Buffered Compares: Excavating the Hidden Parallelism Inside DRAM Architectures with Lightweight Logic

Jinho Lee*, Jung Ho Ahn† and Kiyoung Choi*
*Department of Electrical and Computer Engineering, Seoul National University, Seoul, South Korea
†Department of Transdisciplinary Studies, Seoul National University, Seoul, South Korea
Email: kchoi@snu.ac.kr

***Abstract*—We propose an approach called *buffered compares*, a less-invasive processing-in-memory solution that can be used with existing processor memory interfaces such as DDR3/4 with minimal changes. The approach is based on the observation that multi-bank architecture, a key feature of modern main memory DRAM devices, can be used to provide huge internal bandwidth without any major modification. We place a small buffer and a simple ALU per bank, define a set of new DRAM commands to fill the buffer and feed data to the ALU, and return the result for a set of commands (not for each command) to the host memory controller. By exploiting the under-utilized internal bandwidth using 'compare-n-op' operations, which are frequently used in many applications, we not only reduce the amount of energy-inefficient processor-memory communication, but also accelerate the computation of big data processing applications by utilizing parallelism of the buffered compare units in DRAM banks. Experimental results show that our solution significantly improves the performance and efficiency of the system on the tested workloads.**

I. Introduction

With the emergence of big data applications [1], the centroid of computing paradigm is shifting from computation towards data. Together with the trend of increasing number of cores in a system, the memory bandwidth requirement of a system has steadily increased. However, in contrast to the rapidly growing computing power and bandwidth requirement, actual bandwidth and energy efficiency of off-chip channels are not improving as much, so called the *memory wall* problem [2].

All these circumstances endorse the movement towards the resurgence of processing in memory (*PIM*) [3], [4], which offloads certain computations to processing units near the memory. One way to implement PIM is to add fully functional cores atop DRAM dies with 3D stacking. However, integrating cores with DRAM incurs much overhead and design changes.

By contrast, we leverage existing memory systems to realize PIM with minimal changes to the current ecosystem. Thus, our approach adds minimal amount of computing capability to the memory die for offloading memory-intensive operations while leaving complex or unbounded controls to the processor side. However, considering the gap between internal and external bandwidth of multi-bank DRAM, the approach tries to maximally exploit the excessive internal bandwidth.

To achieve this goal, we focus on compare instructions, mainly targeting table/index scan in in-memory databases [5]. For example, table scan depicted in Fig. 1 searches for a specific data in the given table. With a conventional method, the data are read from the memory to the host processor, and

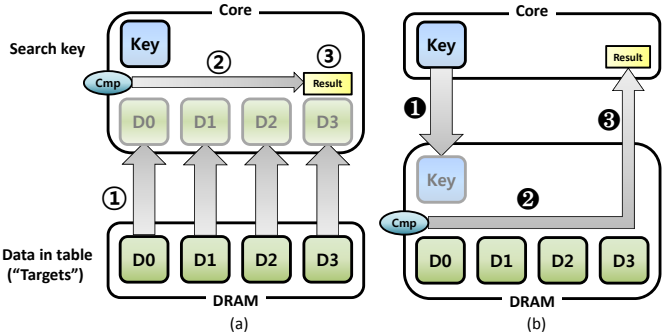

Fig. 1. Scanning in-memory database (a) in a conventional system and (b) in the proposed system with buffered compares.

compares are performed. On the contrary, by executing compare operations of table scan at the memory side, most of the data can be consumed inside the memory, only requiring the internal bandwidth and reducing the off-chip channel access. Similar patterns are found in many big data applications which can be characterized by large footprint, almost no locality, and high memory-boundedness. In those applications, we identify a *compare-n-op* pattern, which performs a compare and an additional operation over a bounded set of data.

In this paper, we propose a novel *buffered compare* scheme, a PIM technique that performs compare-n-op operations inside DRAM banks to speedup many workloads and amplify effective memory bandwidth. In contrast to existing PIM techniques, the buffered compare operations have deterministic latency, so that they can be treated as simple extensions of ordinary DRAM commands, which leaves the DRAM as a 'passive' device (a device that does not invoke any event by itself). Also, without any caches or complicated pipelines of ordinary cores, the buffered compare approach incurs minimal overhead to existing DRAM dies.

Simulation results show that our scheme achieves up to $14.9\times$ speedup and significant energy reduction, at the expense of a minimal increase in the DRAM die area on the tested workloads. Our key contributions are as follows:

- We identify that abundant internal bandwidth unused in modern DRAM architecture provides the opportunity to exploit this extra bandwidth with near-data processing.
- We propose buffered compare architecture that performs compare-n-op operations inside DRAM to provide off-chip bandwidth savings with passive, lightweight logic.
- We suggest a way to solve the system integration issues

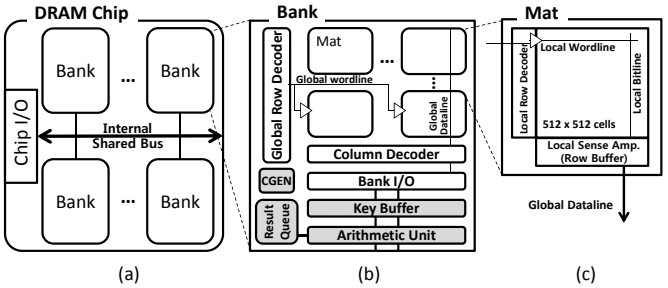

Fig. 2. A high-level view of DRAM architecture. (a) A DRAM chip is composed of multiple banks. (b) Each bank is further divided into an array of mats. (c) A mat has a local row decoder and local sense amplifiers which form a row buffer in conjunction with local sense amplifiers from other mats.

of buffered compare, including programming model, coherence, memory protection and data placement.

- We investigate six workloads that utilize buffered compares to enhance performance and energy efficiency. We also present a detailed circuit-level analysis of buffered compare on performance, power, and area overheads.

II. Background and Motivation

We first illustrate how our idea of buffered compare scheme works by exemplifying the table scan in more detail. We also provide the pertinent details of modern multi-bank DRAM organizations and their operations which the microarchitecture of buffered compare units is based on.

A. Motivational Example

Table scans, the basic operations of database systems, are often critical to the system performance especially for column store databases [5]. When scanning a table, there is a key to search for. The key is compared with the items (called targets) stored in the table. In the conventional system (Fig. 1 (a)), a processor fetches the target data stored in a table (①), performs a compare with the key (②), and generate the compare results (③). In this way, at least tens of cache lines have to be read from the memory to the host processor. However, we only need to know whether each target data matches the key, and the actual value except the match result is totally unnecessary. Even if we use highly-parallel architectures such as GPUs, the amount of memory reads remains the same.

Once we exploit the buffered compares, only two off-chip accesses are required to search for tens of cache blocks. For this, we add a new subsystem called *BCU*s (buffered compare units) per bank in the DRAM die, as shown as gray components in Fig. 2(b). If we perform the compares at the memory side (Fig. 1 (b)), we first send the key to the memory instead of reading the targets (❶), do the comparisons (❷), and read the result once the comparisons are over (❸). This requires one off-chip channel access for writing to the key buffer and another for retrieving the results. Buffered compares not only reduce off-chip bandwidth usage, but also parallelize the compare operations, effectively acting as an accelerator.

B. DRAM Organization and Operations

As shown in Fig. 2, modern DRAM devices are composed of multiple banks, each servicing requests independently from the others (note that the gray boxes represent the newly added components for the proposed technique). A bank is further divided into *mat*s, each containing DRAM cells in a 2D array. A set of local sense amplifiers for the mats in the same row (1D series of mats) are called *row buffer*.

With the row buffers, DRAM achieves high efficiency for sequential accesses. However, activating a new row takes a long time, and thus the performance of random accesses is significantly lower than that of sequential accesses. To alleviate this, the number of banks in a single DRAM device has been increased steadily (eight in DDR3, 16 in DDR4).

On the other hand, when servicing sequential accesses (in a single row), just one bank is enough to supply data for the full bandwidth of the off-chip link. In other words, there is an $8\times$ gap between internal and external bandwidth in DDR3, and thus we are wasting most of the internal bandwidth. Buffered compares efficiently utilize this otherwise wasted bandwidth by performing computations inside the bank.

III. Buffered Compare Architecture

A. Design Philosophy

In the design of the buffered compare scheme for DRAMs, we target its access protocol to be as close as possible to the existing DRAM protocol, and minimize the area overhead. To this end, we make two design choices in architecting the buffered compare unit (BCU). First, every operation of buffered compares has a deterministic latency and is initiated by a DRAM command. Existing conventional DRAM protocols (e.g., DDR3/4) assume that DRAM chips finish all the commands in a pre-determined time. Allowing any non-deterministic latency operation (e.g., conditional branch, pointer traversal) to the BCU would harm the existing DRAM protocol, requiring a whole new protocol design, as exemplified by hybrid memory cube (HMC). By keeping all BCU operations having deterministic latencies, the buffered compare operations can be issued in the form of normal DRAM commands, such as RD, WR, ACT, PRE, and REF. For this purpose, any component that would incur uncertainty, such as caches, branches, or complicated pipelines are avoided in our architecture. This helps not only keep the existing DRAM protocol, but also maintains the logic lightweight.

Second, we tie a BCU and a DRAM bank together. By placing one BCU per bank, we can leverage DRAM's existing data path, and use the bank I/O as an operand register. We restrict the range of data access within a DRAM page as remote accesses inside DRAM are very costly. Using the existing shared bus for remote accesses will easily saturate the global bus bandwidth, while adding extra interconnect would incur significant overhead and disruptive design changes.

B. Buffered Compare Operations

As case studies, we implement the following three compare-n-op operations as shown in Fig. 3, which are frequently used in big data applications.

Compare-n-read compares a range of target data in a bank with the key data in the key buffer and reads the results. It is mainly used for search within data structure. The given range of data are compared with the search key and the results are returned to the processor. The compare results are presented by

```
Compare_n_Read(key, D[n]) :
  res : 2 bits
  for i = 0 to n-1
    res = cmp(D[i], key)
    result.enqueue(res)
  return result
```
(a) compare-n-read

```
Compare_n_Increment(key, D[n]) :
  D[] : array of pair (key, value)
  for i = 0 to n-1
    res = cmp(D[i].key, key)
    if(res == "match") D[i].value++
```
(b) compare-n-increment

```
Compare_n_Select(D[n]) :
  max = 0
  for i = 0 to n-1
    res = cmp(D[i], max)
    if(res == "higher") max = D[i]
  return max
```
(c) compare-n-select

Fig. 3. Three compare-n-op operations used as case studies.

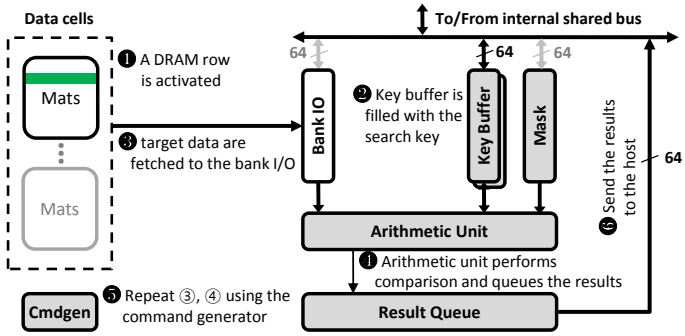

Fig. 4. Architecture and execution flow of a buffered compare in a BCU. Newly added logic blocks for buffered compares are shaded gray. Control Paths and muxes are omitted for clarity.

two bits per item that indicate either match, higher, or lower. The results are stored into the result queue.

Compare-n-increment is used for counting the number of appearances of certain keys. The compare-n-increment operation works on an array of structures with equally sized key and value pair. For example, if the data structure consists of a 32-bit integer key and a 32-bit value, the memory would exhibit a consecutive key-value-key-value pattern. When the target key matches the search key, the value is increased by 1.

Compare-n-select is used for selecting max/min from the given data. The target data are compared with the content of the key buffer, and the larger one is written to the key buffer. Once all the comparison is completed, the content of the key buffer (the max value) is read by the processor.

C. Buffered Compare Unit

Fig. 4 depicts the detailed architecture of the BCU. The widths of all units in the BCU are set to the DRAM access width (64 bytes in our experiment), which is also the width of the bank I/O. Physically, multiple chips function in unison to form a rank, and the DRAM access width is accordingly split into multiples of per-chip data width. Assuming a $\times 8$ device, there are 8 chips per rank, and then the 64-byte datapath is split into 64 bits per chip. For the rest of this paper, we will assume 64 bits for the data width per chip.

Key buffer: The key buffer stores the key to be compared against target items within the access range in the DRAM bank. Each key buffer stores up to 64 bits of data and is filled by an ordinary DRAM WR command. There are two entries of key buffers for double buffering. A memory controller also provides a **mask** for the arithmetic unit in a BCU. It is used to mask out some portion of the data from a DRAM bank that should not be compared against those in the key buffer.

Arithmetic unit: The arithmetic unit performs computations for the buffered compare. It compares the contents in the key buffer and the bank I/O. It also performs corresponding actions: queueing the compare results, conditional increments, and storing the maximum into the key buffer.

Result queue: The result queue is a separate storage dedicated to each bank for the comparison result. It has a queue structure to store the result bits from a series of operations. It stores up to 256 two-bit results. The results can be read by the memory controller with a DRAM RD command.

Command generator: Performing buffered compare operations requires a lot of DRAM commands because the data have to be continuously fetched from the open row to the bank I/O. To save the command bandwidth, we put a simple logic similar to a DMA controller to generate repeated column select and arithmetic unit commands. Still, the latency of all the commands generated over the range is deterministic.

D. Buffered Compare Execution Flow

The execution flow of a compare-n-read operation is illustrated in Fig. 4. First, the DRAM row containing the target data is activated (❶). Second, the key buffer is filled up with the search key (❷); this is done through the off-chip channel, just like a WR command, but the data heads to the key buffer instead of the bank I/O. Then the first 64-bit data is fetched from the cells to the bank I/O (❸). Because this is independent of ❷, the two steps can be overlapped. Once the bank I/O and the key buffer are loaded, the arithmetic unit performs the comparison (❹), and the results are stored into the result queue. Steps ❸ and ❹ are repeated over the determined range (❺). After the repetition is finished, the results stored in the queue is forwarded to the host processor (❻).

The above procedure for other operations is similar. For compare-n-increment, steps ❹ and ❻ are conditional increment and store into the bank I/O, respectively. For compare-n-select, the steps write the maximum to the key buffer and send the key buffer content to the host, respectively.

IV. Integrating Buffered Compares to Systems

A. Challenges for Processing-in-memory Integration

Even though we tried to add minimal overhead to the existing architecture and protocol, there are still some hurdles to overcome to integrate buffered compare to the system. In this section, we will briefly address the challenges and describe the solutions in the following subsections.

First, a programming model is needed to expose buffered compare to the end-users, where we prefer the programmer not to be aware of the detailed DRAM parameters, such as size of a DRAM row, or the number of banks in a rank. It would not only make the programming difficult, but also make the code dependent on the system configuration. The second issue comes with the cache coherence. Because the processor cache might have a copy of the data, it has to be kept coherent with the memory side. While one solution would be to implement a coherence protocol between them,

it would incur too much overhead and offset the benefit from using PIM. Virtual memories give another challenge to the PIM. Because of virtual memories, contiguous range of data might be split over multiple physical pages. Another problem comes from the data placement. In modern DRAM architectures, a rank is usually composed of multiple devices to increase the width of a memory channel. A word is usually interleaved over all the devices within the rank. This raises a problem to buffered compare operations as a full word is needed for processing at a BCU. Lastly, changes are required to the memory controllers. Unlike normal reads/writes, the buffered compare operations are performed over a specified address range, and the memory controller should be carefully to support them.

B. Programming Model

We design our programming model for buffered compare based on the OpenCL framework [6]. As a framework for parallel accelerators, it provides an excellent environment for buffered compares. To support this, we extend the CPU ISA with special instructions, *BCU_compare*, *BCU_cond_inc*, and *BCU_sel*, which have the width same as the SIMD width of the processing unit. Each work item (analogous to a CUDA thread) performs compare for one target item. At the memory controller, the operations from multiple cores are grouped together, divided at the scheduling stage, and sent to the appropriate banks. If the target data range crosses the page boundary, it is split into multiple operations. This choice allows the programmer to be unaware of DRAM parameters, because the memory controller has the parameters available for dividing items into appropriately sized groups.

C. Memory Protection and Coherence

We leverage direct segment [7] with non-cacheable regions for buffered compare operations to solve the protection and coherence issues. Many big data workloads pre-allocate a large chunk of memory at startup, which occupies most of the available memory. Data within this region usually do not fit into the cache hierarchy and do not show much temporal locality, and thus having a cache hierarchy does not give much benefit on them. Also, they rarely take advantage of the traditional paging scheme. We set a base and a limit register to define a "primary region", and keep the region non-cacheable. This way, we eliminate the need for coherence, and the data can be placed contiguously in the physical address space.

D. Data Placement

To solve this problem, we change the data placement for the primary region, as in NDA [8]. In the traditional data placement assuming $\times 8$ DRAM devices, a 64B column is interleaved in an 8-bit granularity. The first device will see the uppermost 8 bits from the eight different 64bit words. Instead, we interleave data in a word (64 bit) granularity for the primary region. Using this, a whole word lies inside a single device, and comparisons can be performed without the inter-chip information. A downside of word interleaving is that it disables the critical word first policy. However, this interleaving is applied only for the primary region, where the data are mostly consumed by the buffered compares and do not benefit from critical-word first policy. Accesses to other regions can still support critical word first policy.

TABLE I
SYSTEM PARAMETERS OF THE MANYCORE CMP

Resource	Value
Core	22nm, OoO, 3GHz, 16 cores, 4 issue
L1 Cache	16KB private I/D, 4-way
L2 Cache	32MB S-NUCA, 16-way
Memory Controller	16GB/s per channel, PAR-BS scheduling, 4 ranks per channel, 16 banks per rank

E. Memory Controller Changes

Buffered compare requires some changes to the memory controller because it is responsible for distributing requests over the devices, and gathering results. We extend existing memory controllers to apply changes needed for our design. Memory controllers usually have a stage for a batch formation, where the outstanding requests are grouped into target banks, and requests to the same row are grouped together to maximize the row-buffer hit rates. At this batch formation stage, the compare-n-op operations from each core are combined to form a ranged operation, which is almost same for forming batches for a row. Then, at the stage where low-level commands are generated, the compare-n-op operations are broken down into commands for buffered compare execution such as filling the key buffer and invoking the arithmetic unit. Note that this is fundamentally not different from generating commands for normal reads and writes that are composed of a few commands such as ACT and RES. Thus the extension can be done without much complexity.

V. EVALUATION

A. Target Applications

1) In-memory Database Scans: **Table scan - column store (TSC)** For in-memory databases, table scans are often used in column store databases. We use compare-n-read to search for a specific key over the full table. We use the data table from SSBM [9] to compare the performance between conventional table scan and scan with buffered compare.

Table scan - row store (TSR) Table scan can also be used in row-store databases. It can also be performed with compare-n-read operations, but a part of the internal bandwidth is wasted by other columns as the columns of interest are far apart.

Max aggregation (MAX) Max aggregation scans over a table and finds the maximum value. When the table is not indexed, all the items in the table will be compared to the temporary maximum value and then the new maximum is stored.

B+ tree scan (BT) B+ tree is a tree structure where each node has a large number of children, used in indexing databases [10] and file systems [11]. The workload from Rodinia [12] benchmark was used. Compare-n-read is used to compare the sequentially stored keys with the search key. For a fair comparison, the baseline system uses binary search.

2) Emerging Big Data Workloads: **Key-value store (KV)** Key-value store is one of the representative big data applications gaining more interest than ever. We used a 4-way cuckoo hashing structure, similar to MemC3 [13]. Compare-n-read is used to look for the keys in a set. To fill the key-value store, the Twitter dataset in CloudSuite [1] was used.

TABLE II
DRAM PARAMETERS OBTAINED FROM SPICE SIMULATION

Spec.		28nm DDR4-2000			
Parameter	Value	Parameter	Value	Parameter	Value
tRCD	14ns	tRAS	34ns	tRP	14ns
tCL	14ns	tCMP	1.4ns	tBL	4ns
E_{ACT}	12.5nJ	E_{IO}	4.0nJ	E_{PRE}	7.5nJ
E_{compare}	0.3pJ	$E_{\text{int\_bank}}$	2.3nJ	$E_{\text{int\_bus}}$	1.9nJ

DNA sequence assembly (SA) Sequence assembly reconstructs the original DNA sequence, from its duplicated, overlapping fragments. To build a k-mer coverage table, a hash table is searched for a specific key (k-mer), and its counter value is incremented by one. This procedure is performed by compare-n-increment with 32-bit key and 32-bit counter value. We used the k-mer coverage building kernel from [14].

B. Simulation Methodology

We evaluated the performance and energy impact of buffered compares on a many-core CMP with simulation parameters shown in Table I. We modified McSimA+ [15] for performance simulation, and McPAT [16] was used for modeling the energy consumption of the cores, caches, and memory controllers at a 22nm logic process. For DRAM timing and energy parameters, SPICE simulation was used. Refer to Table II and Section V-C for details.

C. Overhead Analysis of Buffered Compares

To evaluate the energy overhead and timing parameters of BCUs, we conducted SPICE simulation of the internal DRAM components. The inter-die I/O power values of the off-chip channels were from the latest DDR4 specifications. 28nm DRAM process with 3 metal layers was used, and each 8Gb $\times 8$ device had 16 banks and 8Kb pages with $6F^2$ DRAM cells. A modified version of the PTM low-power model [17] was used for the SPICE simulation. The resulting energy and timing values are tabulated in Table II. E_{int_bank} and E_{int_bus} represent internal access energy within a bank and internal access energy consumed on the bus connecting banks, respectively. Energy consumption of a BCU per access is 0.3pJ, which is almost negligible compared to a data read energy of DRAM, which is 4nJ. BCUs incur a latency of 1.34 ns, which is about 10% of tCL and only applies for buffered compare operations and not for the normal commands. BCUs were modeled with key buffers, masks, arithmetic units, result queues, and command generators. The area of key buffers and comparators in BCUs were modeled using 22T TCAM model [18]. The total area of all BCUs was 0.46 mm^2, which is only 0.53% of the area of an 8Gb DRAM device, where the total area of the DRAM was 86 mm^2 with 50% cell efficiency [19].

D. Experimental Results

We compare our approach with the baseline where all computations are performed in the host processor without any NDP technique. We also compare our approach with active memory operations, which allows memory controllers to perform memory-intensive computations [20], [21]. For this, the memory controllers have modules similar to BCUs. The baseline, active memory operations, and buffered compare are marked as "Baseline", "AMO", and "BC". We assume for active memory operations that there is a sufficient number of computing modules so that there is no contention.

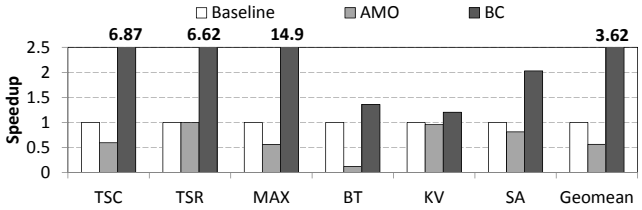

Fig. 5. Speedups of buffered compares over the baseline.

1) Performance Analysis: Fig. 5 shows performance of the buffered compares, which achieve significant speedups over the baseline and active memory operations. For three in-memory DB scan and aggregation workloads (TSC, TSR, and MAX), the speedups are especially high. In TSC, the speedup of BC is $6.87\times$ which comes from transferring only the results of the comparisons. In TSR, the benefit of BC is $6.62\times$, which is slightly lower than TSC, because the table is stored row-wise and the large space between the two items waste internal bandwidth. The MAX workload shows the highest speedup for BC, which is $14.9\times$ compared to the baseline.

In BT, the speedup of BC is 35.9%. In a B+tree node, hundreds of keys are placed sequentially in the DRAM, and therefore looking for the child node can be very efficient using compare-n-read. The speedup of BC on KV and SA workload are 20.5% and $2.03\times$, respectively. Because hashtable searching is finished typically within a few reads, processing a wide range of data with buffered compare on them is not superior.

AMO shows negative gains for all workloads, because AMO does not reduce the number of memory accesses, and it cannot benefit from temporal locality of the on-chip caches. Note that the workload chosen in this work is different from the workloads AMO was proposed for.

2) External and Internal Memory Bandwidth: By internal bandwidth usage, we mean the amount of data read from the bank to the bank I/O or written backwards, either by buffered compare commands or normal read/writes. Fig. 6 shows that the external bandwidth is saturated for most workloads with the baseline, AMO, and BC. Fig. 7 shows an idea why high speedup can be achieved with BC. The internal bandwidth usages have a strong correlation with the speedup achieved over the baseline. The internal bandwidth used by BC is several times higher than the peak external bandwidth, which cannot be obtained without our approach. By contrast, the baseline consumes exactly the same amount of internal bandwidth and external bandwidth, and so does AMO.

3) Energy Consumption: Fig. 8 shows the breakdown of energy consumption normalized to the baseline. BC achieves a large reduction in processor energy, up to 94.0% in MAX, and 73.3% in geometric mean. The reason for such reduction is two-fold: saving in static energy due to the reduced run time and dynamic energy reduction from computation offloaded to the memory. There is no significant difference in energy consumed in the memory, but BC decreases it by a small amount, since the number of off-chip accesses is reduced.

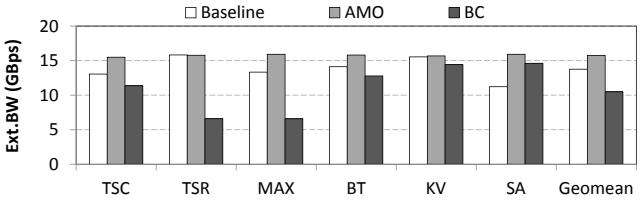

Fig. 6. External bandwidth usage.

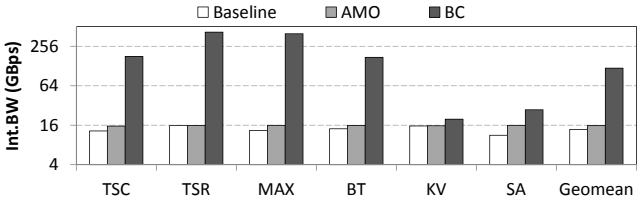

Fig. 7. Internal bandwidth usage. Internal bandwidth is measured by aggregating the amount of data fetched from cells to bank I/O.

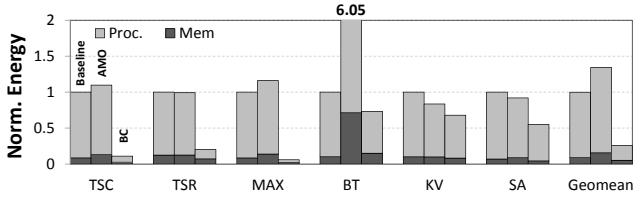

Fig. 8. Normalized energy consumption breakdown.

VI. Related Work

The idea of NDP has come out in the 1990s and there have been a plentiful of studies on integrating logic and DRAM in a single die. EXECUBE [3] was one of the first processing-in-memory (PIM) prototypes. There were numerous follow-up studies including IRAM [4] and smart memories [22]. These PIM approaches usually disruptively re-designed the memory architecture to deploy multiple cores and support communication between them. FlexRAM [23] is closer to our proposal in the sense that FlexRAM chip basically appears as a plain DRAM for applications not compiled for PIM. However, FlexRAM includes a superscalar processor unit and support for inter-chip communications, whereas buffered compare units consist only of a simple logic per bank without inter-unit communications.

Recently, PIM is regaining its interest with the advance of 3D stacking technology. In [24], the impact of using PIM on graph processing was investigated, with cores on the logic die of HMCs. NDA [8] placed a CGRA on top of DRAM. All these differ from our work in that they are all fully-functional, programmable logic with unbounded execution time.

VII. Conclusion

We have proposed buffered compares, which puts light-weight logic inside DRAM banks to perform frequently recurring patterns of big-data workloads. By processing data inside the bank, we can isolate computation and data movement within individual banks. In contrast to other PIM proposals, our scheme requires minimal changes to the existing DDR3/4 memory interfaces. The proposed design adds a negligible area overhead to the DRAM architecture, while increasing the performance up to 14.9× over the baseline, and saving up to 94.0% of energy. Our simple, yet efficient design comes from having deterministic operation latencies, and associating BCUs to their own banks. One limitation of our scheme is that the number of data pins in a device may limit the functionality. With $\times 4$ devices, only up to 32bit data fit into one BCU. Overcoming this limitation along with investigating the application of our technique to 3D memory and adding error correction capability would be our future work.

Acknowledgment

We thank Young Hoon Son on his contributions to SPICE modeling. This work was supported by an IBM Open Collaborative Faculty Award, Samsung Electronics (SNU-490-20150004) and the National Research Foundation of Korea grant funded by the Korea government (NRF-2014R1A2A1A11052936).

References

[1] M. Ferdman *et al.*, "Clearing the clouds: A study of emerging scale-out workloads on modern hardware," in *ASPLOS*, pp. 37–48, 2012.
[2] W. A. Wulf and S. A. McKee, "Hitting the memory wall: Implications of the obvious," *ACM SIGARCH Computer Architecture News*, vol. 23, no. 1, pp. 20–24, 1995.
[3] P. M. Kogge, "EXECUBE-a new architecture for scaleable MPPs," in *ICPP*, pp. 77–84, 1994.
[4] D. Patterson *et al.*, "A case for intelligent RAM," *IEEE Micro*, vol. 17, no. 2, pp. 34–44, 1997.
[5] C. Lemke, K.-U. Sattler, F. Faerber, and A. Zeier, "Speeding up queries in column stores," in *DaWaK*, pp. 117–129, 2010.
[6] J. E. Stone, D. Gohara, and G. Shi, "OpenCL: A parallel programming standard for heterogeneous computing systems," *Computing in Science & Engineering*, vol. 12, no. 3, pp. 66–73, 2010.
[7] A. Basu, J. Gandhi, J. Chang, M. D. Hill, and M. M. Swift, "Efficient virtual memory for big memory servers," in *ISCA*, pp. 237–248, 2013.
[8] A. Farmahini-Farahani, J. Ahn, K. Morrow, and N. S. Kim, "NDA: Near-DRAM acceleration architecture leveraging commodity DRAM devices and standard memory modules," in *HPCA*, pp. 283–295, 2015.
[9] P. O'Neil, E. O'Neil, X. Chen, and S. Revilak, "The star schema benchmark and augmented fact table indexing," in *TPCTC*, pp. 237–252, 2009.
[10] "SQLite." http://sqlite.org.
[11] H. Custer, *Inside the Windows NT File System*. Microsoft Press, 1994.
[12] S. Che *et al.*, "Rodinia: A benchmark suite for heterogeneous computing," in *IISWC*, pp. 44–54, 2009.
[13] B. Fan, D. G. Andersen, and M. Kaminsky, "MemC3: Compact and concurrent memcache with dumber caching and smarter hashing," in *NSDI*, pp. 371–384, 2013.
[14] J. Ahn, "ccTSA: A coverage-centric threaded sequence assembler," *PLoS One*, vol. 7, no. 6, p. e39232, 2012.
[15] J. Ahn, S. Li, O. Seongil, and N. P. Jouppi, "McSimA+: A manycore simulator with application-level+ simulation and detailed microarchitecture modeling," in *ISPASS*, pp. 74–85, 2013.
[16] S. Li *et al.*, "McPAT: An integrated power, area, and timing modeling framework for multicore and manycore architectures," in *MICRO*, pp. 469–480, 2009.
[17] W. Zhao and Y. Cao, "New generation of predictive technology model for sub-45 nm early design exploration," *IEEE Transactions on Electron Devices*, vol. 53, no. 11, pp. 2816–2823, 2006.
[18] R. Sachan *et al.*, "A 40nm 650MHz 0.5 fJ/bit/search TCAM compiler using complementary bit-cell architecture," in *VLSID*, pp. 55–59, 2013.
[19] D. U. Lee *et al.*, "A 1.2 V 8 Gb 8-channel 128 GB/s high-bandwidth memory (HBM) stacked DRAM with effective I/O test circuits," *IEEE Journal of Solid-State Circuits*, vol. 50, no. 1, pp. 191–203, 2015.
[20] Z. Fang, L. Zhang, J. B. Carter, A. Ibrahim, and M. A. Parker, "Active memory operations," in *ICS*, pp. 232–241, 2007.
[21] J. Yoo, S. Yoo, and K. Choi, "Active memory processor for network-on-chip-based architecture," *IEEE Transactions on Computers*, vol. 61, no. 5, pp. 622–635, 2012.
[22] K. Mai *et al.*, "Smart memories: A modular reconfigurable architecture," in *ISCA*, pp. 161–171, 2000.
[23] Y. Kang *et al.*, "FlexRAM: Toward an advanced intelligent memory system," in *ICCD*, pp. 192–201, 1999.
[24] J. Ahn, S. Hong, S. Yoo, O. Mutlu, and K. Choi, "A scalable processing-in-memory accelerator for parallel graph processing," in *ISCA*, pp. 105–117, 2015.