# Ambit: In-Memory Accelerator for Bulk Bitwise Operations Using Commodity DRAM Technology

Vivek Seshadri[1,5] Donghyuk Lee[2,5] Thomas Mullins[3,5] Hasan Hassan[4] Amirali Boroumand[5]
Jeremie Kim[4,5] Michael A. Kozuch[3] Onur Mutlu[4,5] Phillip B. Gibbons[5] Todd C. Mowry[5]

[1]Microsoft Research India [2]NVIDIA Research [3]Intel [4]ETH Zürich [5]Carnegie Mellon University

## Abstract

Many important applications trigger *bulk bitwise operations*, i.e., bitwise operations on large bit vectors. In fact, recent works design techniques that exploit fast bulk bitwise operations to accelerate databases (bitmap indices, BitWeaving) and web search (BitFunnel). Unfortunately, in existing architectures, the throughput of bulk bitwise operations is limited by the memory bandwidth available to the processing unit (e.g., CPU, GPU, FPGA, processing-in-memory).

To overcome this bottleneck, we propose *Ambit*, an Accelerator-in-Memory for bulk bitwise operations. Unlike prior works, Ambit exploits the analog operation of DRAM technology to perform bitwise operations *completely inside DRAM*, thereby exploiting the full internal DRAM bandwidth. Ambit consists of two components. First, simultaneous activation of three DRAM rows that share the same set of sense amplifiers enables the system to perform bitwise AND and OR operations. Second, with modest changes to the sense amplifier, the system can use the inverters present inside the sense amplifier to perform bitwise NOT operations. With these two components, Ambit can perform *any* bulk bitwise operation efficiently inside DRAM. Ambit largely exploits existing DRAM structure, and hence incurs low cost on top of commodity DRAM designs (1% of DRAM chip area). Importantly, Ambit uses the modern DRAM interface *without any changes*, and therefore it can be directly plugged onto the memory bus.

Our extensive circuit simulations show that Ambit works as expected even in the presence of significant process variation. Averaged across seven bulk bitwise operations, Ambit improves performance by 32X and reduces energy consumption by 35X compared to state-of-the-art systems. When integrated with Hybrid Memory Cube (HMC), a 3D-stacked DRAM with a logic layer, Ambit improves performance of bulk bitwise operations by 9.7X compared to processing in the logic layer of the HMC. Ambit improves the performance of three real-world data-intensive applications, 1) database bitmap indices, 2) BitWeaving, a technique to accelerate database scans, and 3) bit-vector-based implementation of sets, by 3X-7X compared to a state-of-the-art baseline using SIMD optimizations. We describe four other applications that can benefit from Ambit, including a recent technique proposed to speed up web search. We believe that large performance and energy improvements provided by Ambit can enable other applications to use bulk bitwise operations.

### CCS CONCEPTS

• **Computer systems organization** → Single instruction, multiple data; • **Hardware** → Hardware accelerator; • **Hardware** → Dynamic memory;

### KEYWORDS

Bulk Bitwise Operations, Processing-in-memory, DRAM, Memory Bandwidth, Performance, Energy, Databases

### ACM Reference Format:

Vivek Seshadri, Donghyuk Lee, Thomas Mullins, Hasan Hassan, Amirali Boroumand, Jeremie Kim, Michael A. Kozuch, Onur Mutlu, Phillip B. Gibbons, and Todd C. Mowry. 2017. Ambit: In-Memory Accelerator for Bulk Bitwise Operations Using Commodity DRAM Technology. In Proceedings of Annual IEEE/ACM International Symposium on Microarchitecture, Cambridge, MA, USA, October 2017 (MICRO-50), 15 pages. http://dx.doi.org/10.1145/3123939.3124544

## 1. Introduction

Many applications trigger *bulk bitwise operations*, i.e., bitwise operations on large bit vectors [61, 108]. In databases, bitmap indices [26, 85], which heavily use bulk bitwise operations, are more efficient than B-trees for many queries [3, 26, 111]. In fact, many real-world databases [3, 8, 10, 11] support bitmap indices. A recent work, WideTable [76], designs an entire database around a technique called BitWeaving [75], which accelerates *scans* completely using bulk bitwise operations. Microsoft recently open-sourced a technology called BitFunnel [40] that accelerates the document filtering portion of web search. BitFunnel relies on fast bulk bitwise AND operations. Bulk bitwise operations are also



MICRO-50, October 14-18, 2017, Cambridge, MA, USA

ACM. ISBN 978-1-4503-4034-2/15/12...$15.00.
DOI: http://dx.doi.org/10.1145/3123939.3124544

prevalent in DNA sequence alignment [20, 21, 113], encryption algorithms [44, 83, 107], graph processing [74], and networking [108]. Thus, accelerating bulk bitwise operations can significantly boost the performance of various applications.

In existing systems, a bulk bitwise operation requires a large amount of data to be transferred on the memory channel. Such large data transfers result in high latency, bandwidth, and energy consumption. In fact, our experiments on a multi-core Intel Skylake [7] and an NVIDIA GeForce GTX 745 [4] show that the available memory bandwidth of these systems limits the throughput of bulk bitwise operations. Recent works (e.g., [16, 17, 24, 37, 42, 47, 48, 115]) propose processing in the logic layer of 3D-stacked DRAM, which stacks DRAM layers on top of a logic layer (e.g., Hybrid Memory Cube [6, 51]). While the logic layer in 3D-stacked memory has much higher bandwidth than traditional systems, it still cannot exploit the maximum internal bandwidth available inside a DRAM chip (Section 7).

To overcome this memory bandwidth bottleneck, we propose a new **A**ccelerator-in-**M**emory for bulk **Bit**wise operations (Ambit). Unlike prior approaches, Ambit uses the analog operation of DRAM technology to perform bulk bitwise operations *completely inside the memory array*. With modest changes to the DRAM design, Ambit can exploit 1) the maximum internal bandwidth available *inside* each DRAM array, and 2) the memory-level parallelism *across* multiple DRAM arrays to significantly improve the performance and efficiency of bulk bitwise operations.

Ambit consists of two parts, Ambit-AND-OR and Ambit-NOT. We propose two new ideas that exploit the operation of the *sense amplifier*, the circuitry that is used to extract data from the DRAM cells. First, in modern DRAM, many rows of DRAM cells (typically 512 or 1024) share a single set of sense amplifiers [27, 59, 68, 97]. **Ambit-AND-OR** exploits the fact that simultaneously activating three rows (rather than one) results in a bitwise majority function across the cells in the three rows. We refer to this operation as *triple-row activation*. We show that by controlling the initial value of one of the three rows, we can use the triple-row activation to perform a bitwise AND or OR of the other two rows. Second, each sense amplifier has two inverters. **Ambit-NOT** uses a row of *dual-contact cells* (a 2-transistor 1-capacitor cell [53, 81]) that connects to both sides of the inverters to perform bitwise NOT of any row of DRAM cells. With the ability to perform AND, OR, and NOT operations, Ambit can perform *any* bulk bitwise operation completely using DRAM technology. Our circuit simulation results show that Ambit-AND-OR and Ambit-NOT work reliably, even in the presence of significant process variation. Sections 3 and 4 describe these two parts of Ambit.

A naïve mechanism to support triple-row activation on three *arbitrary* rows would require a wider off-chip address bus and multiple row decoders to simultaneously communicate and decode three arbitrary addresses. Such a mechanism would incur high cost. We present a practical, low-cost mechanism that employs three ideas. First, we restrict triple-row activations to be performed *only* on a *designated* set of rows (chosen at design time). Before performing Ambit-AND-OR, our mechanism copies data from the source rows into the designated rows. After the operation is completed, it copies the result into the destination row. We exploit a recent work, RowClone [97] to perform the required copy operations efficiently inside the DRAM arrays (Section 3.4). Second, we reserve a few DRAM row addresses and map each of them to triple-row activation on a predefined set of three designated rows. With this approach, the memory controller can communicate a triple-row activation with a single address, thereby eliminating the need for a wider address bus (Section 5.1). Third, we split the row decoder into two parts: one small part that handles all activations to the *designated* rows, and another part to handle activations to regular data rows. This split-row decoder significantly reduces the complexity of changes required to the row decoder design (Section 5.3).

Our implementation of Ambit has three advantages. First, unlike prior systems that are limited by the external DRAM data bandwidth, the performance of Ambit scales linearly with the maximum *internal* bandwidth of DRAM (i.e., row buffer size) and the memory-level parallelism available inside DRAM (i.e., number of banks or subarrays [59]). Second, our implementation does not introduce *any* changes to the DRAM command and address interface. As a result, Ambit can be *directly* plugged onto the system memory bus, allowing a design where applications can *directly* trigger Ambit operations using processor instructions rather than going through a device interface like other accelerators (e.g., GPU). Third, since almost all DRAM technologies (e.g., Hybrid Memory Cube [6, 51], and High-Bandwidth Memory [5, 70]) use the same underlying DRAM microarchitecture [60], Ambit can be integrated with any of these DRAM technologies.

We compare the raw throughput and energy of performing bulk bitwise operations using Ambit to 1) an Intel Skylake [7], 2) an NVIDIA GTX 745 [4], and 3) performing operations on the logic layer of a state-of-the-art 3D-stacked DRAM, HMC 2.0 [6, 51]. Our evaluations show that the bulk bitwise operation throughput of these prior systems is limited by the memory bandwidth. In contrast, averaged across seven bitwise operations, Ambit with 8 DRAM banks improves bulk bitwise operation throughput by 44X compared to Intel Skylake and 32X compared to the GTX 745. Compared to the DDR3 interface, Ambit reduces energy consumption of these operations by 35X on average. Compared to HMC 2.0, Ambit improves bulk bitwise operation throughput by 2.4X. When integrated directly into the HMC 2.0 device, Ambit improves throughput by 9.7X compared to processing in the logic layer of HMC 2.0. Section 7 discusses these results.

Although the Ambit accelerator incurs low cost on top of existing DRAM architectures, we do *not* advocate Ambit as a replacement for *all* commodity DRAM devices. Rather, certain important applications that run in large data centers (e.g., databases, web search, genome analysis) can benefit significantly from increased throughput for bulk bitwise operations.

For such applications, we believe Ambit can improve the scalability and reduce the overall cost of the system (e.g., by requiring fewer servers). We evaluate the performance of Ambit on three real-world data-intensive applications over a state-of-the-art baseline that employs SIMD optimization. First, Ambit improves end-to-end performance of database queries that use bitmap indices [3] by 6.0X, averaged across a range of query parameters. Second, for BitWeaving [75], a recently-proposed technique to accelerate column scan operations in databases, Ambit improves performance by 7.0X, averaged across a range of scan parameters. Third, for the commonly-used *set* data structure, Ambit improves performance of set intersection, union, and difference operations by 3.0X compared to existing implementations [41]. Section 8 describes our full-system simulation framework [22], workloads, results, and four other applications that can significantly benefit from Ambit: BitFunnel [40], masked initialization, encryption algorithms, and DNA read mapping.

We make the following **contributions** in this work.

- To our knowledge, this is the first work that integrates support for bulk bitwise operations directly into a DRAM memory array. We introduce Ambit, an in-memory accelerator that exploits the analog operation of DRAM to perform bulk bitwise operations with high throughput and efficiency.
- We present a low-cost implementation of Ambit, which requires modest changes to the commodity DRAM architecture (less than 1% DRAM chip area overhead). We verify our implementation of Ambit with rigorous circuit simulations. Ambit requires no changes to the DRAM command and address interface, and hence, can be directly plugged onto the system memory bus (Section 5).
- Our evaluations show that Ambit significantly improves the throughput and energy efficiency of bulk bitwise operations compared to state-of-the-art CPUs, GPUs, and processing-in-memory systems using 3D-stacked DRAM. This directly translates to large performance improvements for three real-world applications that use bulk bitwise operations.

## 2. Background on DRAM Operation

Our Ambit accelerator exploits the internal high-bandwidth operation of the modern DRAM chips and tightly integrates into the DRAM architecture. In this section, we provide the necessary background on DRAM operation.

A DRAM-based memory hierarchy consists of *channels*, *modules*, and *ranks* at the top level. Each rank consists of a set of chips that operate in unison. Each rank is further divided into many *banks*. All access-related commands are directed towards a specific bank. Each bank consists of several subarrays and peripheral logic to process commands [27, 29, 59, 60, 67, 68, 97, 116]. Each subarray consists of many rows (typically 512 or 1024) of DRAM cells, a row of *sense amplifiers*, and a *row address decoder*. Figure 1 shows the logical organization of a subarray.[1]

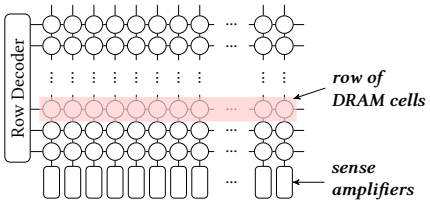


**Figure 1: Logical organization of a DRAM subarray**

At a high level, accessing data from a subarray involves three steps. The first step, *row activation*, copies data from a specified row of DRAM cells to the row of sense amplifiers in the subarray. This step is triggered by the `ACTIVATE` command. Then, data is accessed from the sense amplifiers using a `READ` or `WRITE` command. Each `READ` or `WRITE` accesses only a subset of the sense amplifiers. Once a row is activated, multiple `READ` and `WRITE` commands can be issued to that row. An activated bank is prepared for an access to another row by an operation called *precharging*. This step is triggered by the `PRECHARGE` command. We now explain these operations by focusing on a single DRAM cell and a sense amplifier.

Figure 2 shows the connection between a DRAM cell and a sense amplifier. Each DRAM cell consists of 1) a *capacitor*, and 2) an *access transistor* that controls access to the cell. Each sense amplifier consists of two inverters, and an *enable* signal. The output of each inverter is connected to the input of the other inverter. The wire that connects the cell to the sense amplifier is called the *bitline*, and the wire that controls the

[1]Although the figure logically depicts a subarray as a single monolithic structure, in practice, each subarray is divided into several MATs. The row decoding functionality is also split between a bank-level global row decoder, a subarray-local row decoder, and wordline drivers [50, 59, 71]. While we describe our mechanisms on top of the logical organization, they can be easily engineered to work with the actual physical design.

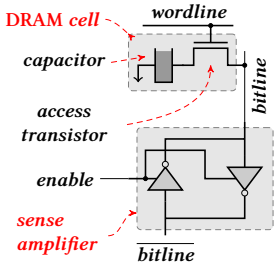


**Figure 2: DRAM cell and sense amplifier**

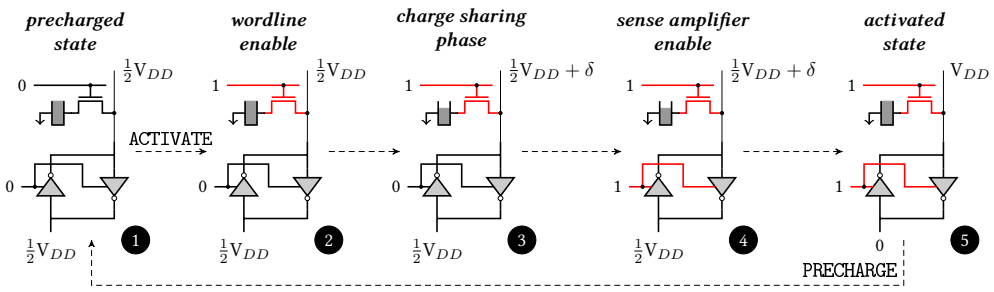


**Figure 3: State transitions involved in DRAM cell activation**

access transistor is called the *wordline*. We refer to the wire on the other end of the sense amplifier as bitline ("bitline bar").

Figure 3 shows the state transitions involved in extracting the state of the DRAM cell. In this figure, we assume that the cell capacitor is initially charged. The operation is similar if the capacitor is initially empty. In the initial *precharged* state ❶, both the bitline and bitline are maintained at a voltage level of $\frac{1}{2}V_{DD}$. The sense amplifier and the wordline are disabled.

The `ACTIVATE` command triggers an access to the cell. Upon receiving the `ACTIVATE`, the wordline of the cell is raised ❷, connecting the cell to the bitline. Since the capacitor is fully charged, and thus, at a higher voltage level than the bitline, charge flows from the capacitor to the bitline until both the capacitor and the bitline reach the same voltage level $\frac{1}{2}V_{DD} + \delta$. This phase is called *charge sharing* ❸. After charge sharing is complete, the sense amplifier is enabled ❹. The sense amplifier senses the difference in voltage level between the bitline and bitline. The sense amplifier then amplifies the deviation to the stable state where the bitline is at the voltage level of $V_{DD}$ (and the bitline is at 0). Since the capacitor is still connected to the bitline, the capacitor also gets fully charged (i.e., *restored*) ❺. If the capacitor was initially empty, then the deviation on the bitline would be negative (towards 0), and the sense amplifier would drive the bitline to 0. Each `ACTIVATE` command operates on an *entire row* of cells (typically 8 KB of data across a rank).

After the cell is activated, data can be accessed from the bitline by issuing a `READ` or `WRITE` to the column containing the cell (not shown in Figure 3; see [28, 45, 59, 67, 68, 71] for details). When data in a different row needs to be accessed, the memory controller takes the subarray back to the initial precharged state ❶ using the `PRECHARGE` command. Upon receiving this command, DRAM first lowers the raised wordline, thereby disconnecting the capacitor from the bitline. After this, the sense amplifier is disabled, and both the bitline and the bitline are driven to the voltage level of $\frac{1}{2}V_{DD}$.

## 3. Ambit-AND-OR

The first component of our mechanism, Ambit-AND-OR, uses the analog nature of the charge sharing phase to perform bulk bitwise AND and OR directly in DRAM. It specifically exploits two facts about DRAM operation:

1. In a subarray, each sense amplifier is shared by many (typically 512 or 1024) DRAM cells on the same bitline.
2. The final state of the bitline after sense amplification depends primarily on the voltage deviation on the bitline after the charge sharing phase.

Based on these facts, we observe that simultaneously activating three cells, rather than a single cell, results in a *bitwise majority function*—i.e., at least two cells have to be fully charged for the final state to be a logical "1". We refer to simultaneous activation of three cells (or rows) as *triple-row activation*. We now conceptually describe triple-row activation and how we use it to perform bulk bitwise AND and OR operations.

### 3.1. Triple-Row Activation (TRA)

A triple-row activation (TRA) simultaneously connects a sense amplifier with three DRAM cells on the same bitline. For ease of conceptual understanding, let us assume that the three cells have the same capacitance, the transistors and bitlines behave ideally (no resistance), and the cells start at a fully refreshed state. Then, based on charge sharing principles [57], the bitline deviation at the end of the charge sharing phase of the TRA is:

$$\begin{aligned} \delta &= \frac{k.C_c.V_{DD} + C_b.\frac{1}{2}V_{DD}}{3C_c + C_b} - \frac{1}{2}V_{DD} \\ &= \frac{(2k-3)C_c}{6C_c + 2C_b}V_{DD} \end{aligned} \quad (1)$$

where, $\delta$ is the bitline deviation, $C_c$ is the cell capacitance, $C_b$ is the bitline capacitance, and $k$ is the number of cells in the fully charged state. It is clear that $\delta > 0$ if and only if $2k - 3 > 0$. In other words, the bitline deviation is positive if $k = 2, 3$ and it is negative if $k = 0, 1$. Therefore, we expect the final state of the bitline to be $V_{DD}$ if at least two of the three cells are initially fully charged, and the final state to be 0, if at least two of the three cells are initially fully empty.

Figure 4 shows an example TRA where two of the three cells are initially in the charged state ❶. When the wordlines of all the three cells are raised simultaneously ❷, charge sharing results in a positive deviation on the bitline. Therefore, after sense amplification ❸, the sense amplifier drives the bitline to $V_{DD}$, and as a result, fully charges all the three cells.[2]

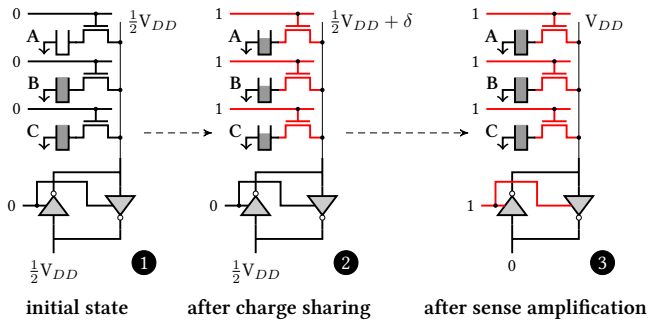


**Figure 4: Triple-row activation**

If $A$, $B$, and $C$ represent the logical values of the three cells, then the final state of the bitline is $AB + BC + CA$ (the *bitwise majority* function). Importantly, we can rewrite this expression as $C(A + B) + \overline{C}(AB)$. In other words, by controlling the value of the cell $C$, we can use TRA to execute a *bitwise AND* or *bitwise OR* of the cells $A$ and $B$. Since activation is a row-level operation in DRAM, TRA operates on an *entire row* of DRAM cells and sense amplifiers, thereby enabling a multi-kilobyte-wide bitwise AND/OR of two rows.

[2] Modern DRAMs use an open-bitline architecture [29, 57, 71, 79], where cells are also connected to bitline. The three cells in our example are connected to the bitline. However, based on the duality principle of Boolean algebra [104], i.e., `not` (A `and` B) ≡ (`not` A) `or` (`not` B), TRA works seamlessly even if all the three cells are connected to bitline.

### 3.2. Making TRA Work

There are five potential issues with TRA that we need to resolve for it to be implementable in a real DRAM design.

1. When simultaneously activating three cells, the deviation on the bitline may be smaller than when activating only one cell. This may lengthen sense amplification or worse, the sense amplifier may detect the wrong value.
2. Equation 1 assumes that all cells have the same capacitance, and that the transistors and bitlines behave ideally. However, due to process variation, these assumptions are not true in real designs [30, 67, 71]. This can affect the reliability of TRA, and thus the correctness of its results.
3. As shown in Figure 4 (state ❸), TRA overwrites the data of all the three cells with the final result value. In other words, TRA overwrites all source cells, thereby destroying their original values.
4. Equation 1 assumes that the cells involved in a TRA are either fully-charged or fully-empty. However, DRAM cells leak charge over time [78]. If the cells involved have leaked significantly, TRA may not operate as expected.
5. Simultaneously activating three *arbitrary* rows inside a DRAM subarray requires the memory controller and the row decoder to simultaneously communicate and decode three row addresses. This introduces a large cost on the address bus and the row decoder, potentially tripling these structures, if implemented naïvely.

We address the first two issues by performing rigorous circuit simulations of TRA. Our results confirm that TRA works as expected (Section 6). In Sections 3.3 and 3.4, we propose a simple implementation of Ambit-AND-OR that addresses *all* of the last three issues at low cost.

### 3.3. Implementation of Ambit-AND-OR

To solve issues 3, 4, and 5 described in Section 3.2, our implementation of Ambit reserves a set of *designated rows* in each subarray that are used to perform TRAs. These designated rows are chosen statically at *design time*. To perform a bulk bitwise AND or OR operation on two arbitrary source rows, our mechanism *first copies* the data of the source rows into the designated rows and performs the required TRA on the designated rows. As an example, to perform a bitwise AND/OR of two rows `A` and `B`, and store the result in row `R`, our mechanism performs the following steps.

1. *Copy* data of row `A` to designated row `T0`
2. *Copy* data of row `B` to designated row `T1`
3. *Initialize* designated row `T2` to 0
4. *Activate* designated rows `T0`, `T1`, and `T2` simultaneously
5. *Copy* data of row `T0` to row `R`

Let us understand how this implementation addresses the last three issues described in Section 3.2. First, by performing the TRA on the designated rows, and *not* directly on the source data, our mechanism avoids overwriting the source data (issue 3). Second, each copy operation refreshes the cells of the destination row by accessing the row [78]. Also, each copy operation takes five-six orders of magnitude lower latency (100 ns—1 $\mu s$) than the refresh interval (64 ms). Since these copy operations (Steps 1 and 2 above) are performed *just before* the TRA, the rows involved in the TRA are very close to the fully-refreshed state just before the TRA operation (issue 4). Finally, since the *designated* rows are chosen statically at design time, the Ambit controller uses a reserved address to communicate the TRA of a *pre-defined* set of three designated rows. To this end, Ambit reserves a set of row addresses *just* to trigger TRAs. For instance, in our implementation to perform a TRA of designated rows `T0`, `T1`, and `T2` (Step 4, above), the Ambit controller simply issues an `ACTIVATE` with the reserved address `B12` (see Section 5.1 for a full list of reserved addresses). The row decoder maps `B12` to *all* the three wordlines of the designated rows `T0`, `T1`, and `T2`. This mechanism requires *no* changes to the address bus and significantly reduces the cost and complexity of the row decoder compared to performing TRA on three *arbitrary* rows (issue 5).

### 3.4. Fast Row Copy and Initialization Using RowClone

Our mechanism needs three row copy operations and one row initialization operation. These operations, if performed naïvely, can nullify the benefits of Ambit, as a row copy or row initialization performed using the memory controller incurs high latency [29, 97]. Fortunately, a recent work, RowClone [97], proposes two techniques to efficiently copy data between rows *directly within* DRAM. The first technique, RowClone-FPM (Fast Parallel Mode), copies data within a subarray by issuing two back-to-back `ACTIVATE`s to the source row and the destination row. This operation takes only 80 ns [97]. The second technique, RowClone-PSM (Pipelined Serial Mode), copies data between two banks by using the internal DRAM bus. Although RowClone-PSM is faster and more efficient than copying data using the memory controller, it is significantly slower than RowClone-FPM.

Ambit relies on using RowClone-FPM for most of the copy operations.[3] To enable this, we propose three ideas. First, to allow Ambit to perform the initialization operation using RowClone-FPM, we reserve two *control* rows in each subarray, `C0` and `C1`. `C0` is initialized to all zeros and `C1` is initialized to all ones. Depending on the operation to be performed, bitwise AND or OR, Ambit copies the data from `C0` or `C1` to the appropriate designated row using RowClone-FPM. Second, we reserve separate designated rows in *each* subarray. This allows each subarray to perform bulk bitwise AND/OR operations on the rows that belong to that subarray by using RowClone-FPM for all the required copy operations. Third, to ensure that bulk bitwise operations are predominantly performed between rows inside the *same* subarray, we rely on 1) an accelerator API that allows applications to spec-

[3] A recent work, Low-cost Interlinked Subarrays (LISA) [29], proposes a mechanism to efficiently copy data across subarrays in the same bank. LISA uses a row of isolation transistors next to the sense amplifier to control data transfer across two subarrays. LISA can potentially benefit Ambit by improving the performance of bulk copy operations. However, as we will describe in Section 4, Ambit-NOT also adds transistors near the sense amplifier, posing some challenges in integrating LISA and Ambit. Therefore, we leave the exploration of using LISA to speedup Ambit as part of future work.

ify bitvectors that are likely to be involved in bitwise operations, and 2) a driver that maps such bitvectors to the *same* subarray (described in Section 5.4.2). With these changes, Ambit can use RowClone-FPM for a significant majority of the bulk copy operations, thereby ensuring high performance for the bulk bitwise operations.

## 4. Ambit-NOT

Ambit-NOT exploits the fact that at the end of the sense amplification process, the voltage level of the $\overline{\text{bitline}}$ represents the negated logical value of the cell. Our key idea to perform bulk bitwise NOT in DRAM is to transfer the data on the $\overline{\text{bitline}}$ to a cell that can *also* be connected to the bitline. For this purpose, we introduce the *dual-contact cell* (shown in Figure 5). A dual-contact cell (DCC) is a DRAM cell with two transistors (a 2T-1C cell similar to the one described in [53, 81]). Figure 5 shows a DCC connected to a sense amplifier. In a DCC, one transistor connects the cell capacitor to the bitline and the other transistor connects the cell capacitor to the $\overline{\text{bitline}}$. We refer to the wordline that controls the capacitor-bitline connection as the *d-wordline* (or data wordline) and the wordline that controls the capacitor-$\overline{\text{bitline}}$ connection as the *n-wordline* (or negation wordline). The layout of the dual-contact cell is similar to Lu et al.'s migration cell [81].

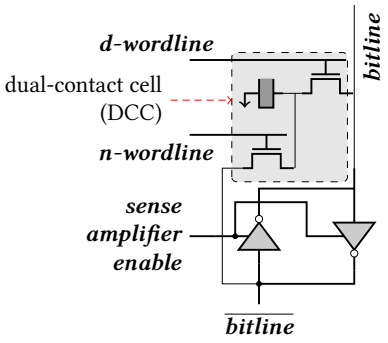


**Figure 5: A dual-contact cell connected to a sense amplifier**

Figure 6 shows the steps involved in transferring the negated value of a *source* cell on to the DCC connected to the same bitline (i.e., sense amplifier) ❶. Our mechanism first activates the source cell ❷. The activation drives the bitline to the data value corresponding to the source cell, $V_{DD}$ in this case and the $\overline{\text{bitline}}$ to the negated value, i.e., 0 ❸. In this *activated* state, our mechanism activates the *n-wordline*. Doing so enables the transistor that connects the DCC to the $\overline{\text{bitline}}$ ❹. Since the $\overline{\text{bitline}}$ is already at a stable voltage level of 0, it overwrites the value in the DCC capacitor with 0, thus copying the negated value of the source cell into the DCC. After this, our mechanism *precharges* the bank, and then copies the negated value from the DCC to the destination cell using RowClone.

**Implementation of Ambit-NOT.** Based on Lu et al.'s [81] layout, the cost of each row of DCC is the same as two regular DRAM rows. Similar to the designated rows used for Ambit-AND-OR (Section 3.3), the Ambit controller uses reserved row addresses to control the *d-wordline*s and *n-wordline*s of the DCC rows—e.g., in our implementation, address B5 maps to the *n-wordline* of the DCC row (Section 5.1). To perform a bitwise NOT of row A and store the result in row R, the Ambit controller performs the following steps.

1. *Activate* row A
2. *Activate n-wordline* of DCC (address B5)
3. *Precharge* the bank.
4. *Copy* data from *d-wordline* of DCC to row R (RowClone)

## 5. Ambit: Putting It All Together

In this section, we describe our implementation of Ambit by integrating Ambit-AND-OR and Ambit-NOT. First, both Ambit-AND-OR and Ambit-NOT reserve a set of rows in each subarray and a set of addresses that map to these rows. We present the full set of reserved addresses and their mapping in detail (Section 5.1). Second, we introduce a new primitive called AAP (ACTIVATE-ACTIVATE-PRECHARGE) that the Ambit controller uses to execute various bulk bitwise operations (Section 5.2). Third, we describe an optimization that lowers the latency of the AAP primitive, further improving the performance of Ambit (Section 5.3). Fourth, we describe how we integrate Ambit with the system stack (Section 5.4). Finally, we evaluate the hardware cost of Ambit (Section 5.5).

### 5.1. Row Address Grouping

Our implementation divides the space of row addresses in each subarray into three distinct groups (Figure 7): 1) **B**itwise group, 2) **C**ontrol group, and 3) **D**ata group.

The *B-group* (or the *bitwise* group) corresponds to the designated rows used to perform bulk bitwise AND/OR opera-

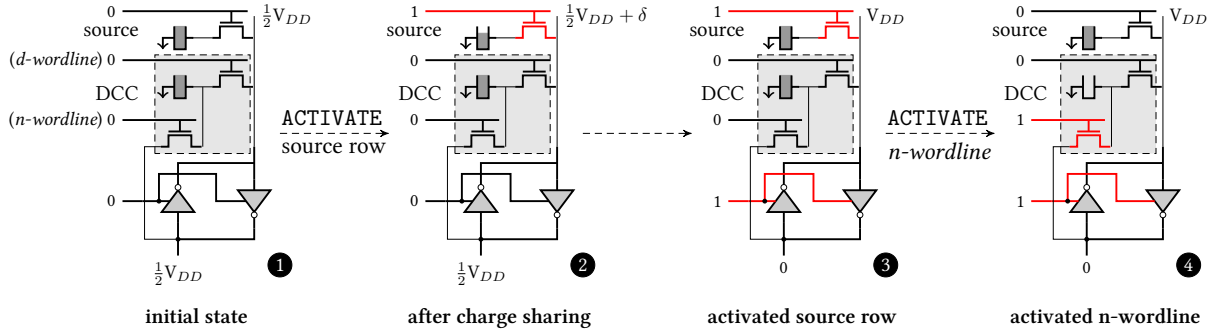


**Figure 6: Bitwise NOT using a dual-contact cell**

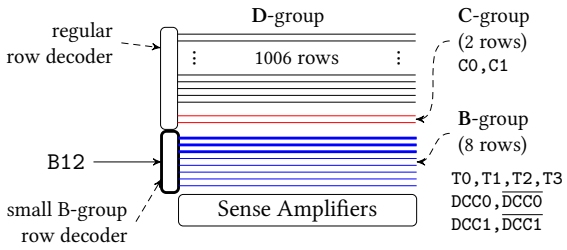


**Figure 7: Row address grouping in a subarray. The figure shows how the B-group row decoder (Section 5.3) simultaneously activates rows T0, T1, and T2 with a single address B12.**

tions (Section 3.3) and the dual-contact rows used to perform bulk bitwise NOT operations (Section 4). Minimally, Ambit requires three designated rows (to perform triple row activations) and one row of dual-contact cells in each subarray. However, to reduce the number of copy operations required by certain bitwise operations (like `xor` and `xnor`), we design each subarray with four designated rows, namely T0—T3, and two rows of dual-contact cells, one on each side of the row of sense amplifiers.[4] We refer to the *d-wordline*s of the two DCC rows as DCC0 and DCC1, and the corresponding *n-wordline*s as $\overline{\text{DCC0}}$ and $\overline{\text{DCC1}}$. The B-group contains 16 reserved addresses: B0—B15. Table 1 lists the mapping between the 16 addresses and the wordlines. The first eight addresses individually activate each of the 8 wordlines in the group. Addresses B12—B15 activate three wordlines simultaneously. Ambit uses these addresses to trigger triple-row activations. Finally, addresses B8—B11 activate two wordlines. Ambit uses these addresses to copy the result of an operation simultaneously to two rows. This is useful for `xor`/`xnor` operations to *simultaneously* negate a row of source data and also copy the source row to a designated row. Note that this is just an example implementation of Ambit and a real implementation may use more designated rows in the B-group, thereby enabling more complex bulk bitwise operations with fewer copy operations.

| Addr. | Wordline(s) | Addr. | Wordline(s) |
|---|---|---|---|
| B0 | T0 | B8 | $\overline{\text{DCC0}}$, T0 |
| B1 | T1 | B9 | $\overline{\text{DCC1}}$, T1 |
| B2 | T2 | B10 | T2, T3 |
| B3 | T3 | B11 | T0, T3 |
| B4 | DCC0 | B12 | T0, T1, T2 |
| B5 | $\overline{\text{DCC0}}$ | B13 | T1, T2, T3 |
| B6 | DCC1 | B14 | DCC0, T1, T2 |
| B7 | $\overline{\text{DCC1}}$ | B15 | DCC1, T0, T3 |

**Table 1: Mapping of B-group addresses to corresponding activated wordlines**

The *C-group* (or the *control* group) contains the two pre-initialized rows for controlling the bitwise AND/OR operations (Section 3.4). Specifically, this group contains two addresses: C0 (row with all zeros) and C1 (row with all ones).

[4]Each `xor`/`xnor` operation involves multiple `and`, `or`, and `not` operations. We use the additional designated row and the DCC row to store *intermediate* results computed as part of the `xor`/`xnor` operation (see Figure 8c).

The *D-group* (or the *data* group) corresponds to the rows that store regular data. This group contains all the addresses that are neither in the *B-group* nor in the *C-group*. Specifically, if each subarray contains 1024 rows, then the *D-group* contains 1006 addresses, labeled D0—D1005. Ambit exposes only the D-group addresses to the software stack. To ensure that the software stack has a contiguous view of memory, the Ambit controller interleaves the row addresses such that the D-group addresses across all subarrays are mapped contiguously to the processor's physical address space.

With these groups, the Ambit controller can use the *existing* DRAM interface to communicate all variants of ACTIVATE to the Ambit chip *without* requiring new commands. Depending on the address group, the Ambit DRAM chip internally processes each ACTIVATE appropriately. For instance, by just issuing an ACTIVATE to address B12, the Ambit controller triggers a triple-row activation of T0, T1, and T2. We now describe how the Ambit controller uses this row address mapping to perform bulk bitwise operations.

## 5.2. Executing Bitwise Ops: The AAP Primitive

Let us consider the operation, `Dk = not Di`. To perform this bitwise-NOT operation, the Ambit controller sends the following sequence of commands.

1. ACTIVATE Di; 2. ACTIVATE B5; 3. PRECHARGE;
4. ACTIVATE B4; 5. ACTIVATE Dk; 6. PRECHARGE;

The first three steps are the same as those described in Section 4. These three operations copy the negated value of row Di into the DCC0 row (as described in Figure 6). Step 4 activates DCC0, the *d-wordline* of the first DCC row, transferring the negated source data onto the bitlines. Step 5 activates the destination row, copying the data on the bitlines, i.e., the negated source data, to the destination row. Step 6 prepares the array for the next access by issuing a PRECHARGE.

The bitwise-NOT operation consists of two steps of ACTIVATE-ACTIVATE-PRECHARGE operations. We refer to this sequence as the AAP primitive. Each AAP takes two addresses as input. `AAP (addr1, addr2)` corresponds to the following sequence of commands:

```
ACTIVATE addr1; ACTIVATE addr2; PRECHARGE;
```

Logically, an AAP operation copies the result of the row activation of the first address (addr1) to the row(s) mapped to the second address (addr2).

Most bulk bitwise operations mainly involve a sequence of AAP operations. In a few cases, they require a regular ACTIVATE followed by a PRECHARGE, which we refer to as AP. AP takes one address as input. `AP (addr)` maps to the following two commands:

```
ACTIVATE addr; PRECHARGE;
```

Figure 8 shows the steps taken by the Ambit controller to execute three bulk bitwise operations: `and`, `nand`, and `xor`.

Let us consider the `and` operation, `Dk = Di and Dj`, shown in Figure 8a. The four AAP operations directly map to the steps described in Section 3.3. The first AAP copies the first source row (Di) into the designated row T0. Similarly, the second

a) Dk = Di **and** Dj

```
AAP (Di,  B0) ;T0 = Di
AAP (Dj,  B1) ;T1 = Dj
AAP (C0,  B2) ;T2 = 0
AAP (B12, Dk) ;Dk = T0 & T1
```

b) Dk = Di **nand** Dj

```
AAP (Di,  B0) ;T0 = Di
AAP (Dj,  B1) ;T1 = Dj
AAP (C0,  B2) ;T2 = 0
AAP (B12, B5) ;DCC0 = !(T0 & T1)
AAP (B4,  Dk) ;Dk = DCC0
```

c) Dk = Di **xor** Dj
= (Di & !Dj) | (!Di & Dj)

```
AAP (Di,  B8)  ;DCC0 = !Di, T0 = Di
AAP (Dj,  B9)  ;DCC1 = !Dj, T1 = Dj
AAP (C0,  B10) ;T2 = T3 = 0
 AP (B14)      ;T1 = DCC0 & T1
 AP (B15)      ;T0 = DCC1 & T0
AAP (C1,  B2)  ;T2 = 1
AAP (B12, Dk)  ;Dk = T0 | T1
```

or/nor/xnor can be implemented by appropriately modifying the control rows of and/nand/xor.

**Figure 8: Command sequences for different bitwise operations**

AAP copies the second source row Dj to row T1, and the third AAP copies the control row "0" to row T2 (to perform a bulk bitwise AND). Finally, the last AAP 1) issues an ACTIVATE to address B12, which simultaneously activates the rows T0, T1, and T2, resulting in an and operation of the rows T0 and T1, 2) issues an ACTIVATE to Dk, which copies the result of the and operation to the destination row Dk, and 3) precharges the bank to prepare it for the next access.

While each bulk bitwise operation involves multiple copy operations, this copy overhead can be reduced by applying standard compilation techniques. For instance, accumulation-like operations generate intermediate results that are immediately consumed. An optimization like dead-store elimination may prevent these values from being copied unnecessarily. Our evaluations (Section 8) take into account the overhead of the copy operations *without* such optimizations.

### 5.3. Accelerating AAP with a Split Row Decoder

The latency of executing any bulk bitwise operation using Ambit depends on the latency of the AAP primitive. The latency of the AAP in turn depends on the latency of ACTIVATE, i.e., $t_{RAS}$, and the latency of PRECHARGE, i.e., $t_{RP}$. The naïve approach to execute an AAP is to perform the three operations serially. Using this approach, the latency of AAP is $2t_{RAS} + t_{RP}$ (80 ns for DDR3-1600 [52]). While even this naïve approach offers better throughput and energy efficiency than existing systems (not shown here), we propose a simple optimization that significantly reduces the latency of AAP.

Our optimization is based on two observations. First, the second ACTIVATE of an AAP is issued to an already *activated bank*. As a result, this ACTIVATE does *not* require *full* sense amplification, which is the dominant portion of $t_{RAS}$ [45, 67, 71]. This enables the opportunity to reduce the latency for the second ACTIVATE of each AAP. Second, when we examine all the bitwise operations in Figure 8, with the exception of one AAP in nand, we find that *exactly one* of the two ACTIVATEs in each AAP is to a *B-group* address. This enables the opportunity to use a *separate* decoder for *B-group* addresses, thereby overlapping the latency of the two row activations in each AAP.

To exploit both of these observations, our mechanism splits the row decoder into two parts. The first part decodes all *C/D-group* addresses and the second smaller part decodes *only* B-group addresses. Such a split allows the subarray to *simultaneously* decode a *C/D-group* address along with a *B-group* address. When executing an AAP, the Ambit controller issues the second ACTIVATE of an AAP after the first activation has sufficiently progressed. This forces the sense amplifier to overwrite the data of the second row to the result of the first activation. This mechanism allows the Ambit controller to significantly overlap the latency of the two ACTIVATEs. This approach is similar to the inter-segment copy operation used by Tiered-Latency DRAM [68]. Based on SPICE simulations, our estimate of the latency of executing the back-to-back ACTIVATEs is only 4 ns larger than $t_{RAS}$. For DDR3-1600 (8-8-8) timing parameters [52], this optimization reduces the latency of AAP from 80 ns to 49 ns.

Since only addresses in the *B-group* are involved in triple-row activations, the complexity of simultaneously raising three wordlines is restricted to the small *B-group* decoder. As a result, the split row decoder also reduces the complexity of the changes Ambit introduces to the row decoding logic.

### 5.4. Integrating Ambit with the System

Ambit can be plugged in as an I/O (e.g., PCIe) device and interfaced with the CPU using a device model similar to other accelerators (e.g., GPU). While this approach is simple, as described in previous sections, the address and command interface of Ambit is *exactly the same* as that of commodity DRAM. This enables the opportunity to directly plug Ambit onto the system memory bus and control it using the memory controller. This approach has several benefits. First, applications can *directly* trigger Ambit operations using CPU instructions rather than going through a device API, which incurs additional overhead. Second, since the CPU can *directly* access Ambit memory, there is no need to copy data between the CPU memory and the accelerator memory. Third, existing cache coherence protocols can be used to keep Ambit memory and the on-chip cache coherent. To plug Ambit onto the system memory bus, we need additional support from the rest of the system stack, which we describe in this section.

**5.4.1. ISA Support.** To enable applications to communicate occurrences of bulk bitwise operations to the processor, we introduce new instructions of the form,

```
bbop dst, src1, [src2], size
```

where bbop is the bulk bitwise operation, dst is the destination address, src1 and src2 are the source addresses, and size denotes the length of operation in bytes. Note that size must be a multiple of DRAM row size. For bitvectors that are not a multiple of DRAM row size, we assume that the application will appropriately pad them with dummy data, or perform the residual (sub-row-sized) operations using the CPU.

**5.4.2. Ambit API/Driver Support.** For Ambit to provide significant performance benefit over existing systems, it is critical to ensure that most of the required copy operations are performed using RowClone-FPM, i.e., the source rows and the destination rows involved in bulk bitwise operations are present in the same DRAM subarray. To this end, we expect the manufacturer of Ambit to provide 1) an API that

enables applications to specify bitvectors that are likely to be involved in bitwise operations, and 2) a driver that is aware of the internal mapping of DRAM rows to subarrays and maps the bitvectors involved in bulk bitwise operations to the same DRAM subarray. Note that for a large bitvector, Ambit does *not* require the entire bitvector to fit inside a *single* subarray. Rather, each bitvector can be interleaved across multiple subarrays such that the corresponding portions of each bitvector are in the same subarray. Since each subarray contains over 1000 rows to store application data, an application can map hundreds of *large* bitvectors to Ambit, such that the copy operations required by *all* the bitwise operations across all these bitvectors can be performed efficiently using RowClone-FPM.

**5.4.3. Implementing the `bbop` Instructions.** Since all Ambit operations are row-wide, Ambit requires the source and destination rows to be row-aligned and the size of the operation to be a multiple of the size of a DRAM row. The microarchitecture implementation of a `bbop` instruction checks if each instance of the instruction satisfies this constraint. If so, the CPU sends the operation to the memory controller, which completes the operation using Ambit. Otherwise, the CPU executes the operation itself.

**5.4.4. Maintaining On-chip Cache Coherence.** Since both CPU and Ambit can access/modify data in memory, before performing any Ambit operation, the memory controller must 1) flush any dirty cache lines from the source rows, and 2) invalidate any cache lines from destination rows. Such a mechanism is already required by Direct Memory Access (DMA) [31], which is supported by most modern processors, and also by recently proposed mechanisms [48, 97]. As Ambit operations are always row-wide, we can use structures like the Dirty-Block Index [98] to speed up flushing dirty data. Our mechanism invalidates the cache lines of the destination rows in parallel with the Ambit operation.

**5.4.5. Error Correction and Data Scrambling.** In DRAM modules that support Error Correction Code (ECC), the memory controller must first read the data and ECC to verify data integrity. Since Ambit reads and modifies data directly in memory, it does not work with the existing ECC schemes (e.g., SECDED [43]). To support ECC with Ambit, we need an ECC scheme that is homomorphic [93] over all bitwise operations, i.e., ECC(A `and` B) = ECC(A) `and` ECC(B), and similarly for other bitwise operations. The only scheme that we are aware of that has this property is triple modular redundancy (TMR) [82], wherein ECC(A) = AA. The design of lower-overhead ECC schemes that are homomorphic over all bitwise operations is an open problem. For the same reason, Ambit does *not* work with data scrambling mechanisms that pseudo-randomly modify the data written to DRAM [36]. Note that these challenges are also present in any mechanism that *interprets* data directly in memory (e.g., the Automata Processor [33, 106]). We leave the evaluation of Ambit with TMR and exploration of other ECC and data scrambling schemes to future work.

### 5.5. Ambit Hardware Cost

As Ambit largely exploits the structure and operation of existing DRAM design, we estimate its hardware cost in terms of the overhead it imposes on top of today's DRAM chip and memory controller.

**5.5.1. Ambit Chip Cost.** In addition to support for RowClone, Ambit has only two changes on top of the existing DRAM chip design. First, it requires the row decoding logic to distinguish between the *B-group* addresses and the remaining addresses. Within the *B-group*, it must implement the mapping described in Table 1. As the *B-group* contains only 16 addresses, the complexity of the changes to the row decoding logic is low. The second source of cost is the implementation of the dual-contact cells (DCCs). In our design, each sense amplifier has only one DCC on each side, and each DCC has two wordlines associated with it. In terms of area, each DCC row costs roughly two DRAM rows, based on estimates from Lu et al. [81]. We estimate the overall storage cost of Ambit to be roughly 8 DRAM rows per subarray—for the four designated rows and the DCC rows (< 1% of DRAM chip area).

**5.5.2. Ambit Controller Cost.** On top of the existing memory controller, the Ambit controller must statically store 1) information about different address groups, 2) the timing of different variants of the `ACTIVATE`, and 3) the sequence of commands required to complete different bitwise operations. When Ambit is plugged onto the system memory bus, the controller can interleave the various AAP operations in the bitwise operations with other regular memory requests from different applications. For this purpose, the Ambit controller must also track the status of on-going bitwise operations. We expect the overhead of these additional pieces of information to be small compared to the benefits enabled by Ambit.

**5.5.3. Ambit Testing Cost.** Testing Ambit chips is similar to testing regular DRAM chips. In addition to the regular DRAM rows, the manufacturer must test if the TRA operations and the DCC rows work as expected. In each subarray with 1024 rows, these operations concern only 8 DRAM rows and 16 addresses of the B-group. In addition, all these operations are triggered using the `ACTIVATE` command. Therefore, we expect the overhead of testing an Ambit chip on top of testing a regular DRAM chip to be low.

When a component is found to be faulty during testing, DRAM manufacturers use a number of techniques to improve the overall yield; The most prominent among them is using spare rows to replace faulty DRAM rows. Similar to some prior works [67, 68, 97], Ambit requires faulty rows to be mapped to spare rows within the *same* subarray. Note that, since Ambit preserves the existing DRAM command interface, an Ambit chip that fails during testing can still be shipped as a regular DRAM chip. This significantly reduces the impact of Ambit-specific failures on overall DRAM yield.

## 6. Circuit-level SPICE Simulations

We use SPICE simulations to confirm that Ambit works reliably. Of the two components of Ambit, our SPICE results show that Ambit-NOT *always* works as expected and is *not* affected by process variation. This is because, Ambit-NOT operation is very similar to existing DRAM operation (Section 4). On the other hand, Ambit-AND-OR requires triple-row activation, which involves charge sharing between three cells on a bitline. As a result, it can be affected by process variation in various circuit components.

To study the effect of process variation on TRA, our SPICE simulations model variation in *all* the components in the subarray (cell capacitance, transistor length/width/resistance, bitline/wordline capacitance and resistance, and voltage levels). We implement the sense amplifier using 55nm DDR3 model parameters [14], and PTM low-power transistor models [9, 117]. We use cell/transistor parameters from the Rambus power model [14] (cell capacitance = 22fF; transistor width/height = 55nm/85nm).[5]

We first identify the worst case for TRA, wherein every component has process variation that works toward making TRA fail. Our results show that even in this extremely adversarial scenario, TRA works reliably for up to ±6% variation in each component.

In practice, variations across components are not so highly correlated. Therefore, we use Monte-Carlo simulations to understand the practical impact of process variation on TRA. We increase the amount of process variation from ±5% to ±25% and run 100,000 simulations for each level of process variation. Table 2 shows the percentage of iterations in which TRA operates incorrectly for each level of variation.

| Variation | ±0% | ±5% | ±10% | ±15% | ±20% | ±25% |
|---|---|---|---|---|---|---|
| % Failures | 0.00% | 0.00% | 0.29% | 6.01% | 16.36% | 26.19% |

**Table 2: Effect of process variation on TRA**

Two conclusions are in order. First, as expected, up to ±5% variation, there are zero errors in TRA. Second, even with ±10% and ±15% variation, the percentage of erroneous TRAs across 100,000 iterations each is just 0.29% and 6.01%. These results show that Ambit is reliable even in the presence of significant process variation.

The effect of process variation is expected to get worse with smaller technology nodes [55]. However, as Ambit largely uses the existing DRAM structure and operation, many techniques used to combat process variation in existing chips can be used for Ambit as well (e.g., spare rows or columns). In addition, as described in Section 5.5.3, Ambit chips that fail testing only for TRA can potentially be shipped as regular DRAM chips, thereby alleviating the impact of TRA failures on overall DRAM yield, and thus cost.

[5]In DRAM, temperature affects mainly cell leakage [30, 46, 67, 78, 79, 87, 92, 114]. As TRA is performed on cells that are almost fully-refreshed, we do not expect temperature to affect TRA.

## 7. Analysis of Throughput & Energy

We compare the raw throughput of bulk bitwise operations using Ambit to a multi-core Intel Skylake CPU [7], an NVIDIA GeForce GTX 745 GPU [4], and processing in the logic layer of an HMC 2.0 [6] device. The Intel CPU has 4 cores with Advanced Vector eXtensions [49], and two 64-bit DDR3-2133 channels. The GTX 745 contains 3 streaming multi-processors, each with 128 CUDA cores [77], and one 128-bit DDR3-1800 channel. The HMC 2.0 device consists of 32 vaults each with 10 GB/s bandwidth. We use two Ambit configurations: *Ambit* that integrates our mechanism into a regular DRAM module with 8 banks, and *Ambit-3D* that extends a 3D-stacked DRAM similar to HMC with support for Ambit. For each bitwise operation, we run a microbenchmark that performs the operation repeatedly for many iterations on large input vectors (32 MB), and measure the throughput of the operation. Figure 9 plots the results of this experiment for the five systems (the y-axis is in log scale).

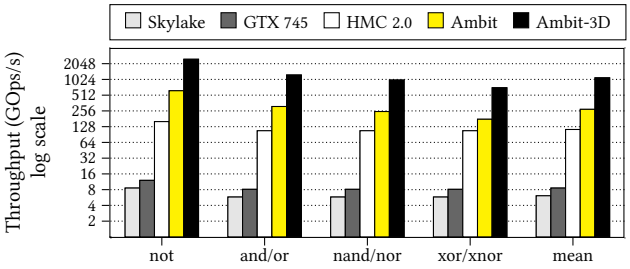


**Figure 9: Throughput of bulk bitwise operations.**

We draw three conclusions. First, the throughput of Skylake, GTX 745, and HMC 2.0 are limited by the memory bandwidth available to the respective processors. With an order of magnitude higher available memory bandwidth, HMC 2.0 achieves 18.5X and 13.1X better throughput for bulk bitwise operations compared to Skylake and GTX 745, respectively. Second, Ambit, with its ability to exploit the maximum internal DRAM bandwidth and memory-level parallelism, outperforms all three systems. On average, Ambit (with 8 DRAM banks) outperforms Skylake by 44.9X, GTX 745 by 32.0X, and HMC 2.0 by 2.4X. Third, 3D-stacked DRAM architectures like HMC contain a large number of banks (256 banks in 4GB HMC 2.0). By extending 3D-stacked DRAM with support for Ambit, Ambit-3D improves the throughput of bulk bitwise operations by 9.7X compared to HMC 2.0.

We estimate energy for DDR3-1333 using the Rambus power model [14]. Our energy numbers include only the DRAM and channel energy, and not the energy consumed by the processor. For Ambit, some activations have to raise multiple wordlines and hence, consume higher energy. Based on our analysis, the activation energy increases by 22% for each additional wordline raised. Table 3 shows the energy consumed per kilo-byte for different bitwise operations. Across all bitwise operations, Ambit reduces energy consumption by 25.1X—59.5X compared to copying data with the memory controller using the DDR3 interface.

| | Design | not | and/or | nand/nor | xor/xnor |
|---|---|---|---|---|---|
| DRAM & Channel Energy (nJ/KB) | **DDR3** | 93.7 | 137.9 | 137.9 | 137.9 |
| | **Ambit** (↓) | 1.6 59.5X | 3.2 43.9X | 4.0 35.1X | 5.5 25.1X |

**Table 3: Energy of bitwise operations. (↓) indicates energy reduction of Ambit over the traditional DDR3-based design.**

## 8. Effect on Real-World Applications

We evaluate the benefits of Ambit on real-world applications using the Gem5 full-system simulator [22]. Table 4 lists the main simulation parameters. Our simulations take into account the cost of maintaining coherence, and the overhead of RowClone to perform copy operations. We assume that application data is mapped such that all bitwise operations happen across rows *within a subarray*. We quantitatively evaluate three applications: 1) a database bitmap index [3, 8, 10, 11], 2) BitWeaving [75], a mechanism to accelerate database column scan operations, and 3) a bitvector-based implementation of the widely-used *set* data structure. In Section 8.4, we discuss four other applications that can benefit from Ambit.

| | |
|---|---|
| Processor | x86, 8-wide, out-of-order, 4 Ghz 64-entry instruction queue |
| L1 cache | 32 KB D-cache, 32 KB I-cache, LRU policy |
| L2 cache | 2 MB, LRU policy, 64 B cache line size |
| Memory Controller | 8 KB row size, FR-FCFS [94, 118] scheduling |
| Main memory | DDR4-2400, 1-channel, 1-rank, 16 banks |

**Table 4: Major simulation parameters**

### 8.1. Bitmap Indices

Bitmap indices [26] are an alternative to traditional B-tree indices for databases. Compared to B-trees, bitmap indices 1) consume less space, and 2) can perform better for many queries (e.g., joins, scans). Several major databases support bitmap indices (e.g., Oracle [8], Redis [10], Fastbit [3], rlite [11]). Several real applications (e.g., [1, 2, 12, 32]) use bitmap indices for fast analytics. As bitmap indices heavily rely on bulk bitwise operations, Ambit can accelerate bitmap indices, thereby improving overall application performance.

To demonstrate this benefit, we use the following workload from a real application [32]. The application uses bitmap indices to track users' characteristics (e.g., gender) and activities (e.g., did the user log in to the website on day 'X'?) for $u$ users. Our workload runs the following query: "How many unique users were active every week for the past $w$ weeks? and How many male users were active each of the past $w$ weeks?" Executing this query requires $6w$ bulk bitwise `or`, $2w$-1 bulk bitwise `and`, and $w$+1 bulk bitcount operations. In our mechanism, the bitcount operations are performed by the CPU. Figure 10 shows the end-to-end query execution time of the baseline and Ambit for the above experiment for various values of $u$ and $w$.

We draw two conclusions. First, as each query has $O(w)$ bulk bitwise operations and each bulk bitwise operation takes $O(u)$ time, the query execution time increases with increasing value $uw$. Second, Ambit significantly reduces the query execution time compared to the baseline, by 6X on average.

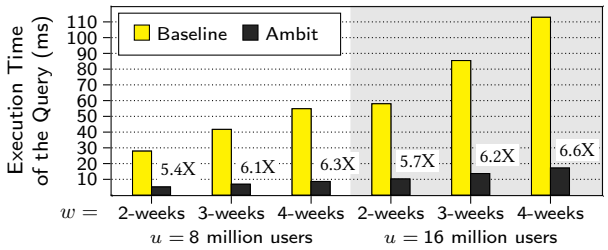


**Figure 10: Bitmap index performance. The value above each bar indicates the reduction in execution time due to Ambit.**

While we demonstrate the benefits of Ambit using one query, as all bitmap index queries involve several bulk bitwise operations, we expect Ambit to provide similar performance benefits for any application using bitmap indices.

### 8.2. BitWeaving: Fast Scans using Bitwise Operations

Column scan operations are a common part of many database queries. They are typically performed as part of evaluating a predicate. For a column with integer values, a predicate is typically of the form, `c1 <= val <= c2`, for two integer constants `c1` and `c2`. Recent works [75, 110] observe that existing data representations for storing columnar data are inefficient for such predicate evaluation especially when the number of bits used to store each value of the column is less than the processor word size (typically 32 or 64). This is because 1) the values do not align well with word boundaries, and 2) the processor typically does not have comparison instructions at granularities smaller than the word size. To address this problem, BitWeaving [75] proposes two column representations, called BitWeaving-H and BitWeaving-V. As BitWeaving-V is faster than BitWeaving-H, we focus our attention on BitWeaving-V, and refer to it as just BitWeaving.

BitWeaving stores the values of a column such that the first bit of all the values of the column are stored contiguously, the second bit of all the values of the column are stored contiguously, and so on. Using this representation, the predicate `c1 <= val <= c2`, can be represented as a series of bitwise operations starting from the most significant bit all the way to the least significant bit (we refer the reader to the BitWeaving paper [75] for the detailed algorithm). As these bitwise operations can be performed in parallel across multiple values of the column, BitWeaving uses the hardware SIMD support to accelerate these operations. With support for Ambit, these operations can be performed in parallel across a larger set of values compared to 128/256-bit SIMD available in existing CPUs, thereby enabling higher performance.

We show this benefit by comparing the performance of BitWeaving using a baseline CPU with support for 128-bit SIMD to the performance of BitWeaving accelerated by Ambit for the following commonly-used query on a table `T`:

'`select count(*) from T where c1 <= val <= c2`'

Evaluating the predicate involves a series of bulk bitwise operations and the `count(*)` requires a bitcount operation.

The execution time of the query depends on 1) the number of bits ($b$) used to represent each value `val`, and 2) the number of rows ($r$) in the table `T`. Figure 11 shows the speedup of Ambit over the baseline for various values of $b$ and $r$.

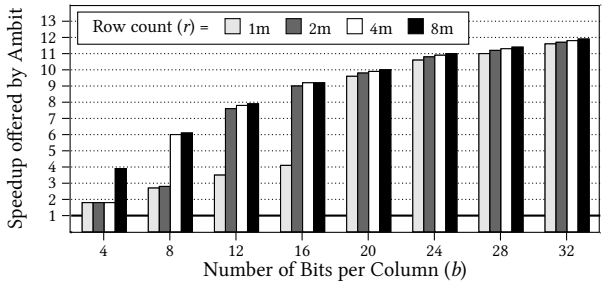


**Figure 11: Speedup offered by Ambit for BitWeaving over our baseline CPU with SIMD support**

We draw three conclusions. First, Ambit improves the performance of the query by between 1.8X and 11.8X (7.0X on average) compared to the baseline for various values of $b$ and $r$. Second, the performance improvement of Ambit increases with increasing number of bits per column ($b$), because, as $b$ increases, the fraction of time spent in performing the bitcount operation reduces. As a result, a larger fraction of the execution time can be accelerated using Ambit. Third, for $b$ = 4, 8, 12, and 16, we observe large jumps in the speedup of Ambit as we increase the row count. These large jumps occur at points where the working set stops fitting in the on-chip cache. By exploiting the high bank-level parallelism in DRAM, Ambit can outperform the baseline (by up to 4.1X) *even when* the working set fits in the cache.

### 8.3. Bitvectors vs. Red-Black Trees

Many algorithms heavily use the *set* data structure. Red-black trees [41] (RB-trees) are typically used to implement a set [13]. However, a set with a limited domain can be implemented using a bitvector—a set that contains only elements from 1 to $N$, can be represented using an $N$-bit bitvector (e.g., Bitset [13]). Each bit indicates whether the corresponding element is present in the set. Bitvectors provide constant-time *insert* and *lookup* operations compared to $O(\log n)$ time taken by RB-trees. However, set operations like *union*, *intersection*, and *difference* have to scan the *entire* bitvector regardless of the number of elements actually present in the set. As a result, for these three operations, depending on the number of elements in the set, bitvectors may outperform or perform worse than RB-trees. With support for fast bulk bitwise operations, we show that Ambit significantly shifts the trade-off in favor of bitvectors for these three operations.

To demonstrate this, we compare the performance of set *union*, *intersection*, and *difference* using: RB-tree, bitvectors with 128-bit SIMD support (Bitset), and bitvectors with Ambit. We run a benchmark that performs each operation on $m$ input sets and stores the result in an output set. We restrict the domain of the elements to be from 1 to $N$. Therefore, each set can be represented using an $N$-bit bitvector. For each of the three operations, we run multiple experiments varying the number of elements ($e$) *actually* present in each input set. Figure 12 shows the execution time of RB-tree, Bitset, and Ambit normalized to RB-tree for the three operations for $m = 15$, and $N = 512k$.

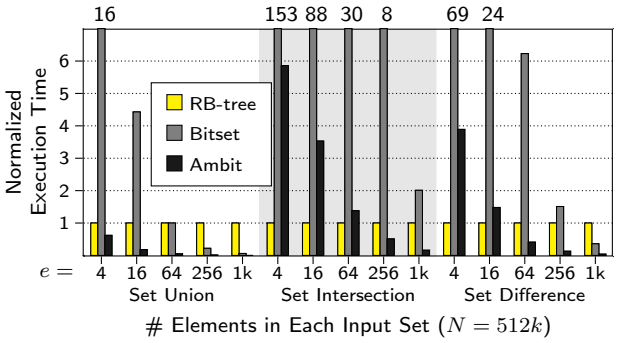


**Figure 12: Performance of set operations**

We draw three conclusions. First, by enabling much higher throughput for bulk bitwise operations, Ambit outperforms the baseline Bitset on all the experiments. Second, as expected, when the number of elements in each set is very small (16 out of $512k$), RB-Tree performs better than Bitset and Ambit (with the exception of *union*). Third, even when each set contains only 64 or more elements out of $512k$, Ambit significantly outperforms RB-Tree, 3X on average. We conclude that Ambit makes the bitvector-based implementation of a set more attractive than the commonly-used red-black-tree-based implementation.

### 8.4. Other Applications

**8.4.1. BitFunnel: Web Search.** Microsoft recently open-sourced BitFunnel [40], a technology that improves the efficiency of document filtering in web search. BitFunnel represents both documents and queries as a bag of words using Bloom filters [23], and uses bitwise AND operations on specific locations of the Bloom filters to efficiently identify documents that contain all the query words. With Ambit, this operation can be significantly accelerated by simultaneously performing the filtering for thousands of documents.

**8.4.2. Masked Initialization.** *Masked initializations* [90] are very useful in applications like graphics (e.g., for clearing a specific color in an image). By expressing such masked operations using bitwise AND/OR operations, we can easily accelerate such masked initializations using Ambit.

**8.4.3. Encryption.** Many encryption algorithms heavily use bitwise operations (e.g., XOR) [44, 83, 107]. The Ambit support for fast bulk bitwise operations can i) boost the performance of existing encryption algorithms, and ii) enable new encryption algorithms with high throughput and efficiency.

**8.4.4. DNA Sequence Mapping.** In DNA sequence mapping, prior works [20, 21, 58, 69, 73, 91, 95, 109, 112, 113] propose algorithms to map sequenced *reads* to the reference genome. Some works [20, 21, 58, 84, 91, 113] heavily use bulk bitwise operations. Ambit can significantly improve the performance of such DNA sequence mapping algorithms [58].

## 9. Related Work

To our knowledge, this is the first work that proposes an accelerator to perform bulk bitwise operations completely inside DRAM with high efficiency *and* low cost. We now compare Ambit to prior related works that use different techniques to accelerate bitwise operations.

Two patents [18, 19] from Mikamonu describe a DRAM design with 3T-1C cells and additional logic (e.g., muxes) to perform NAND/NOR operations on the data inside DRAM. While this architecture can perform bitwise operations inside DRAM, the 3T-1C cells result in *significant area overhead* to the DRAM array, and hence greatly reduce overall memory density/capacity. In contrast, Ambit builds on top of the existing DRAM architecture, with very few modifications, and therefore, incurs low cost.

Compute Cache [15] proposes a mechanism to perform bitwise operations inside on-chip SRAM caches. While Compute Cache improves performance and reduces energy consumption by reducing data movement between the cache and the CPU, the main drawback of Compute Cache is that it requires the data structures involved in bitwise operations to fit in the on-chip cache. However, this may not be the case for data-intensive applications, such as databases, graph analytics, and web search. For such applications, Ambit enables significant performance improvements as it can exploit the large capacity and parallelism present inside the DRAM chip. In general, since Compute Cache and Ambit benefit applications with different working set characteristics, they can be employed in conjunction to achieve greater benefits.

Pinatubo [74] proposes a mechanism to perform bulk bitwise operations inside Phase Change Memory (PCM) [66]. Similarly, recent works [63, 64, 65, 72] propose mechanisms to perform bitwise operations and other operations (3-bit full adder) completely inside a memristor array. As the underlying memory technology is different, the mechanisms proposed by these works is completely different from Ambit. Moreover, given that DRAM is faster than PCM or memristors, Ambit can offer higher throughput compared to these mechanisms.

Prior works [54, 97, 100, 102] exploit memory architectures to accelerate specific operations. RowClone [97] efficiently performs bulk copy and initialization inside DRAM. Kang et al. [54] propose a mechanism to exploit SRAM to accelerate "sum of absolute differences" computation. ISAAC [102] proposes a mechanism to accelerate dot-product computation using memristors. Gather-Scatter DRAM [100] accelerates strided access patterns via simple changes to a DRAM module. None of these mechanisms can perform bitwise operations.

Prior works (e.g., [34, 35, 38, 39, 56, 62, 86, 88, 103, 105]) propose designs that *integrate custom processing logic* into the DRAM chip to perform bandwidth-intensive operations. These approaches have two drawbacks. First, the extensive logic added to DRAM significantly increases the chip cost. Second, logic designed using DRAM process is generally slower than regular processors.

Many works (e.g., [16, 17, 24, 25, 37, 42, 47, 48, 80, 89, 115]) propose mechanisms to perform computation in the logic layer of 3D-stacked memory architectures. Even though they provide higher bandwidth compared to off-chip memory, 3D-stacked architectures are still bandwidth limited compared to the maximum internal bandwidth available inside a DRAM chip [70]. We have shown in Section 7 that, for bulk bitwise operations, Ambit outperforms processing in the logic layer of HMC 2.0 [6]. However, processing in the logic layer can still be used to synergistically perform other operations, while Ambit can perform bulk bitwise operations inside DRAM.

## 10. Conclusion

We introduce Ambit, a new accelerator that performs bulk bitwise operations within a DRAM chip by exploiting the analog operation of DRAM. Ambit consists of two components. The first component uses simultaneous activation of three DRAM rows to perform bulk bitwise AND/OR operations. The second component uses the inverters present in each sense amplifier to perform bulk bitwise NOT operations. With these two components, Ambit can perform *any* bulk bitwise operation efficiently in DRAM. Our evaluations show that, on average, Ambit enables 32X/35X improvement in the throughput/energy of bulk bitwise operations compared to a state-of-the-art system. This improvement directly translates to performance improvement in three data-intensive applications. Ambit is generally applicable to any memory device that uses DRAM (e.g, 3D-stacked DRAM, embedded DRAM). We believe that the Ambit's support for fast and efficient bulk bitwise operations can enable better design of other applications to take advantage of such operations, which would result in large improvements in performance and efficiency.

## Acknowledgments

We thank the reviewers of ISCA 2016/2017, MICRO 2016/2017, and HPCA 2017 for their valuable comments. We thank the members of the SAFARI group and PDL for their feedback. We acknowledge the generous support of our industrial partners, especially Google, Huawei, Intel, Microsoft, Nvidia, Samsung, Seagate, and VMWare. This work was supported in part by NSF, SRC, and the Intel Science and Technology Center for Cloud Computing. A preliminary version of this work was published in IEEE CAL [99], which introduced bulk bitwise AND/OR in DRAM, and in ADCOM [96], which introduced the idea of *processing using memory*. An earlier pre-print of this paper was posted on arxiv.org [101].

## References

[1] Belly Card Engineering. `https://tech.bellycard.com/`.
[2] bitmapist. `https://github.com/Doist/bitmapist`.
[3] FastBit: An Efficient Compressed Bitmap Index Technology. `https://sdm.lbl.gov/fastbit/`.
[4] GeForce GTX 745. `http://www.geforce.com/hardware/desktop-gpus/geforce-gtx-745-oem/specifications`.
[5] High Bandwidth Memory DRAM. `http://www.jedec.org/standards-documents/docs/jesd235`.

[6] Hybrid Memory Cube Specification 2.0. http://www.hybridmemorycube.org/files/SiteDownloads/HMC-30G-VSR_HMCC_Specification_Rev2.0_Public.pdf.

[7] 6th Generation Intel Core Processor Family Datasheet. http://www.intel.com/content/www/us/en/processors/core/desktop-6th-gen-core-family-datasheet-vol-1.html.

[8] Using Bitmap Indexes in Data Warehouses. https://docs.oracle.com/cd/B28359_01/server.111/b28313/indexes.htm.

[9] Predictive Technology Model. http://ptm.asu.edu/.

[10] Redis - bitmaps. http://redis.io/topics/data-types-intro.

[11] rlite. https://github.com/seppo0010/rlite.

[12] Spool. http://www.getspool.com/.

[13] std::set, std::bitset. http://en.cppreference.com/w/cpp/.

[14] DRAM Power Model. https://www.rambus.com/energy/, 2010.

[15] S. Aga, S. Jeloka, A. Subramaniyan, S. Narayanasamy, D. Blaauw, and R. Das. Compute Caches. In *HPCA*, 2017.

[16] J. Ahn, S. Hong, S. Yoo, O. Mutlu, and K. Choi. A Scalable Processing-in-memory Accelerator for Parallel Graph Processing. In *ISCA*, 2015.

[17] J. Ahn, S. Yoo, O. Mutlu, and K. Choi. PIM-enabled Instructions: A Low-overhead, Locality-aware Processing-in-memory Architecture. In *ISCA*, 2015.

[18] A. Akerib, O. Agam, E. Ehrman, and M. Meyassed. Using Storage Cells to Perform Computation. US Patent 8908465, 2014.

[19] A. Akerib and E. Ehrman. In-memory Computational Device. US Patent 9653166, 2015.

[20] M. Alser, H. Hassan, H. Xin, O. Ergin, O. Mutlu, and C. Alkan. GateKeeper: A New Hardware Architecture for Accelerating Pre-Alignment in DNA Short Read Mapping. *Bioinformatics*, 2017.

[21] G. Benson, Y. Hernandez, and J. Loving. A Bit-Parallel, General Integer-Scoring Sequence Alignment Algorithm. In *CPM*, 2013.

[22] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood. The Gem5 Simulator. *SIGARCH CAN*, 2011.

[23] B. H. Bloom. Space/time Trade-offs in Hash Coding with Allowable Errors. *ACM Communications*, 13, July 1970.

[24] A. Boroumand, S. Ghose, B. Lucia, K. Hsieh, K. Malladi, H. Zheng, and O. Mutlu. LazyPIM: An Efficient Cache Coherence Mechanism for Processing-in-Memory. *IEEE CAL*, 2017.

[25] A. Boroumand, S. Ghose, M. Patel, H. Hassan, B. Lucia, N. Hajinazar, K. Hsieh, K. T. Malladi, H. Zheng, and O. Mutlu. LazyPIM: Efficient Support for Cache Coherence in Processing-in-Memory Architectures. *arXiv preprint arXiv:1706.03162*, 2017.

[26] C.-Y. Chan and Y. E. Ioannidis. Bitmap Index Design and Evaluation. In *SIGMOD*, 1998.

[27] K. K. Chang, D. Lee, Z. Chisti, A. R. Alameldeen, C. Wilkerson, Y. Kim, and O. Mutlu. Improving DRAM Performance by Parallelizing Refreshes with Accesses. In *HPCA*, 2014.

[28] K. K. Chang, A. Kashyap, H. Hassan, S. Ghose, K. Hsieh, D. Lee, T. Li, G. Pekhimenko, S. Khan, and O. Mutlu. Understanding Latency Variation in Modern DRAM Chips: Experimental Characterization, Analysis, and Optimization. In *SIGMETRICS*, 2016.

[29] K. K. Chang, P. J. Nair, D. Lee, S. Ghose, M. K. Qureshi, and O. Mutlu. Low-cost Inter-linked Subarrays (LISA): Enabling Fast Inter-subarray Data Movement in DRAM. In *HPCA*, 2016.

[30] K. K. Chang, A. G. Yaălikçi, S. Ghose, A. Agrawal, N. Chatterjee, A. Kashyap, D. Lee, M. O'Connor, H. Hassan, and O. Mutlu. Understanding Reduced-voltage Operation in Modern DRAM Devices: Experimental Characterization, Analysis, and Mechanisms. *SIGMETRICS*, 2017.

[31] J. Corbet, A. Rubini, and G. Kroah-Hartman. *Linux Device Drivers*, page 445. O'Reilly Media, 2005.

[32] D. Denir, I. AbdelRahman, L. He, and Y. Gao. Audience Insights Query Engine. https://www.facebook.com/business/news/audience-insights.

[33] P. Dlugosch, D. Brown, P. Glendenning, M. Leventhal, and H. Noyes. An Efficient and Scalable Semiconductor Architecture for Parallel Automata Processing. *IEEE TPDS*, 2014.

[34] J. Draper, J. Chame, M. Hall, C. Steele, T. Barrett, J. LaCoss, J. Granacki, J. Shin, C. Chen, C. W. Kang, I. Kim, and G. Daglikoca. The Architecture of the DIVA Processing-in-memory Chip. In *ICS*, 2002.

[35] D. Elliott, M. Stumm, W. M. Snelgrove, C. Cojocaru, and R. McKenzie. Computational RAM: Implementing Processors in Memory. *IEEE DT*, 1999.

[36] C. F. Falconer, C. P. Mozak, and A. J. Normal. Suppressing Power Supply Noise Using Data Scrambling in Double Data Rate Memory Systems. US Patent 8503678, 2009.

[37] A. Farmahini-Farahani, J. H. Ahn, K. Morrow, and N. S. Kim. NDA: Near-DRAM Acceleration Architecture Leveraging Commodity DRAM Devices and Standard Memory Modules. In *HPCA*, 2015.

[38] B. B. Fraguela, J. Renau, P. Feautrier, D. Padua, and J. Torrellas. Programming the FlexRAM Parallel Intelligent Memory System. In *PPoPP*, 2003.

[39] M. Gokhale, B. Holmes, and K. Iobst. Processing in Memory: The Terasys Massively Parallel PIM Array. *Computer*, 1995.

[40] B. Goodwin, M. Hopcroft, D. Luu, A. Clemmer, M. Curmei, S. Elnikety, and Y. He. BitFunnel: Revisiting Signatures for Search. In *SIGIR*, 2017.

[41] L. J. Guibas and R. Sedgewick. A Dichromatic Framework for Balanced Trees. In *SFCS*, 1978.

[42] Q. Guo, N. Alachiotis, B. Akin, F. Sadi, G. Xu, T. M. Low, L. Pileggi, J. C. Hoe, and F. Franchetti. 3D-stacked Memory-side Acceleration: Accelerator and System Design. In *WoNDP*, 2013.

[43] R. W. Hamming. Error Detecting and Error Correcting Codes. *BSTJ*, 1950.

[44] J.-W. Han, C.-S. Park, D.-H. Ryu, and E.-S. Kim. Optical Image Encryption Based on XOR Operations. *SPIE OE*, 1999.

[45] H. Hassan, G. Pekhimenko, N. Vijaykumar, V. Seshadri, D. Lee, O. Ergin, and O. Mutlu. ChargeCache: Reducing DRAM Latency by Exploiting Row Access Locality. In *HPCA*, 2016.

[46] H. Hassan, N. Vijaykumar, S. Khan, S. Ghose, K. Chang, G. Pekhimenko, D. Lee, O. Ergin, and O. Mutlu. SoftMC: A Flexible and Practical Open-source Infrastructure for Enabling Experimental DRAM Studies. In *HPCA*, 2017.

[47] K. Hsieh, E. Ebrahimi, G. Kim, N. Chatterjee, M. O'Connor, N. Vijaykumar, O. Mutlu, and S. W. Keckler. Transparent Offloading and Mapping (TOM): Enabling Programmer-transparent Near-data Processing in GPU Systems. In *ISCA*, 2016.

[48] K. Hsieh, S. Khan, N. Vijaykumar, K. K. Chang, A. Boroumand, S. Ghose, and O. Mutlu. Accelerating Pointer Chasing in 3D-stacked Memory: Challenges, Mechanisms, Evaluation. In *ICCD*, 2016.

[49] Intel. Intel Instruction Set Architecture Extensions. https://software.intel.com/en-us/intel-isa-extensions.

[50] K. Itoh. *VLSI Memory Chip Design*, volume 5. Springer Science & Business Media, 2013.

[51] J. Jeddeloh and B. Keeth. Hybrid Memory Cube: New DRAM Architecture Increases Density and Performance. In *VLSIT*, 2012.

[52] JEDEC. DDR3 SDRAM Standard, JESD79-3D. http://www.jedec.org/sites/default/files/docs/JESD79-3D.pdf, 2009.

[53] H. Kang and S. Hong. One-Transistor Type DRAM. US Patent 7701751, 2009.

[54] M. Kang, M.-S. Keel, N. R. Shanbhag, S. Eilert, and K. Curewitz. An Energy-efficient VLSI Architecture for Pattern Recognition via Deep Embedding of Computation in SRAM. In *ICASSP*, 2014.

[55] U. Kang, H.-s. Yu, C. Park, H. Zheng, J. Halbert, K. Bains, S. Jang, and J. S. Choi. Co-architecting Controllers and DRAM to Enhance DRAM Process Scaling. In *The Memory Forum*, 2014.

[56] Y. Kang, W. Huang, S.-M. Yoo, D. Keen, Z. Ge, V. Lam, P. Pattnaik, and J. Torrellas. FlexRAM: Toward an Advanced Intelligent Memory System. In *ICCD*, 1999.

[57] B. Keeth, R. J. Baker, B. Johnson, and F. Lin. *DRAM Circuit Design: Fundamental and High-Speed Topics*. Wiley-IEEE Press, 2007.

[58] J. S. Kim, D. Senol, H. Xin, D. Lee, S. Ghose, M. Alser, H. Hassan, O. Ergin, C. Alkan, and O. Mutlu. GRIM-filter: Fast Seed Filtering in Read Mapping Using Emerging Memory Technologies. *arXiv preprint arXiv:1708.04329*, 2017.

[59] Y. Kim, V. Seshadri, D. Lee, J. Liu, and O. Mutlu. A Case for Exploiting Subarray-level Parallelism (SALP) in DRAM. In *ISCA*, 2012.

[60] Y. Kim, W. Yang, and O. Mutlu. Ramulator: A Fast and Extensible DRAM Simulator. *IEEE CAL*, 2016.

[61] D. E. Knuth. The Art of Computer Programming. Fascicle 1: Bitwise Tricks & Techniques; Binary Decision Diagrams, 2009.

[62] P. M. Kogge. EXECUBE: A New Architecture for Scaleable MPPs. In *ICPP*, 1994.

[63] S. Kvatinsky, A. Kolodny, U. C. Weiser, and E. G. Friedman. Memristor-based IMPLY Logic Design Procedure. In *ICCD*, 2011.

[64] S. Kvatinsky, D. Belousov, S. Liman, G. Satat, N. Wald, E. G. Friedman, A. Kolodny, and U. C. Weiser. MAGIC —Memristor-Aided Logic. *IEEE TCAS II: Express Briefs*, 2014.

[65] S. Kvatinsky, G. Satat, N. Wald, E. G. Friedman, A. Kolodny, and U. C. Weiser. Memristor-Based Material Implication (IMPLY) Logic: Design Principles and Methodologies. *IEEE TVLSI*, 2014.

[66] B. C. Lee, E. Ipek, O. Mutlu, and D. Burger. Architecting Phase Change Memory As a Scalable DRAM Alternative. In *ISCA*, 2009.

[67] D. Lee, Y. Kim, G. Pekhimenko, S. Khan, V. Seshadri, K. K. Chang, and O. Mutlu. Adaptive-Latency DRAM: Optimizing DRAM Timing for the Common-case. In *HPCA*, 2015.

[68] D. Lee, Y. Kim, V. Seshadri, J. Liu, L. Subramanian, and O. Mutlu. Tiered-Latency DRAM: A Low Latency and Low Cost DRAM Architecture. In *HPCA*, 2013.

[69] D. Lee, F. Hormozdiari, H. Xin, F. Hach, O. Mutlu, and C. Alkan. Fast and Accurate Mapping of Complete Genomics Reads. *Methods*, 2015.

[70] D. Lee, S. Ghose, G. Pekhimenko, S. Khan, and O. Mutlu. Simultaneous Multi-Layer Access: Improving 3D-Stacked Memory Bandwidth at Low Cost. *ACM TACO*, 2016.

[71] D. Lee, S. Khan, L. Subramanian, S. Ghose, R. Ausavarungnirun, G. Pekhimenko, V. Seshadri, and O. Mutlu. Design-Induced Latency Variation in Modern DRAM Chips: Characterization, Analysis, and Latency Reduction Mechanisms. In *SIGMETRICS*, 2017.

[72] Y. Levy, J. Bruck, Y. Cassuto, E. G. Friedman, A. Kolodny, E. Yaakobi, and S. Kvatinsky. Logic Operations in Memory Using a Memristive Akers Array. *Microelectronics Journal*, 2014.

[73] H. Li and R. Durbin. Fast and Accurate Long-read Alignment with Burrows–Wheeler Transform. *Bioinformatics*, 2010.

[74] S. Li, C. Xu, Q. Zou, J. Zhao, Y. Lu, and Y. Xie. Pinatubo: A Processing-in-Memory Architecture for Bulk Bitwise Operations in Emerging Non-Volatile Memories. In *DAC*, 2016.

[75] Y. Li and J. M. Patel. BitWeaving: Fast Scans for Main Memory Data Processing. In *SIGMOD*, 2013.

[76] Y. Li and J. M. Patel. WideTable: An Accelerator for Analytical Data Processing. *Proc. VLDB Endow.*, 2014.

[77] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym. NVIDIA Tesla: A Unified Graphics and Computing Architecture. *IEEE Micro*, 2008.

[78] J. Liu, B. Jaiyen, R. Veras, and O. Mutlu. RAIDR: Retention-Aware Intelligent DRAM Refresh. In *ISCA*, 2012.

[79] J. Liu, B. Jaiyen, Y. Kim, C. Wilkerson, and O. Mutlu. An Experimental Study of Data Retention Behavior in Modern DRAM Devices: Implications for Retention Time Profiling Mechanisms. In *ISCA*, 2013.

[80] Z. Liu, I. Calciu, M. Herlihy, and O. Mutlu. Concurrent Data Structures for Near-Memory Computing. In *SPAA*, 2017.

[81] S.-L. Lu, Y.-C. Lin, and C.-L. Yang. Improving DRAM Latency with Dynamic Asymmetric Subarray. In *MICRO*, 2015.

[82] R. E. Lyons and W. Vanderkulk. The Use of Triple-Modular Redundancy to Improve Computer Reliability. *IBM JRD*, 1962.

[83] S. A. Manavski. CUDA Compatible GPU as an Efficient Hardware Accelerator for AES Cryptography. In *ICSPC*, 2007.

[84] G. Myers. A Fast Bit-vector Algorithm for Approximate String Matching Based on Dynamic Programming. *JACM*, 1999.

[85] E. O'Neil, P. O'Neil, and K. Wu. Bitmap Index Design Choices and Their Performance Implications. In *IDEAS*, 2007.

[86] M. Oskin, F. T. Chong, and T. Sherwood. Active Pages: A Computation Model for Intelligent Memory. In *ISCA*, 1998.

[87] M. Patel, J. S. Kim, and O. Mutlu. The Reach Profiler (REAPER): Enabling the Mitigation of DRAM Retention Failures via Profiling at Aggressive Conditions. In *ISCA*, 2017.

[88] D. Patterson, T. Anderson, N. Cardwell, R. Fromm, K. Keeton, C. Kozyrakis, R. Thomas, and K. Yelick. A Case for Intelligent RAM. *IEEE Micro*, 1997.

[89] A. Pattnaik, X. Tang, A. Jog, O. Kayiran, A. K. Mishra, M. T. Kandemir, O. Mutlu, and C. R. Das. SchEduling Techniques for GPU Architectures with Processing-in-memory Capabilities. In *PACT*, 2016.

[90] A. Peleg and U. Weiser. MMX Technology Extension to the Intel Architecture. *IEEE Micro*, 1996.

[91] K. R. Rasmussen, J. Stoye, and E. W. Myers. Efficient Q-gram Filters for Finding All $\varepsilon$-matches Over a Given Length. *JCB*, 2006.

[92] P. J. Restle, J. W. Park, and B. F. Lloyd. DRAM Variable Retention Time. In *IEDM*, 1992.

[93] R. L. Rivest, L. Adleman, and M. L. Dertouzos. On Data Banks and Privacy Homomorphisms. *FSC*, 1978.

[94] S. Rixner, W. J. Dally, U. J. Kapasi, P. Mattson, and J. D. Owens. Memory Access Scheduling. In *ISCA*, 2000.

[95] S. M. Rumble, P. Lacroute, A. V. Dalca, M. Fiume, A. Sidow, and M. Brudno. SHRiMP: Accurate Mapping of Short Color-space Reads. *PLOS Computational Biology*, 2009.

[96] V. Seshadri and O. Mutlu. *Simple Operations in Memory to Reduce Data Movement, ADCOM*, Chapter 5. Elsevier, 2017.

[97] V. Seshadri, Y. Kim, C. Fallin, D. Lee, R. Ausavarungnirun, G. Pekhimenko, Y. Luo, O. Mutlu, P. B. Gibbons, M. A. Kozuch, and T. C. Mowry. RowClone: Fast and Energy-efficient In-DRAM Bulk Data Copy and Initialization. In *MICRO*, 2013.

[98] V. Seshadri, A. Bhowmick, O. Mutlu, P. B. Gibbons, M. A. Kozuch, and T. C. Mowry. The Dirty-block Index. In *ISCA*, 2014.

[99] V. Seshadri, K. Hsieh, A. Boroumand, D. Lee, M. A. Kozuch, O. Mutlu, P. B. Gibbons, and T. C. Mowry. Fast Bulk Bitwise AND and OR in DRAM. *IEEE CAL*, 2015.

[100] V. Seshadri, T. Mullins, A. Boroumand, O. Mutlu, P. B. Gibbons, M. A. Kozuch, and T. C. Mowry. Gather-scatter DRAM: In-DRAM Address Translation to Improve the Spatial Locality of Non-unit Strided Accesses. In *MICRO*, 2015.

[101] V. Seshadri, D. Lee, T. Mullins, H. Hassan, A. Boroumand, J. Kim, M. A. Kozuch, O. Mutlu, P. B. Gibbons, and T. C. Mowry. Buddy-RAM: Improving the Performance and Efficiency of Bulk Bitwise Operations Using DRAM. *arXiv preprint arXiv:1611.09988*, 2016.

[102] A. Shafiee, A. Nag, N. Muralimanohar, R. Balasubramonian, J. P. Strachan, M. Hu, R. S. Williams, and V. Srikumar. ISAAC: A Convolutional Neural Network Accelerator with In-Situ Analog Arithmetic in Crossbars. In *ISCA*, 2016.

[103] D. E. Shaw, S. Stolfo, H. Ibrahim, B. K. Hillyer, J. Andrews, and G. Wiederhold. The NON-VON Database Machine: An Overview. `http://hdl.handle.net/10022/AC:P:11530`, 1981.

[104] R. Sikorski. *Boolean Algebras*, volume 2. Springer, 1969.

[105] H. S. Stone. A Logic-in-Memory Computer. *IEEE Trans. Comput.*, 1970.

[106] A. Subramaniyan and R. Das. Parallel Automata Processor. In *ISCA*, 2017.

[107] P. Tuyls, H. D. L. Hollmann, J. H. V. Lint, and L. Tolhuizen. XOR-based Visual Cryptography Schemes. *Designs, Codes and Cryptography*.

[108] H. S. Warren. *Hacker's Delight*. Addison-Wesley Professional, 2nd edition, 2012. ISBN 0321842685, 9780321842688.

[109] D. Weese, A.-K. Emde, T. Rausch, A. Döring, and K. Reinert. RazerS – fast Read Mapping with Sensitivity Control. *Genome research*, 2009.

[110] T. Willhalm, I. Oukid, I. Muller, and F. Faerber. Vectorizing Database Column Scans with Complex Predicates. In *ADMS*, 2013.

[111] K. Wu, E. J. Otoo, and A. Shoshani. Compressing Bitmap Indexes for Faster Search Operations. In *SSDBM*, 2002.

[112] H. Xin, D. Lee, F. Hormozdiari, S. Yedkar, O. Mutlu, and C. Alkan. Accelerating Read Mapping with FastHASH. *BMC Genomics*, 2013.

[113] H. Xin, J. Greth, J. Emmons, G. Pekhimenko, C. Kingsford, C. Alkan, and O. Mutlu. Shifted Hamming Distance: A Fast and Accurate SIMD-friendly Filter to Accelerate Alignment Verification in Read Mapping. *Bioinformatics*, 2015.

[114] D. S. Yaney, C. Y. Lu, R. A. Kohler, M. J. Kelly, and J. T. Nelson. A Meta-stable Leakage Phenomenon in DRAM Charge Storage - Variable Hold Time. In *IEDM*, 1987.

[115] D. Zhang, N. Jayasena, A. Lyashevsky, J. L. Greathouse, L. Xu, and M. Ignatowski. TOP-PIM: Throughput-oriented Programmable Processing in Memory. In *HPDC*, 2014.

[116] T. Zhang, K. Chen, C. Xu, G. Sun, T. Wang, and Y. Xie. Half-DRAM: A High-bandwidth and Low-power DRAM Architecture from the Rethinking of Fine-grained Activation. In *ISCA*, 2014.

[117] W. Zhao and Y. Cao. New Generation of Predictive Technology Model for Sub-45 nm Early Design Exploration. *IEEE TED*, 2006.

[118] W. K. Zuravleff and T. Robinson. Controller for a Synchronous DRAM that Maximizes Throughput by Allowing Memory Requests and Commands to be Issued Out of Order. US Patent 5630096, 1997.