggregated computing approaches to resolving the memory bottleneck issue in IMDBMSs, we introduced a near-memory acceleration scheme called Acceleration DIMM (AxDIMM). Since data-intensive operations can be processed within AxDIMM, it reduces CPU resource consumption as well as memory bandwidth usage.

3 DBA engines are implemented on the custom FPGA board with the standard DIMM-compatible interface to conduct the scan operations against the data in the AxDIMM memory. The application, such an IMDBMS, accesses the memory space of AxDIMM uisng the mmap function without any additional device driver and calls the offloading APIs provided by the AxDIMM library. The implemented scan operations read the bit-packed compressed data, evaluate the range or inlist predicates, and write the filtered results to the result buffer.

Our experiments showed a significant improvement in AxDIMM, up to 6.8x latency and throughput compared to SIMD (AXV2) implementation on CPU. What's more, its profiling results showed that most of the CPU usage and the memory bound was eliminated with AxDIMM. The results are quite promising, since AxDIMM has a smaller number of computing units (3 DBA engines) and a slower working frequency (200 MHz) than the host CPU.

Even with these promising results of disaggregated memory computing, we also observed a few important architectural limitations in our near-memory processing approach, such as concurrent accesses by both host CPUs and DBA engines and cache coherency. Fortunately, new memory interfaces like CXL would be a good solution to overcome these issues. As our next research topic, we'd like to implement DBA offloading in CXL memory devices and evaluate whether or not CXL can actually resolve the existing architectural limitations. Furthermore, we will extend our approach to include a CXL-based memory pooling system.

REFERENCES

[1] 1998. *Write Combining Memory Implementation Guidelines.* https://download.intel.com/design/PentiumII/applnots/24442201.pdf

[2] 2021. *Direct Access for files.* https://www.kernel.org/doc/Documentation/filesystems/dax.txt

[3] 2021. *Intel® VTune™ Profiler.* https://www.intel.com/content/www/us/en/developer/tools/oneapi/vtune-profiler.html

[4] 2021. *Mmap Allocator.* https://github.com/johannesthoma/mmap_allocator

[5] 2021. *Process Page Table.* https://www.kernel.org/doc/Documentation/vm/pagemap.txt

[6] 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, Piscataway, NJ, USA, 802–814. https://doi.org/10.1109/MICRO.2018.00070

[7] Gustavo Alonso, Timothy Roscoe, David Cock, Mohsen Ewaida, Kaan Kara, Dario Korolija, David Sidler, and Zeke Wang. 2020. Tackling Hardware/Software co-design from a database perspective. In *CIDR*.

[8] Amirali Boroumand, Saugata Ghose, Youngsok Kim, Rachata Ausavarungnirun, and etc. 2018. Google Workloads for Consumer Devices: Mitigating Data Movement Bottlenecks. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems* (Williamsburg, VA, USA) *(ASPLOS '18)*. ACM, New York, NY, USA, 316–331. https://doi.org/10.1145/3173162.3173177

[9] Amirali Boroumand, Saugata Ghose, Youngsok Kim, Rachata Ausavarungnirun, Eric Shiu, Rahul Thakur, Daehyun Kim, Aki Kuusela, Allan Knies, Parthasarathy Ranganathan, et al. 2018. Google workloads for consumer devices: Mitigating data movement bottlenecks. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems.* 316–331.

[10] Fabrice Devaux. 2019. The true processing in memory accelerator. In *2019 IEEE Hot Chips 31 Symposium (HCS)*. IEEE Computer Society, 1–24.

[11] Ashutosh Dhar, Sitao Huang, Jinjun Xiong, Damir Jamsek, Bruno Mesnet, Jian Huang, Nam Sung Kim, Wen-mei Hwu, and Deming Chen. 2019. Near-memory and in-storage FPGA acceleration for emerging cognitive computing workloads. In *2019 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*. IEEE, 68–75.

[12] Franz Färber, Norman May, Wolfgang Lehner, Philipp Große, Ingo Müller, Hannes Rauhe, and Jonathan Dees. 2012. The SAP HANA Database–An Architecture Overview. *IEEE Data Eng. Bull.* 35, 1 (2012), 28–33.

[13] Z. István, D. Sidler, and G. Alonso. 2016. Runtime Parameterizable Regular Expression Operators for Databases. In *2016 IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 204–211. https://doi.org/10.1109/FCCM.2016.61

[14] Kaan Kara, Jana Giceva, and Gustavo Alonso. 2017. FPGA-Based Data Partitioning. In *Proceedings of the 2017 ACM International Conference on Management of Data* (Chicago, Illinois, USA) *(SIGMOD '17)*. Association for Computing Machinery, New York, NY, USA, 433–445. https://doi.org/10.1145/3035918.3035946

[15] Tomas Karnagel, Dirk Habich, and Wolfgang Lehner. 2017. Adaptive work placement for query processing on heterogeneous computing resources. *Proceedings of the VLDB Endowment* 10, 7 (2017), 733–744.

[16] Tomas Karnagel, Dirk Habich, and Wolfgang Lehner. 2017. Adaptive Work Placement for Query Processing on Heterogeneous Computing Resources. *Proc. VLDB Endow.* 10, 7 (March 2017), 733–744. https://doi.org/10.14778/3067421.3067423

[17] Tomas Karnagel, Renè Müller, and Guy M. Lohman. 2015. Optimizing GPU-accelerated Group-By and Aggregation.. In *ADMS@VLDB*, Rajesh Bordawekar, Tirthankar Lahiri, Bugra Gedik, and Christian A. Lang (Eds.). 13–24. http://dblp.uni-trier.de/db/conf/vldb/adms2015.html#KarnagelML15

[18] Liu Ke, Xuan Zhang, Jinin So, Jong-Geon Lee, Shin-Haeng Kang, Sukhan Lee, Songyi Han, Yeongon Cho, Jin Hyun Kim, Yongsuk Kwon, et al. 2021. Near-Memory Processing in Action: Accelerating Personalized Recommendation with AxDIMM. *IEEE Micro* (2021).

[19] Robert Lasch, Suleyman S Demirsoy, Norman May, Veeraraghavan Ramamurthy, Christian Färber, and Kai-Uwe Sattler. 2020. Accelerating re-pair compression using FPGAs. In *Proceedings of the 16th International Workshop on Data Management on New Hardware.* 1–8.

[20] Robert Lasch, Suleyman S. Demirsoy, Norman May, Veeraraghavan Ramamurthy, Christian Färber, and Kai-Uwe Sattler. 2020. Accelerating Re-Pair Compression Using FPGAs. In *Proceedings of the 16th International Workshop on Data Management on New Hardware* (Portland, Oregon) *(DaMoN '20)*. Association for Computing Machinery, New York, NY, USA, Article 8, 8 pages. https://doi.org/10.1145/3399666.3399931

[21] Donghun Lee, Andrew Chang, Minseon Ahn, Jongmin Gim, Jungmin Kim, Jaemin Jung, Kang-Woo Choi, Vincent Pham, Oliver Rebholz, Krishna Malladi, et al. 2020. Optimizing Data Movement with Near-Memory Acceleration of In-memory DBMS.. In *EDBT*. 371–374.

[22] Sukhan Lee, Shin-haeng Kang, Jaehoon Lee, Hyeonsu Kim, Eojin Lee, Seungwoo Seo, Hosang Yoon, Seungwon Lee, Kyounghwan Lim, Hyunsung Shin, et al. 2021. Hardware Architecture and Software Stack for PIM Based on Commercial DRAM Technology: Industrial Product. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 43–56.

[23] Nusrat Jahan Lisa, Annett Ungethüm, Dirk Habich, Wolfgang Lehner, Tuan DA Nguyen, and Akash Kumar. 2018. Column Scan Acceleration in Hybrid CPU-FPGA Systems.. In *ADMS@ VLDB*. 22–33.

[24] Nusrat Jahan Lisa, Annett Ungethüm, Dirk Habich, Wolfgang Lehner, Tuan D. A. Nguyen, and Akash Kumar. 2018. Column Scan Acceleration in Hybrid CPU-FPGA Systems. In *International Workshop on Accelerating Analytics and Data Management Systems Using Modern Processor and Storage Architectures, ADMS@VLDB 2018, Rio de Janeiro, Brazil, August 27, 2018.* 22–33. http://www.adms-conf.org/2018-camera-ready/habich_adms2018.pdf

[25] Clemens Lutz, Sebastian Breß, Steffen Zeuch, Tilmann Rabl, and Volker Markl. 2020. Pump up the volume: Processing large data on GPUs with fast interconnects. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data.* 1633–1649.

[26] Mahmoud Mohsen, Norman May, Christian Färber, and David Broneske. 2020. Fpga-accelerated compression of integer vectors. In *Proceedings of the 16th International Workshop on Data Management on New Hardware.* 1–10.

[27] Mahmoud Mohsen, Norman May, Christian Färber, and David Broneske. 2020. FPGA-Accelerated Compression of Integer Vectors. In *Proceedings of the 16th International Workshop on Data Management on New Hardware* (Portland, Oregon) *(DaMoN '20)*. Association for Computing Machinery, New York, NY, USA, Article 9, 10 pages. https://doi.org/10.1145/3399666.3399932

[28] Wojciech Muła, Nathan Kurz, and Daniel Lemire. 2016. Faster Population Counts Using AVX2 Instructions. *Computer Journal* 61 (11 2016). https://doi.org/10.1093/comjnl/bxx046

[29] Hasso Plattner. 2009. A common database approach for OLTP and OLAP using an in-memory column database. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*. 1–2.

[30] Hasso Plattner. 2014. The impact of columnar in-memory databases on enterprise systems: implications of eliminating transaction-maintained aggregates. *Proceedings of the VLDB Endowment* 7, 13 (2014), 1722–1729.

[31] Iraklis Psaroudakis, Florian Wolf, Norman May, Thomas Neumann, Alexander Böhm, Anastasia Ailamaki, and Kai-Uwe Sattler. 2014. Scaling up mixed workloads: a battle of data freshness, flexibility, and scheduling. In *Technology Conference on Performance Evaluation and Benchmarking*. Springer, 97–112.

[32] Anil Shanbhag, Samuel Madden, and Xiangyao Yu. 2020. A study of the fundamental performance characteristics of GPUs and CPUs for database analytics. In *Proceedings of the 2020 ACM SIGMOD international conference on Management of data*. 1617–1632.

[33] Anil Shanbhag, Samuel Madden, and Xiangyao Yu. 2020. A Study of the Fundamental Performance Characteristics of GPUs and CPUs for Database Analytics. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data* (Portland, OR, USA) *(SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 1617–1632. https://doi.org/10.1145/3318464.3380595

[34] David Sidler, Zsolt István, Muhsen Owaida, and Gustavo Alonso. 2017. Accelerating pattern matching queries in hybrid CPU-FPGA architectures. In *Proceedings of the 2017 ACM International Conference on Management of Data*. 403–415.

[35] David Sidler, Zsolt István, Muhsen Owaida, and Gustavo Alonso. 2017. Accelerating Pattern Matching Queries in Hybrid CPU-FPGA Architectures. In *Proceedings of the 2017 ACM International Conference on Management of Data* (Chicago, Illinois, USA) *(SIGMOD '17)*. ACM, New York, NY, USA, 403–415. https://doi.org/10.1145/3035918.3035954

[36] Vishal Sikka, Franz Färber, Wolfgang Lehner, Sang Kyun Cha, Thomas Peh, and Christof Bornhövd. 2012. Efficient transaction processing in SAP HANA database: the end of a column store myth. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*. 731–742.

[37] Evangelia Sitaridi, Orestis Polychroniou, and Kenneth A. Ross. 2016. SIMD-accelerated Regular Expression Matching. In *Proceedings of the 12th International Workshop on Data Management on New Hardware* (San Francisco, California) *(DaMoN '16)*. ACM, New York, NY, USA, Article 8, 7 pages. https://doi.org/10.1145/2933349.2933357

[38] Thomas Willhalm, Ismail Oukid, Ingo Müller, and Franz Faerber. 2013. Vectorizing Database Column Scans with Complex Predicates.. In *ADMS@ VLDB*. 1–12.

[39] Thomas Willhalm, Ismail Oukid, Ingo Müller, and Franz Faerber. 2013. Vectorizing Database Column Scans with Complex Predicates. In *ADMS@ VLDB*. 1–12.

[40] Thomas Willhalm, Nicolae Popovici, Yazan Boshmaf, Hasso Plattner, and etc. 2009. SIMD-scan: Ultra Fast In-memory Table Scan Using On-chip Vector Processing Units. *Proc. VLDB Endow.* 2, 1 (Aug. 2009), 385–394. https://doi.org/10.14778/1687627.1687671