

4-2015

# Fast Bulk Bitwise AND and OR in DRAM

Vivek Seshadri  
*Carnegie Mellon University*

Kevin Hsieh  
*Carnegie Mellon University*

Amirali Boroumand  
*Carnegie Mellon University*

Donghyuk Lee  
*Carnegie Mellon University*

Michael A. Kozuch  
*Intel Research Pittsburgh*

*See next page for additional authors*

Follow this and additional works at: <http://repository.cmu.edu/ece>

 Part of the [Electrical and Computer Engineering Commons](#)

---

## Published In

Computer Architecture Letters, forthcoming.

This Article is brought to you for free and open access by the Carnegie Institute of Technology at Research Showcase @ CMU. It has been accepted for inclusion in Department of Electrical and Computer Engineering by an authorized administrator of Research Showcase @ CMU. For more information, please contact [research-showcase@andrew.cmu.edu](mailto:research-showcase@andrew.cmu.edu).

---

**Authors**

Vivek Seshadri, Kevin Hsieh, Amirali Boroumand, Donghyuk Lee, Michael A. Kozuch, Onur Mutlu, Phillip B. Gibbons, and Todd C. Mowry

# Fast Bulk Bitwise AND and OR in DRAM

Vivek Seshadri\*, Kevin Hsieh\*, Amirali Boroumand\*, Donghyuk Lee\*,  
Michael A. Kozuch†, Onur Mutlu\*, Phillip B. Gibbons†, Todd C. Mowry\*

\*Carnegie Mellon University †Intel Pittsburgh

**Abstract**—Bitwise operations are an important component of modern day programming, and are used in a variety of applications such as databases. In this work, we propose a new and simple mechanism to implement *bulk* bitwise AND and OR operations in DRAM, which is faster and more efficient than existing mechanisms. Our mechanism exploits existing DRAM operation to perform a bitwise AND/OR of two DRAM rows completely within DRAM. The key idea is to simultaneously connect three cells to a bitline before the sense-amplification. By controlling the value of one of the cells, the sense amplifier forces the bitline to the bitwise AND or bitwise OR of the values of the other two cells. Our approach can improve the throughput of bulk bitwise AND/OR operations by  $9.7X$  and reduce their energy consumption by  $50.5X$ . Since our approach exploits existing DRAM operation as much as possible, it requires negligible changes to DRAM logic. We evaluate our approach using a real-world implementation of a bit-vector based index for databases. Our mechanism improves the performance of commonly-used range queries by 30% on average.

## 1 INTRODUCTION

Bitwise operations (e.g., AND, OR) are an important component of modern day programming. They have a wide variety of applications, and can often replace arithmetic operations with more efficient algorithms [13,21]. In fact, most modern processors provide support to accelerate a variety of bitwise operations (e.g., Intel AVX [10]).

We focus our attention on two operations: bitwise AND and bitwise OR. In addition to their uses in simple masking and initializing, these operations are heavily used in databases with bitmap indices [6,17]. Bitmap indices are an alternative to commonly-used data structures like the B-tree. Bitmap indices can be more efficient than B-trees for performing range queries and joins [1,6,22]. These queries frequently perform bitwise AND or OR operations on a large amount of data (multiple KBs/MBs), which we refer to as a *bulk bitwise operation*.

To perform a bulk bitwise operation, existing systems must first read the source data into the processor from main memory and later write the result back to main memory. The drawback of this approach is that it requires a large amount of data to be transferred between the processor and main memory. As a result, existing systems incur high latency, bandwidth, and energy to execute these bulk bitwise operations.

In this work, we propose a new mechanism to perform bulk bitwise AND and OR operations *completely within DRAM*. DRAM stores data in terms of charge in a capacitor. Since the capacitors are very small, a *sense amplifier* is used to detect the charge on a capacitor. Our mechanism exploits the fact that each sense amplifier is connected to *many* DRAM cells. We show that by simultaneously connecting three cells (rather than one) to a sense amplifier, and by controlling the value of one of the cells, the sense amplifier can be forced to perform the bitwise AND or the bitwise OR of the values in the remaining two cells. We develop this idea into a simple mechanism that simultaneously activates three rows of DRAM cells to perform bitwise AND/OR of two of the three rows—essentially a multiple-kilobyte-wide bitwise AND/OR operation.

Our mechanism heavily exploits both the internal operation of DRAM and the external interface with the memory controller. Specifically, the main change required by our mechanism to existing DRAM chips is the support to simultaneously activate three rows within any given subarray. This requires changes to only the row decoding logic, and as a result, our mechanism incurs low implementation cost (Section 3.2).

Manuscript submitted: 23-Mar-2015. Manuscript accepted: 11-Apr-2015. Final manuscript received: 18-Apr-2015.

This work was supported by NSF (awards 0953246, 1212962, and 1320531), and Intel Science and Tech. Center, Samsung, Google, Facebook, and SRC.

We compare the throughput of performing bitwise AND/OR operations using our mechanism to a state-of-the-art baseline mechanism that uses the Intel AVX instructions [10] instead. Our mechanism, which uses DRAM as a co-processor, can achieve  $9.7X$  higher throughput and consume  $50.5X$  lower energy compared to a baseline that has to read all the data from DRAM into the CPU (Section 4). Our evaluation of an open source bitmap index implementation (FastBit [1]) shows that our mechanism can improve performance of commonly-used index-intensive queries by 30% on average (Section 5).

## 2 BACKGROUND ON DRAM OPERATION

At a high level, each DRAM chip consists of multiple banks that can be accessed mostly independently. Each bank is further divided into multiple subarrays [12]. Figure 1 shows the organization of a DRAM subarray. Each subarray consists of multiple rows of DRAM cells connected to an array of *sense amplifiers*. Each row of DRAM cells share a *wordline* that controls the connection between the cells of that row and the sense amplifiers. Similarly, each column of DRAM cells share a *bitline* that connects those cells to the corresponding sense amplifier. Each DRAM cell contains 1) a *capacitor*, and 2) an *access transistor* that acts as a switch between the capacitor and the bitline (as shown in Figure 2). The access transistor is controlled by the wordline of the corresponding DRAM row.

Figure 3 shows the steps involved in DRAM cell access. We start with a fully charged cell in this example. The operation is similar if the cell is empty. In the initial *precharged* state ❶, the bitline is charged to  $\frac{1}{2}V_{DD}$ , and the sense amplifier is disabled (indicated by the *dis* subscript). The cell access is triggered by the *ACTIVATE* command to the corresponding DRAM row, which raises the wordline corresponding to the row ❷. In the phase that follows, known as *charge sharing*, charge flows from the cell to the bitline, raising the voltage level of the bitline to  $\frac{1}{2}V_{DD} + \delta$  (state ❸). In this state, the sense amplifier is enabled (❹), and it *senses* the deviation in the voltage level of the bitline

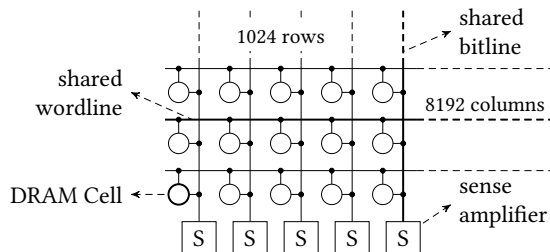


Fig. 1. DRAM subarray. Each circle denotes a DRAM cell, which contains a capacitor and an access transistor.

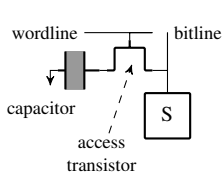


Fig. 2. DRAM cell

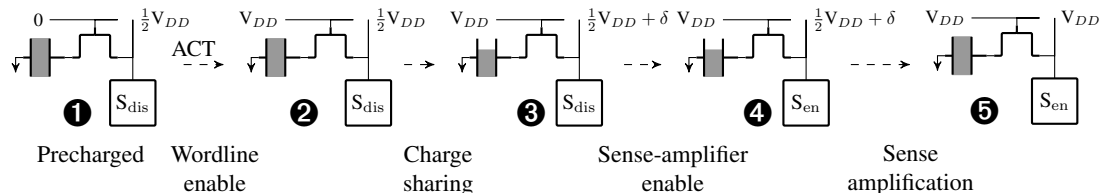


Fig. 3. DRAM cell operation: Steps involved in accessing a DRAM cell

from  $\frac{1}{2}V_{DD}$  and *amplifies* the deviation until the bitline reaches  $V_{DD}$  ⑤. Since the capacitor remains connected to the bitline, the capacitor also gets restored to its original fully charged state. As each wordline is shared by an entire row of DRAM cells, a single activation process transfers the data from *all* the cells of that row to the corresponding sense amplifiers.

After the sense amplification, data can be accessed from the sense amplifiers using the `READ/WRITE` command to the corresponding column of data within the DRAM row. Once the data is accessed, the subarray can be taken back to the precharged state by issuing a `PRECHARGE` command.

The main takeaway from the DRAM cell operation is that the final state of the bitline depends on the deviation of the bitline voltage from  $\frac{1}{2}V_{DD}$  after the charge sharing phase. If the deviation is positive (i.e., towards  $V_{DD}$ ), the bitline is driven to  $V_{DD}$ . On the other hand, if the deviation is negative (i.e., towards 0), the bitline is driven to 0.

### 3 IN-DRAM AND AND OR

Our idea is simple. When three cells are simultaneously connected to a bitline, the deviation on the bitline voltage after *charge sharing* is towards the majority value of the three cells. Specifically, if at least two of the three cells are initially in the charged state, the effective voltage level of the three cells is more than  $\frac{2}{3}V_{DD}$ , resulting in a positive deviation on the bitline voltage. On the other hand, if at most one cell is initially in the charged state, the effective voltage level of the three cells is less than  $\frac{1}{3}V_{DD}$ , resulting in a negative deviation on the bitline voltage. Figure 4 shows an example of this operation with two of the three cells in the charged state ①. As a result, there is a positive deviation on the bitline voltage after charge sharing ②, and sense amplification drives the bitline to  $V_{DD}$ , and all the three cells become fully charged ③.

If  $R$ ,  $A$ , and  $B$  represent the logical states of the three cells, the final state of the bitline after sense amplification is  $RA + RB + AB$  (i.e., at least two values should be 1 for the final state to be 1). Importantly, this expression can be rewritten as  $R(A + B) + R(AB)$ . In other words, if the initial state of  $R$  is 1, the final state of the bitline is the bitwise OR of  $A$  and  $B$ . Otherwise, if the initial state of  $R$  is 0, the final state of the bitline is the bitwise AND of  $A$  and  $B$ . Therefore, by controlling the initial state of the cell  $R$ , we can execute a bitwise AND or bitwise OR operation using the sense amplifier.

There are two challenges with our approach. First, when connecting three cells, the deviation on the bitline (positive or negative) will be lower than the deviation on the bitline when only one cell is connected. For a typical bitline to cell capacitance ratio of 6 [23], the reduction in the deviation is only

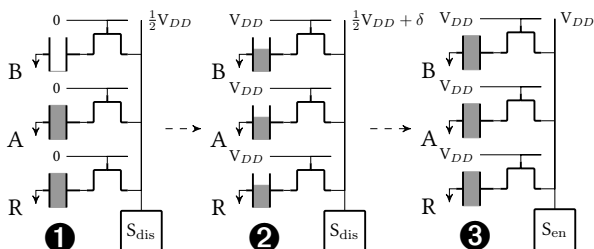


Fig. 4. Simultaneously connecting three cells to the bitline

20% even in the worst case. As we will describe in Section 3.1, our implementation ensures that the cells are fully refreshed before the triple-row activation. Second, our mechanism causes all the three source rows to be overwritten with the result of the bitwise operation.

To ensure that the source data does not get modified, our mechanism first *copies* the data from the two source rows to two temporary rows ( $D1$  and  $D2$ ). Depending on the operation to be performed, our mechanism initializes a third temporary row ( $D3$ ) to 0 or 1, and simultaneously activates the three rows  $D1$ ,  $D2$ , and  $D3$ . It finally copies the result to row  $C$ . For example, to perform a bitwise AND of rows  $A$  and  $B$ , our mechanism performs the following five steps: 1) Copy data from row  $A$  to row  $D1$ , 2) Copy data from row  $B$  to row  $D2$ , 3) Initialize all bits of row  $D3$  to 0, 4) Simultaneously activate  $D1$ ,  $D2$ , and  $D3$ , and 5) Copy result to row  $C$ .

While the above implementation is simple, the copy operations, if performed naively, will nullify the benefits of our mechanism to perform bitwise operations completely in DRAM. Fortunately, our recent prior work, RowClone [20], has proposed two techniques to perform row-to-row copy operations quickly and efficiently within DRAM. The first technique, RowClone-FPM (Fast Parallel Mode), which is the fastest and most efficient (85ns), copies data between two rows within a subarray by simply issuing back-to-back `ACTIVATES` to the source row and the destination row, without an intervening `PRECHARGE`. The second technique, RowClone-PSM (Pipelined Serial Mode), efficiently copies data between banks by overlapping the read to the source bank with the write to the destination bank.

With RowClone, all three copy operations (Steps 1, 2, and 5), and the initialization operation (Step 3) can be performed efficiently within DRAM. To use RowClone for the initialization operation, we reserve two additional rows ( $R0$  and  $R1$ ) that are pre-initialized to 0 and 1, respectively. Depending on the operation to be performed (AND or OR), our mechanism uses RowClone to copy either  $R0$  or  $R1$  to row  $D3$ . Furthermore, to maximize the use of RowClone-FPM, we reserve five rows ( $D1$ ,  $D2$ ,  $D3$ ,  $R0$ , and  $R1$ ) in each subarray.

In the best case, when all the three rows involved in the operation ( $A$ ,  $B$ , and  $C$ ) are present in the same subarray, our mechanism reduces to performing *four* RowClone-FPM operations. The first three steps (two copies and one initialization) each involve one RowClone-FPM. Steps 4 and 5 together involve a single RowClone-FPM, with the first activation being a triple-row activation. In the worst case, when the three rows are in different subarrays within the same bank, the three copy operations (1, 2, and 4/5) need to be performed using RowClone-PSM while the initialization operation (Step 3) can still be performed using RowClone-FPM.

#### 3.1 Positive Aspects of Our Mechanism

Our mechanism has three positive aspects. First, the source data is copied into the rows  $D1$ ,  $D2$ , and  $D3$  just before the triple-row activation. Since each copy operation takes much less than  $1\mu s$  (five orders of magnitude less than the refresh interval), the cells involved in the triple-row activation are very close to the fully refreshed state before the operation. This ensures reliable operation of our mechanism and results in

low impact on the latency of triple-row activation.<sup>1</sup> Second, in any subarray, the triple-row activation is *always* performed on the rows  $D1$ ,  $D2$ , and  $D3$ , as opposed to any three *arbitrary* set of rows within the subarray. These rows can be chosen at *design time* and can be controlled by a separate small row decoder. While using three arbitrary rows for the triple-row activation may require three independent full row decoders, our approach of choosing the three rows at design time and controlling them using a separate small decoder incurs significantly lower cost. The separate row decoder can determine if the three rows are activated individually (for the copy operations) or together (for the triple-row activation). Finally, as we will describe in Section 4, having a separate row decoder for  $D1$ ,  $D2$ , and  $D3$  allows us to further accelerate the copy operations using a more aggressive copy mechanism.

### 3.2 Changes to DRAM

Our mechanism requires three changes to the DRAM logic and interface. First, as mentioned in Section 3.1, our mechanism requires each subarray to have a separate small row decoder to control the three reserved rows ( $D1$ ,  $D2$ , and  $D3$ ). Second, our mechanism requires a variant of the `ACTIVATE` command through which the memory controller can communicate the triple-row activation to DRAM. Finally, our mechanism requires the DRAM and the controller to support RowClone [20]. RowClone itself incurs very low cost (less than 0.01% die area). In addition to these changes, our mechanism also reserves five rows in each subarray. For a typical subarray with 1024 rows [12], this results in less than 0.5% loss in memory capacity.

### 3.3 Processor and Software Support

The processor support required by our mechanism is similar to the support required by RowClone [20]. The processor must 1) provide instructions with which the software can communicate occurrences of bulk bitwise operations, 2) support cache coherence for data read and modified directly in DRAM, and 3) communicate to software the minimum size of the bitwise operation that can be accelerated by DRAM (depending on how the physical address space is interleaved on to the DRAM hierarchy). The application must be modified to use the new bulk bitwise instructions. Fortunately, this may require changes only to some libraries (e.g., FastBit [1]) that are shared by many applications. These libraries are generally optimized to exploit any available hardware acceleration.

## 4 LATENCY, THROUGHPUT, AND ENERGY ANALYSIS

As our goal in this paper is to demonstrate the potential of our mechanism, we focus our attention on the case when we can use RowClone-FPM to perform all the copy operations.

**Latency.** To complete an intra-subarray copy, RowClone-FPM uses two `ACTIVATE`s (back-to-back) followed by a `PRECHARGE` operation. Assuming typical DRAM timing parameters ( $t_{RAS} = 35ns$  and  $t_{RP} = 15ns$ ), each copy operation consumes  $85ns$ . As our mechanism is essentially four RowClone-FPM operations (as described in the previous section), the overall latency of a bitwise AND/OR operation is  $4 \times 85ns = 340ns$ .

In a RowClone-FPM operation, although the second `ACTIVATE` does not involve any sense amplification (the sense amplifiers are already activated), the RowClone paper [20] assumes the `ACTIVATE` consumes the full  $t_{RAS}$  latency. However, by controlling the rows  $D1$ ,  $D2$ , and  $D3$  using a separate row decoder, it is possible to overlap the `ACTIVATE` to the destination fully with the `ACTIVATE` to the source row, by raising the wordline of the destination row towards the end of the sense amplification of the source row. This mechanism

is similar to the inter-segment copy operation described in Tiered-Latency DRAM [15] (Section 4.4). With this aggressive mechanism, the latency of a RowClone-FPM operation reduces to  $50ns$  (one `ACTIVATE` and one `PRECHARGE`). Therefore, the overall latency of a bitwise AND/OR operation is  $200ns$ . We will refer to this enhanced mechanism as *aggressive*, and the approach that uses the simple back-to-back `ACTIVATE` operations as *conservative*.

**Throughput.** To compare the potential throughput of bitwise AND/OR operations enabled by our mechanism to a baseline system, we created a benchmark that repeatedly performs bitwise AND of two vectors of a given size ( $S$ ), and stores the result in a third vector (the throughput results with bitwise OR are same as that of bitwise AND). We ran this microbenchmark (with various values of  $S$ ) on a single-socket Intel Core i7-4790K processor connected to two 8GB DDR3-1333 DIMMs each on a different channel. Our baseline implementation uses the AVX support [10] provided by the x86 ISA to perform bitwise AND operations. The processor has a 32KB L1 cache, 256KB L2 cache, and an 8MB L3 cache. The throughput of the baseline heavily depends on whether the working set of the benchmark fits in the L1, L2 or the L3 cache.

Figure 5 plots the results of this experiment. The x-axis shows the value of  $S$  (i.e., vector size) and the y-axis plots the throughput of the bitwise operation in terms of gigabytes of AND (or OR) operations per second. The figure also plots the estimated throughput of our conservative and aggressive mechanisms. There are three conclusions we can draw from the figure.

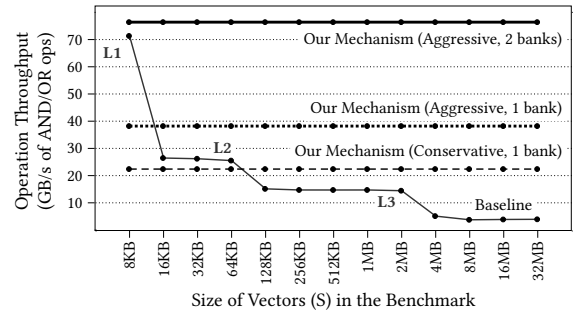


Fig. 5. Throughput comparison with Intel AVX extensions

First, as the vector size ( $S$ ) increases, the throughput of the baseline system drops — with sharp dips at points (highlighted in the figure) when the cumulative working set no longer fits in a particular level of cache. Note that the working set is roughly three times  $S$ . While the baseline starts with an impressive 71 GB/s of AND operations when the working set fits the L1 cache, the throughput drops to 3.9 GB/s when the working set does not fit into any level of on-chip cache.

Second, both of our mechanisms significantly outperform the baseline throughput when the working set of the application does not fit into any on-chip cache (e.g.,  $S \geq 4MB$ ). The throughput of our conservative mechanism (22.4 GB/s) is over 5X the baseline throughput and that of our aggressive mechanism (38.2 GB/s) is close to 10X the baseline throughput.

Third, although we run our benchmark on only a single core, the throughput of the baseline is limited by the available memory bandwidth. In sharp contrast, the throughput of our mechanism scales linearly with the number of banks, since our mechanism can be employed simultaneously in many DRAM banks. In other words, with just two DRAM banks, our mechanism can achieve a throughput of 76.4 GB/s of AND/OR operations, whereas the throughput of the baseline, even with multiple cores, will not be much better than 3.9 GB/s.

**Energy.** We use the Rambus power model [19] to estimate the energy consumption of our mechanisms and the baseline. For

1. As described in Section 5, even if we conservatively assume a 2X higher latency for the triple-row activation, our mechanism yields 20% performance improvement on our database benchmark.

the baseline, our estimate only includes the energy of accessing DRAM and *does not* include the energy consumed in accessing the on-chip caches and performing the computation. Based on our estimates, our *conservative* mechanism reduces energy consumption by 31.6X, and our *aggressive* mechanism reduces energy consumption by 50.5X compared to the baseline.

In summary, our mechanism is significantly more efficient than the baseline system, providing more than an order of magnitude improvement in throughput and reduction in energy consumption of bulk bitwise AND/OR operations.

## 5 ANALYSIS OF A REAL-WORLD BITMAP INDEX

We analyze our mechanism's performance on a real-world bitmap index library, FastBit [1], widely-used in physics simulations and network analysis. Fastbit can enable faster and more efficient searching/retrieval compared to B-trees.

To construct an index, FastBit uses multiple bitmap bins, each corresponding to either a distinct value or a range of values. FastBit relies on fast bitwise AND/OR operations on these bitmaps to accelerate *joins* and *range queries*. For example, to execute a range query, FastBit performs a bitwise OR of all bitmaps that correspond to the specified range.

We initialized FastBit on our baseline system using the sample STAR [3] data set. We then ran a set of indexing-intensive range queries that touch various numbers of bitmap bins. For each query, we measure the fraction of query execution time spent on bitwise OR operations. Table 1 shows the corresponding results. For each query, the table shows the number of bitmap bins involved in the query and the percentage of time spent in bitwise OR operations. On average, 31% of the query execution is spent on bitwise OR operations (with small variance across queries).

TABLE 1. Fraction of time spent in OR operations

Number of bins	3	9	20	45	98	118	128
Fraction of time spent in OR	29%	29%	31%	32%	34%	34%	34%

To estimate the performance of our mechanism, we measure the number of bitwise OR operations required to complete the query. We then compute the amount of time taken by our mechanism to complete these operations and then use that to estimate the performance of the overall query execution. To perform a bitwise OR of more than two rows, our mechanism is invoked two rows at a time. Figure 6 shows the potential performance improvement using our two mechanisms (conservative and aggressive), each with either 1 bank or 4 banks.

As our results indicate, our aggressive mechanism with 4 banks improves the performance of range queries by 30% (on average) compared to the baseline, eliminating almost all the overhead of bitwise operations. As expected, the aggressive mechanism performs better than the conservative mechanism. Similarly, using more banks provides better performance. Even if we assume a 2X higher latency for the triple-row activation, our conservative mechanism with 1 bank improves performance by 18% (not shown in the figure).

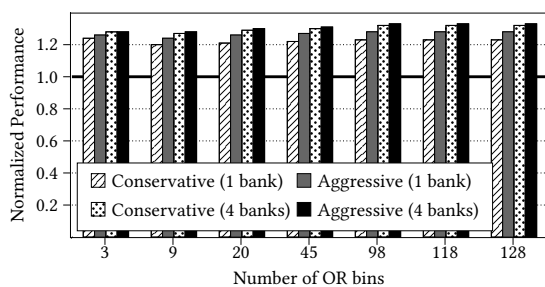


Fig. 6. Range query performance improvement over baseline

## 6 RELATED WORK

To our knowledge, this is the first work to exploit the analog operation of DRAM to perform bitwise operations. Many works (e.g., EXECUBE [14], IRAM [18], DIVA [7]) have proposed mechanisms to integrate logic on the DRAM chip to perform bandwidth-intensive operations. The main limitation of such approaches is that adding large amounts of logic to DRAM significantly increases DRAM cost. Some new DRAM architectures [2,11,16] contain a logic layer. Many techniques (e.g., [4,5,8,9,24]) have been proposed to exploit the logic layer to implement some computation close to DRAM. However, these architectures still have limited memory bandwidth available to the logic layer, and our mechanism can provide much higher throughput and efficiency for bitwise AND/OR operations. Therefore, the spare space in the logic layer can be used to implement *other* functionality, while our mechanism can still be used to implement bitwise AND/OR operations.

## 7 CONCLUSION

We presented a new and simple mechanism to perform bulk bitwise AND/OR operations completely within DRAM. Our mechanism exploits the analog operation of modern DRAM, and hence incurs low implementation cost. Our evaluations show that our approach can potentially increase throughput and energy-efficiency of bitwise AND/OR operations by more than an order of magnitude compared to the baseline, and significantly improve the performance of real-world applications.

## REFERENCES

- [1] FastBit: An Efficient Compressed Bitmap Index Technology. <https://sdm.lbl.gov/fastbit/>.
- [2] High Bandwidth Memory DRAM. <http://www.jedec.org/standards-documents/docs/jesd235>.
- [3] The STAR experiment. <http://www.star.bnl.gov/>.
- [4] J. Ahn et al. A Scalable Processing-in-Memory Accelerator for Parallel Graph Processing. In *ISCA*, 2015.
- [5] J. Ahn et al. PIM-Enabled Instructions: A Low-Overhead, Locality-Aware Processing-in-Memory Architecture. In *ISCA*, 2015.
- [6] C. Chan and Y. E. Ioannidis. Bitmap Index Design and Evaluation. In *SIGMOD*, 1998.
- [7] J. Draper et al. The Architecture of the DIVA Processing-in-memory Chip. In *ICS*, 2002.
- [8] A. F-Farahani et al. NDA: Near-DRAM acceleration architecture leveraging commodity DRAM devices and standard memory modules. In *HPCA*, 2015.
- [9] Q. Guo et al. 3D-Stacked Memory-Side Acceleration: Accelerator and System Design. In *WoNDP*, 2013.
- [10] Intel. Intel Instruction Set Architecture Extensions. <https://software.intel.com/en-us/intel-isa-extensions>.
- [11] J. Jeddelloh and B. Keeth. Hybrid Memory Cube: New DRAM Architecture Increases Density and Performance. In *VLSIT*, 2012.
- [12] Y. Kim et al. A Case for Exploiting Subarray-level Parallelism (SALP) in DRAM. In *ISCA*, 2012.
- [13] D. E. Knuth. The Art of Computer Programming. Fascicle 1: Bitwise Tricks & Techniques; Binary Decision Diagrams, 2009.
- [14] P. M. Kogge. EXECUBE - A new architecture for scalable MPPs. In *ICPP*, 1994.
- [15] D. Lee et al. Tiered-Latency DRAM: A Low Latency and Low Cost DRAM Architecture. In *HPCA*, 2013.
- [16] G. H. Loh. 3D-Stacked Memory Architectures for Multi-core Processors. In *ISCA*, 2008.
- [17] E. O'Neil et al. Bitmap Index Design Choices and Their Performance Implications. In *IDEAS*, 2007.
- [18] D. Patterson et al. A case for Intelligent RAM. *IEEE Micro*, 1997.
- [19] Rambus. DRAM Power Model, 2010.
- [20] V. Seshadri et al. RowClone: Fast and Energy-efficient in-DRAM Bulk Data Copy and Initialization. In *MICRO*, 2013.
- [21] Henry S Warren. *Hacker's Delight*. Pearson Education, 2012.
- [22] K. Wu et al. Compressing bitmap indexes for faster search operations. In *SSDBM*, 2002.
- [23] Y. Yanagawa et al. In-substrate-bitline sense amplifier with array-noise-gating scheme for low-noise 4F2 DRAM array operable at 10-fF cell capacitance. In *VLSI*, 2011.
- [24] D. Zhang et al. TOP-PIM: Throughput-oriented Programmable Processing in Memory. In *HPDC*, 2014.