ComputeDRAM: In-Memory Compute Using Off-the-Shelf DRAMs

Fei Gao
feig@princeton.edu
Department of Electrical Engineering
Princeton University

Georgios Tziantzioulis
georgios.tziantzioulis@princeton.edu
Department of Electrical Engineering
Princeton University

David Wentzlaff
wentzlaf@princeton.edu
Department of Electrical Engineering
Princeton University

ABSTRACT

In-memory computing has long been promised as a solution to the "Memory Wall" problem. Recent work has proposed using charge-sharing on the bit-lines of a memory in order to compute in-place and with massive parallelism, all without having to move data across the memory bus. Unfortunately, prior work has required modification to RAM designs (e.g. adding multiple row decoders) in order to open multiple rows simultaneously. So far, the competitive and low-margin nature of the DRAM industry has made commercial DRAM manufacturers resist adding any additional logic into DRAM. This paper addresses the need for in-memory computation with little to no change to DRAM designs. It is the first work to demonstrate in-memory computation with off-the-shelf, unmodified, commercial, DRAM. This is accomplished by violating the nominal timing specification and activating multiple rows in rapid succession, which happens to leave multiple rows open simultaneously, thereby enabling bit-line charge sharing. We use a constraint-violating command sequence to implement and demonstrate row copy, logical OR, and logical AND in unmodified, commodity, DRAM. Subsequently, we employ these primitives to develop an architecture for arbitrary, massively-parallel, computation. Utilizing a customized DRAM controller in an FPGA and commodity DRAM modules, we characterize this opportunity in hardware for all major DRAM vendors. This work stands as a proof of concept that in-memory computation is possible with unmodified DRAM modules and that there exists a financially feasible way for DRAM manufacturers to support in-memory compute.

CCS CONCEPTS

• **Computer systems organization → Parallel architectures**;
• **Hardware → Dynamic memory**.

KEYWORDS

DRAM, in-memory computing, bit-serial, main memory

ACM Reference Format:
Fei Gao, Georgios Tziantzioulis, and David Wentzlaff. 2019. ComputeDRAM: In-Memory Compute Using Off-the-Shelf DRAMs. In *The 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-52), October 12–16, 2019, Columbus, OH, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3352460.3358260

1 INTRODUCTION

In modern computing systems, moving data between compute resources and main memory utilizes a large portion of the overall system energy and significantly contributes to program execution time. As increasing numbers of processor cores have been integrated onto a single chip, the amount of memory bandwidth has not kept up, thereby leading to a "Memory Wall" [48, 62]. Making matters worse, the communication latency between compute resources and off-chip DRAM has not improved as fast as the amount of computing resources have increased.

To address these challenges, near-memory compute [3, 11, 29, 35, 55], Processors-in-Memory [19, 23, 27, 39, 52, 59], and in-memory compute [28, 33, 46] have all been proposed. This paper focuses on the most aggressive solution, performing computations **with** the memory. Unfortunately, performing computations with memory resources has relied on either emerging memory technologies [14, 16, 60] or has required additional circuits be added to RAM arrays. While some solutions have been demonstrated in silicon [1, 2, 45, 57], none of these solutions have gained widespread industry adoption largely due to requiring additional circuits to be added to already cost optimized and low-margin RAM implementations.

In this paper, **we demonstrate a novel method that performs computation with off-the-shelf, unmodified, commercial DRAM**. Utilizing a customized memory controller, we are able to change the timing of standard DRAM memory transactions, operating outside of specification, to perform massively parallel logical AND, logical OR, and row copy operations. Using these base operations and by storing and computing each value and its logical negation, we can compute arbitrary functions in a massively parallel bit-serial manner. Our novel operations function at the circuit level by forcing the commodity DRAM chip to open multiple rows simultaneously by activating them in rapid succession. Previous works [56, 57] have shown that with multiple rows opened, row copy and logical operations can be completed using bit-line charge sharing. However, to achieve this, all previous techniques relied on hardware modifications. To our knowledge, this is the first work to perform row copy, logical AND, and logical OR using off-the-shelf, unmodified, commercial DRAM.

With a slight modification to the DRAM controller, in-memory compute is able to save the considerable energy needed to move data across memory buses and cache hierarchies. We regard this work as an important proof of concept and indication that in-memory computation using DRAM is practical, low-cost, and should be included in all future computer systems. Taking a step further, we characterize the robustness of each computational operation and identify which operations can be performed by which vendor's DRAM. While our results show that not all vendors' off-the-shelf DRAMs support all operations, the virtue that we are able to find any unmodified DRAMs that do support a complete set of operations demonstrates that DRAM vendors can likely support in-memory compute in DRAM at little to no monetary cost. In-memory compute also shows potential as an alternative to using traditional silicon CMOS logic gates for compute. With the slowing and potential ending of Moore's Law, the computing community has to look beyond relying on non-scaling silicon CMOS logic transistors for alternatives such as relying more heavily on in-memory computation.

This work has the potential for large impact on the computing industry with minimal hardware design changes. Currently, DRAM is solely used to store data. Our discovery that off-the-shelf DRAM can be used to perform computations, as is, or with nominal modifications by DRAM vendors, means that with a small update to the DRAM controller, all new computers can perform massively parallel computations without needing to have data transit the memory bus and memory hierarchy. This massively parallel bit-serial computation has the potential to save significant energy and can complete computations in-parallel with the main processor. We envision this type of computing being used for data-parallel applications or sub-routines within a larger application. Prior work has even proposed using such architectures applied to image [53] and signal [10] processing, computer vision [20], and the emerging field of neural networks [21].

The main contributions of this work include:

- This is the first work to demonstrate row copy in off-the-shelf, unmodified, commercial, DRAM.
- This is the first work to demonstrate bit-wise logical AND and OR in off-the-shelf, unmodified, commercial, DRAM.
- We carefully characterize the capabilities and robustness of the in-memory compute operations across DDR3 DRAM modules from all major DRAM vendors.
- We characterize the effect that supply voltage and temperature have on the introduced operations' robustness.
- We present an algorithmic technique for performing arbitrary computations based on non-inverting operations. This is key as the DRAM bit-line charge sharing operations only support non-inverting operations.
- We construct a software framework to run massively parallel bit-serial computations and demonstrate it in a real system.

2 BACKGROUND

This section provides a brief introduction on DRAM operations, focusing on the attributes that affect our design; a detailed description of DRAM memory organization is discussed by Jacob et al. [32].

2.1 DRAM System Organization

DRAM memory follows a hierarchical system organization of: channels, ranks, banks, and rows/columns (Figure 1). An example organization could have one DIMM (dual in-line memory module) per channel, and up to two ranks per module. Each rank consists of eight physical chips, which are accessed in parallel and share the same command bus. When memory is accessed, all chips see the same address, and the data read out from different chips in one rank are concatenated. Each chip contains eight independent banks, and each bank is composed of rows and columns of DRAM memory cells. A row refers to a group of storage cells that are activated simultaneously by a row activation command, and a column refers to the smallest unit of data (one byte per chip) that can be addressed. Since it is impractical to use a single row decoder to address tens of thousands of rows in a bank, or to use long bit-lines to connect the cells in a bank vertically, the bank is further divided into sub-arrays, each of which contains 512 rows typically. Within a sub-array, each bit-line connects all the cells in one bit of a column and the sense amplifier at the local row buffer.

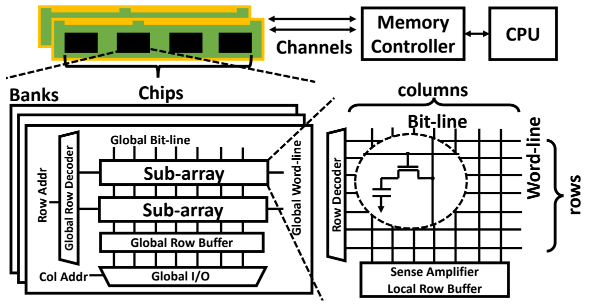

Figure 1: DRAM hierarchy

As the processor (or I/O device) operates, it generates a stream of memory accesses (reads/writes), which are forwarded to the Memory Controller (MC). The MC's role is to receive memory requests, generate a sequence of DRAM commands for the memory system, and issue these commands properly timed across the memory channel. Deviation from the DRAM timing specification directly affects the operation. The generated side-effects from non-nominal operation of a DRAM module can affect the stored data. It is these non-nominal side-effects that our work exploits for performing computation with memory.

2.2 DRAM commands and timing

In this section, we detail the attributes of the DRAM commands and timing that are important for our technique. Four commands are essential for DRAM:

① PRECHARGE: The PRECHARGE command applies to a whole bank. It first closes the currently opened row by zeroing all word-lines in the target bank, and subsequently drives all bit-lines to $V_{dd}/2$ as an initial value.

② ACTIVATE: The ACTIVATE command targets a specific row. Before an ACTIVATE is issued, a PRECHARGE command must be sent to the corresponding bank to ensure the initial voltage on the bit-line is $V_{dd}/2$. In nominal operations, at most one row can be activated in a bank at any time [37]. During ACTIVATE, the word-line of the addressed row is raised high, which connects the cells of that row directly to the bit-lines. Charge sharing then occurs between the storage cell and the bit-line. Although the capacitance of the cell is relatively small compared to the bit-line's, it could still make the

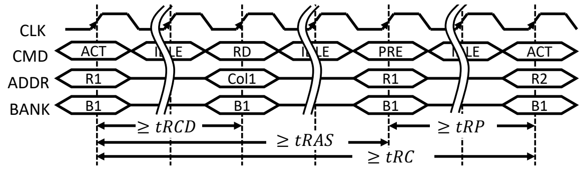

Figure 2: An example command sequence to read the DRAM from bank B_1, row R_1. If we want to read from another row R_2 in the same bank, we need to first precharge that bank and activate R_2.

bit-line voltage slightly higher or lower than $V_{dd}/2$, depending on the value stored in the cell. After the voltage on the bit-line is raised or lowered from $V_{dd}/2$, a sense amplifier is enabled which drags the voltage to V_{dd} or GND, thereby amplifying the value held in the bit-line. At the same time, the bit-cell is still connected to the bit-line, thus the value in the cell is preserved. Finally, the global bit-lines connect the output of the activated local row buffer to the global row buffer.

③ READ / ④ WRITE: The READ/WRITE commands apply to four or eight consecutive columns according to the burst mode. Based on the starting column address, the corresponding columns of data in the global row buffer are read out to or written from the I/O pins. Nominally, these commands must be sent after a row is activated in that bank, so that the target row is stored in the global row buffer.

As different commands take different amounts of time to complete, it is the responsibility of the memory controller to space out each command in order to not violate the nominal DRAM timing, thereby, ensuring reliable and correct operation. We present a standard DRAM read procedure in Figure 2 with marked time intervals t_{RCD}, t_{RAS}, t_{RP}, and t_{RC}, which are discussed below:

- Row Access to Column Access Delay (t_{RCD}): the minimum time between an ACTIVATE and a READ/WRITE.
- Row Access Strobe (t_{RAS}): the minimum time after the ACTIVATE that a PRECHARGE can be sent. This is time used to open a row, enable the sense amplifier, and wait for the voltage to reach V_{DD} or GND.
- Row Precharge (t_{RP}): the minimum time after the PRECHARGE in a bank before a new row access. It ensures that the previously activated row is closed and the bit-line voltage has reached $V_{dd}/2$.
- Row Cycle ($t_{RC} = t_{RAS} + t_{RP}$): the interval required between accesses to different rows in the same bank.

In this work, we explore how command sequences that violate t_{RP} and t_{RAS} can implement new functionality in DRAM.

3 COMPUTE IN DRAM

DRAM modules rely on MCs for correct operation, meaning that the adherence to DRAM timing constraints and the correct ordering of command sequences are enforced by the MC. This responsibility provides interesting opportunities to the MC designer to use the DRAM outside of specification [12, 36, 47]. Violation of timing constraints leads to non-nominal states in the DRAM which can potentially affect the stored data, either constructively or destructively. In this section, we detail how to transform a sequence

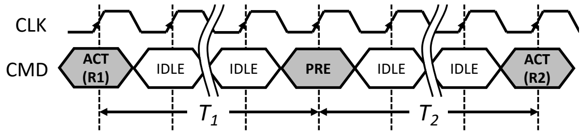

Figure 3: Command sequence for in-memory operations. We vary T_1 and T_2 to perform different operations.

of commands with nominal timing, into a sequence that violates timing constraints but performs in-memory computations.

Our contribution lies on identifying the command sequence and timing pairs that force the DRAM into a state where charge sharing occurs. The idea of utilizing charge sharing to perform operations has been previously discussed by Shibata et al. [58], Yang et al. [65], Craninckx et al. [15], and Monteiro [49]. Similarly, the idea of opening multiple rows to perform operations on the bit-line has been discussed by Akerib et al. [4], Aga et al. [1], Seshadri et al. [56, 57], and Li et al. [45]. **However, all previous work required hardware modifications to enable in-memory computing**. In contrast, in our work, by using specially-timed commands that violate timing specification, we managed to perform row copy and logical AND/OR operations without requiring any hardware modification in commodity, off-the-shelf DRAMs.

Our starting point is the command sequence in Figure 3. Three commands, ACTIVATE(R_1), PRECHARGE, and ACTIVATE(R_2), that target two different rows (R_1 and R_2) of the same bank are executed. The timing intervals between commands are labelled T_1 and T_2, and are controlled by the number of idle cycles in between. Under nominal operation, where T_1 is required to be longer than t_{RAS}, and T_2 longer than t_{RP}, this command sequence will open row R_1, close it, and then open row R_2, without any effect on the data stored in these rows. However, by appropriately reducing the timing intervals T_1 and T_2, outside the specification limits, we can force the chip to reach a non-nominal state that realizes a set of different basic operations.

3.1 Basic In-Memory Operations

By violating the timing constraints, we are able to perform three basic in-memory operations: row copy (a copy of data from one row to another), logical AND, and logical OR. To achieve the different basic operations with the same command sequence as seen in Figure 3, we manipulate the timing intervals and the initial data stored in memory.

3.1.1 Row Copy. Row copy is the simplest among the three operations and performs a copy from one row, R_1, to another, R_2. To perform row copy, we load the data of R_1 into the bit-line, and then overwrite R_2 using the sense amplifier. We achieve this by reducing the timing interval T_2, shown in Figure 3, to a value significantly shorter than t_{RP}. This causes the second ACTIVATE command to interrupt the PRECHARGE command. In regard to T_1, the only requirement for it is to be long enough for the sense amplifier to fully drive the bit-line with the data contained in row R_1. Note that the value of T_2 must not be too short as it could allow another row to be opened. This can be exploited to construct other operations (AND/OR) but is destructive for row copy.

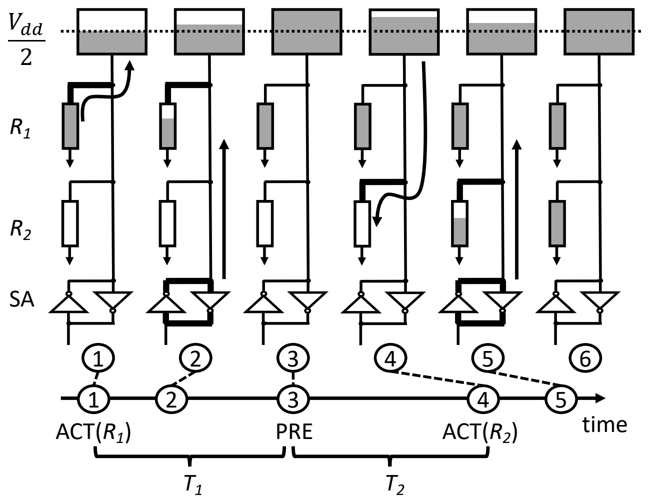

Figure 4: Timeline for a single bit of a column in a row copy operation. The data in R_1 is loaded to the bit-line, and overwrites R_2.

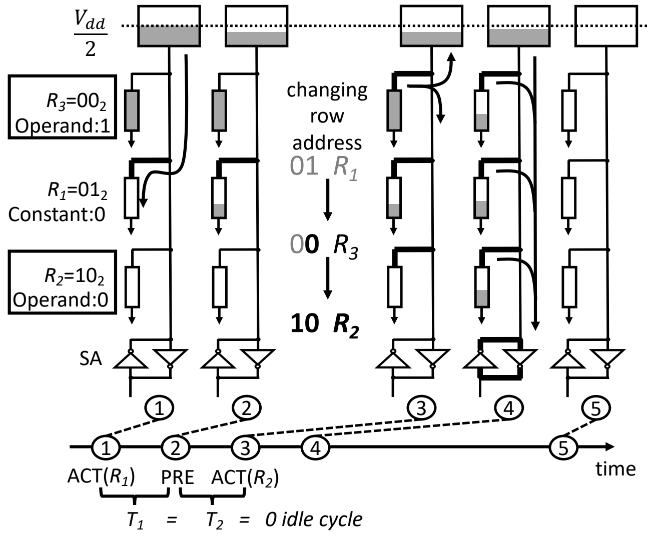

Figure 5: Logical AND in ComputeDRAM. R_1 is loaded with constant zero, and R_2 and R_3 store operands (0 and 1). The result ($0 = 1 \wedge 0$) is finally set in all three rows.

As the ACTIVATE and PRECHARGE commands generate the same effect across all columns of a row, we focus on a single bit of a column to provide insight into how row copy works. Figure 4 provides the timeline of a row copy operation for a single bit of a column. The rectangular boxes represent bit-line and cell capacitors. The boxes' width corresponds to the capacitance (C) and their height represents the voltage (V), thus the area of the shadowed portion corresponds to the charge ($Q = CV$). The voltage level of $V_{dd}/2$ is marked on the bit-line capacitor. Bold lines indicate that the word-line is open, or the sense amplifier is enabled.

Starting from left to right in Figure 4, we see each step of the operation. The initial state of the column is that of: the bit-line set to $V_{dd}/2$, the cell in R_1 charged (storing one), and the cell in R_2 discharged (storing zero). In step ①, we send the first ACTIVATE command, which opens the row R_1. Step ② shows the result of the charge sharing, and the sense amplifier starting to drive both the cell in R_1 and the bit-line to V_{dd}. This step lasts for a while to allow R_1 to recover the charge in its cell. In step ③, we execute the PRECHARGE command, which will attempt to close row R_1 and drive the bit-line to $V_{dd}/2$. The state of the column after a period of T_2 from executing the PRECHARGE command is shown in step ④. At this point, the word-line of R_1 has been zeroed out to close R_1, but the bit-line did not have enough time to be fully discharged to $V_{dd}/2$ yet, and the bit-line voltage level is way above the middle point. At the same time, the second ACTIVATE command is executed so as to interrupt the PRECHARGE process. The second ACTIVATE opens row R_2 and the charge starts to flow from the bit-line to the cells in R_2. As the capacitance of the bit-line is much larger than the cell's [56], a relatively significant amount of charge is stored in the bit-line, and the bit-line voltage can still be larger than $V_{dd}/2$ after the charge sharing, as shown in step ⑤. Thus, when the sense amplifier is enabled to drive the bit-line, together with the cell in R_2, it will reinforce their values to V_{dd}. This successfully completes the copy of data from row R_1 to row R_2 in step ⑥.

From the above procedure, two observations can be made. First, the row copy operation relies on charge sharing on the bit-line, which requires that the source and destination rows must share the same physical bit-line. Thus, due to the physical implementation of banks, the row copy operation is restricted to rows within the same sub-array. Second, the essential part of row copy are the two consecutive ACTIVATE commands. **However, the PRECHARGE command is indispensable to the operation as we found that the second ACTIVATE would not be recognized by the DRAM if we did not interpose the PRECHARGE command before it**.

3.1.2 AND/OR. By further reducing the timing intervals in the command sequence of Figure 3, we managed to **open three different rows simultaneously**. This allows us to utilize the charge sharing result to build logical operations, which happen in-place. We store all-zeros or all-ones to one of the opened rows in advance, execute the command sequence, and the logical AND/OR of the other two rows will finally appear in all three rows. We discuss how to preserve the operands in Section 4.2.

More specifically, in order to perform the logical AND/OR operations, we set both T_1 and T_2 to the minimum value, which means we execute the ACTIVATE(R_1), PRECHARGE, and ACTIVATE(R_2) commands in rapid succession with no idle cycles in between. Assuming that the addresses of the two rows, R_1 and R_2, are selected appropriately, the resulting command sequence will cause a single third row R_3 to be opened implicitly. We present the timeline of AND and OR operations in Figures 5 and 6, respectively. The rows shown in the figures are consistent with the physical layout: the row addresses are 0, 1, and 2 from top to bottom. Notice that both operations use exactly the same commands and timing. The only two differences are the rows used to store operands, which are enclosed in rectangles in the figure, and the operation-selecting constant saved in the remaining row.

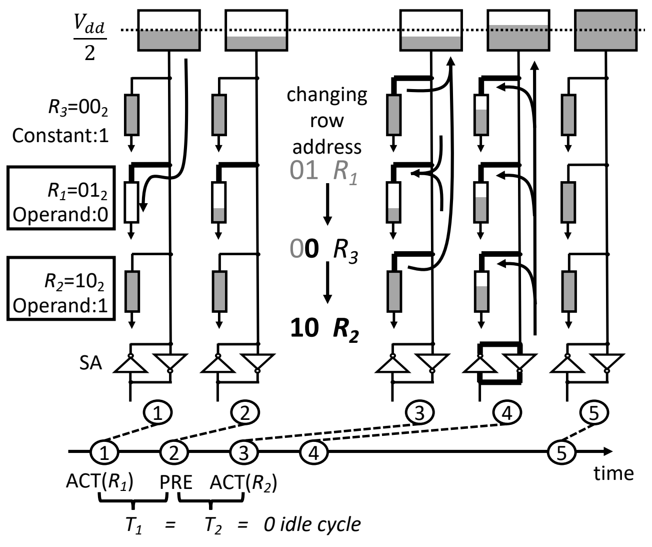

Figure 6: Logical OR in ComputeDRAM. R_3 is loaded with the constant one, and R_1 and R_2 store operands (0 and 1). The result ($1 = 1 \lor 0$) is finally set in all three rows.

R_3 \ R_1R_2	00	01	10	11
0	0	0	X	1
1	0	1	1	1

Figure 7: Truth table for the results produced from charge sharing among rows R_1, R_2 and R_3.

As before, in step ① of Figures 5 and 6, we send the first `ACTIVATE` command which opens R_1. In contrast to the procedure for row copy, we interrupt the first `ACTIVATE` by sending the `PRECHARGE` immediately after it in step ②. Due to the short timing interval, T_1, between the two commands, the sense amplifier is not enabled yet. This is important, because otherwise, the value in $R1$ will be restored in the bit-line and will overwrite other rows in the end. In step ③, the second `ACTIVATE` is sent, thus interrupting the `PRECHARGE` command. As described in Section 2, the `PRECHARGE` command does two things: it zeros out the word-line to close the opened row, and drives the bit-line to $V_{dd}/2$. An adequately small T_2 timing interval can prevent both parts from being realized, meaning that the opened row R_1 will not be closed. On its execution, the second `ACTIVATE` command will change the row address from R_1 to R_2. In the process of changing the activated row address, an intermediate value R_3 appears on the row address bus. Previously, the `PRECHARGE` was interrupted right in the beginning, so the bank is still in the state of "setting the word-line" as in the activation process. This will drive the word-line according to the value on the row address bus, leading to the opening of the intermediate row R_3. Besides this intermediate row, the specified destination R_2 will be opened at the end of step ③. Remember that R_1 is kept open from the start, therefore we have three different rows simultaneously activated. After the charge sharing, all cells and the bit-line reach to the same voltage level in step ④. Whether the resultant voltage is above $V_{dd}/2$ or not depends on the **majority value** in the cells of R_1, R_2 and R_3; we later discuss how to implement logical AND/OR based on this by setting the operation-selecting constant row in advance. Finally, in step ⑤, the result is stored in all three rows. Similar to the row copy operation, the logical AND/OR operations can be performed only within rows of the same sub-array.

An important consideration when implicitly opening a third row is how to control which row is opened. According to our experiment, when R_1 is 01_2 and R_2 is 10_2, $R_3 = 00_2$ is opened. Furthermore, we observed that the order of activation influences the behavior: when R_1 and R_2 are 10_2 and 01_2 respectively, $R_3 = 11_2$ is opened instead. Based on this observation, we speculate that the row address is updated from least to most significant bit, as shown in the middle section of Figures 5 and 6. Notice that in the case that the addresses of R_1 and R_2 have more divergence in their bits, we found that additional rows (more than three) are opened. To exert control over the behavior of our operations and for simplicity, we fix the lower two bits of the addresses of R_1 and R_2 to 01_2 and 10_2 and require their higher bits to be identical. This forces the address of R_3 to have its two lower bits be 00_2 and higher bits be the same as R_1 and R_2, ensuring that only one intermediate row is activated.

We now discuss how the content of the rows affects the outcome of the operation, and how we construct logical AND/OR. Theoretically, assuming that the cell capacitance in all three rows is the same and that they are activated simultaneously, the outcome of the operation would be the majority value. That is, if at least two rows hold a value of one, the outcome will also be one, and *vice versa*. However, in reality, we open row R_1 first, therefore it has more time to influence the bit-line, thus the three rows are not equivalent. Figure 7 presents the truth table from experimentally collected results for all combinations of values in R_1, R_2, R_3. Apart from the combination where $R_1 = 1$, $R_2 = 0$, and $R_3 = 0$, marked with X to indicate an unpredictable outcome, all other combinations generate the desired result. Based on this observation, we only use the combinations from the truth table in Figure 7 that generate robust results, choosing the appropriate rows as operands and the remaining as the initial constant, to implement the logical AND/OR operations. Fixing the value of R_1 to zero reduces the truth table to the dotted circle, meaning a logical AND would be performed on the values in R_2 and R_3. Similarly, by fixing the value of R_3 to constant one, we reduce the truth table to the solid circle, this time performing a logical OR on the values in R_1 and R_2.

Therefore, in addition to the timing intervals, another prerequisite of logical AND/OR is the constant zero/one. Before we perform an AND operation, we need to copy all-zeros to the row R_1. For OR operation, similarly, we need to copy all-ones to the row R_3. When the DRAM is initialized for computation, we store constant zeros and ones into two reserved rows in each sub-array.

3.2 Influence on Refresh Rate

Compared with normal memory access, we need to think about whether our in-memory compute operations require a closer previous refreshment and whether they give a shorter retention time to the cells they touch. For the logical AND/OR operations, we do assume that the cells in all three operand rows are fully charged/discharged, which means they need to be refreshed just before the operation. In our implementation, as described in Section 4.2, we copy operands to reserved rows to perform AND/OR, thus they are always recently refreshed. For row copy, the first `ACTIVATE` is not interrupted and the sense amplifier is enabled. So, as long as the charge in the cell is good for a normal access, it should be good for a row copy. We preserved enough time after the last `ACTIVATE` for both row copy and AND/OR, thus the retention time should not be influenced. Only the retention time for the source row in a row copy might be reduced due to a short $T1$ (in Figure 3). However, the length of $T1$ is not critical to the functionality, and can be enlarged to make it more conservative.

3.3 Operation Reliability

In previous sections, we described the command sequence and timings that perform the row copy and logical AND/OR operations. Under ideal conditions, the operations should reliably produce the desired computations. However, through our experiments, we observed instances of rows and columns that when utilized would produce erroneous results. The source of these errors stems from manufacturing variations across columns and the practice of row remapping to circumvent faulty cells [43].

3.3.1 Manufacturing Variations. Due to imperfections in the manufacturing process, size and capacitance of elements are not uniform across the chip [13, 44]. These variations make the columns behave differently given the same command sequence. The behavior divergence leads to a partial execution of an operation (e.g., only a portion of the columns in a row are copied with a row copy operation), or erroneous results. The offending columns are not necessarily incapable of performing the operation. Rather, we have found that often they require different timing intervals to achieve the operation. As it is not always possible to find a timing interval that makes **all** columns work, the timing interval that covers most columns is selected. The columns that still fail to perform the operation are regarded as “bad” columns and are excluded from in-memory computations.

3.3.2 Row Remapping. Apart from variations due to the manufacturing process, imperfections that render some DRAM cells unusable can also manifest. Such errors are discovered and resolved during post-manufacturing testing, with the row addresses pointing to these faulty cells being remapped to redundant rows [5]. This creates a complication to our technique. To perform our in-memory computing operations, we ensure that the operand rows are in the same sub-array by checking their addresses. However, for the remapped rows, the row addresses are not consistent with the physical locations. As we cannot guarantee that they are in the same sub-array with any other row, we mark these as “bad” rows to be excluded from in-memory computations. Section 4.5 discusses how we use software remapping and an error table to solve these challenges.

4 IN-MEMORY COMPUTE FRAMEWORK

In the previous section, we described how we implemented three basic in-memory operations. However, the three operations fall short of allowing us to perform arbitrary computations as inversion is needed for generality. Furthermore, as variations and defects in DRAM memories are inevitable, there will always be rows and columns that cannot be used for computation. In the following section, we propose a software framework that could perform arbitrary computations using the three basic operations as building blocks. In parallel, we address the issue of errors due to process variation and manufacturing imperfections.

4.1 Arbitrary Computations with Memory

Having the row copy and logical OR/AND operations as a starting point, the missing functionality for arbitrary computations is the NOT operation. To resolve this limitation, previous work suggested *in situ* implementation of the NOT operation in DRAM [57]. We propose an alternative approach that avoids modifications to DRAM design and requires the existence of both the regular and the negated version of a value to perform computations. Thus, all variables in our model are composed of two parts: one in regular form and one negated. The availability of both versions is then utilized to create universal functions such as NAND, which could be used to build arbitrary computations. Equations 1 to 5 provide examples of different possible operations. The right-hand side of all equations is constructed using only logical AND and OR.

To allow for arbitrary computations at any given point of execution, for every nominal computation, additional steps need to be taken to generate the negated pair. For example, to compute XOR between A and B, using only AND/OR operations and the negated inputs requires three operations: $\bar{A} \wedge B$, $A \wedge \bar{B}$, and the logical OR of the two previous results. Under our scheme, an additional three operations ($\bar{A} \vee \bar{B}$, $A \vee B$, and the logical AND of the two previous results) are required to compute XNOR, so as to extend the pairwise value invariant of our framework to subsequent computations. As is evident, the adoption of a pairwise value format utilizes twice the amount of memory space and twice the number of operations. We believe that compared to the improvements in functionality, these overheads are acceptable. The performance overhead imposed by the pairwise value format is alleviated by the mass parallelism that row-wise computation enables. Furthermore, we actually don't have to calculate all the negated intermediate values for some complex functions. Through logical minimization, some intermediate steps can be reduced.

$$\text{NOT: } \neg(A, \bar{A}) = (\bar{A}, A) \tag{1}$$
$$\text{AND: } (A, \bar{A}) \wedge (B, \bar{B}) = (A \wedge B, \bar{A} \vee \bar{B}) \tag{2}$$
$$\text{OR: } (A, \bar{A}) \vee (B, \bar{B}) = (A \vee B, \bar{A} \wedge \bar{B}) \tag{3}$$
$$\text{NAND: } \neg((A, \bar{A}) \wedge (B, \bar{B})) = (\bar{A} \vee \bar{B}, A \wedge B) \tag{4}$$
$$\text{XOR: } (A, \bar{A}) \oplus (B, \bar{B}) = ((\bar{A} \wedge B) \vee (A \wedge \bar{B}), (\bar{A} \vee \bar{B}) \wedge (A \vee B)) \tag{5}$$

4.2 Implementation Choices

AND and OR operations can be performed on many different pairs of rows in one sub-array. But as shown in Figures 5 and 6, the computation result overwrites the data in all of the three opened rows. In order to preserve the operands, and also to make the software simple and consistent, we only perform the computation in the first three rows in each sub-array. More specifically, the lower nine bits of R_1, R_2 and R_3 are always 000000001_2, 000000010_2 and

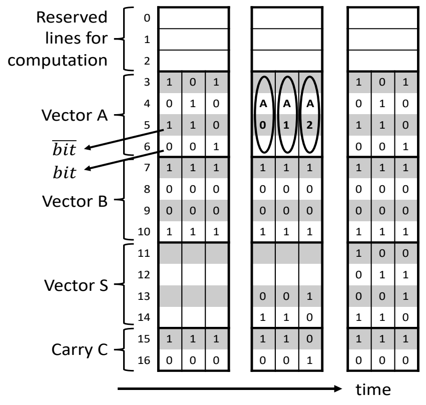

Figure 8: An example of element-wise ADD in ComputeDRAM. Each vector has three, 2-bit elements. Each element is stored vertically, with nominal bits in white boxes and negated bits in gray boxes.

000000000_2 (assuming 512 rows in a sub-array). These three rows in each sub-array are reserved for computation in our model. With more complex software, we could perform more operations in place without moving data to the designated rows. But for simplicity in our design, we currently always use row copy to copy the operands and the operation-selecting constant to these three reserved rows, perform the computation, and copy the result back to the destination row.

4.3 Bit-Serial Arithmetic

As our operations work at row-level, a parallelism of 65536× can be achieved if we perform computation at a single-bit granularity.[1] Even more parallelism can be exploited if we use multiple DRAMs on a shared command bus. This makes bit-serial computation a great candidate for our design.

Bit-serial computing has been previously used with success in applications such as image and signal processing [8, 10, 17, 53], Deep Neural Networks (DNN) [21], and computer vision [20] due to the extremely high throughput it can provide. Another benefit is the flexibility for arbitrary width computation. The most important reason to use bit-serial computing is that if we instead used a bit-parallel approach, we would not be able to perform complex operations, such as addition, which require carry propagation. This is because the use of DRAM prohibits us from propagating carry across different columns.

In our scheme, to perform an operation in DRAM, the input vectors should be aligned, and have the same length and width. The Least Significant Bit (LSB) of the value resides in the lowest row of that vector. The negated bit is stored in the row right above the nominal bit; the rows holding the negated data are shaded in the Figure 8.

A vector containing m n-bit items is stored in $2n$ rows and m bits of columns, with each item occupying one bit, out of eight, of a column. Due to the pairwise value format described in Section 4.1 that allows for arbitrary computations, a total of $2n$ rows are required. Figure 8 provides an example of an addition in such a scheme, and a visualization of how the data of a bit-serial computation scheme would be laid out in DRAM. The example works on two, 2-bit operand vectors, A and B, where A has three items: 0, 2, 1, and the items in B are all 1. The row addresses are marked on the left. As mentioned in Section 3, the first three rows – 0,1, and 2 – are reserved for performing operations. The rows reserved for carry propagation – 15 and 16 – are also visible. We have omitted the rows holding the all-one and all-zero and the rows allocated for intermediate results.

At first, the carry bits are initialized to zero using the all-zero constant row. The computation proceeds from low bits to high bits. Rows 5, 6, 9, 10 are used to calculate the LSB of the summation vector S and the carry C. The DRAM operations are applied to the entire row, so all of the three items in S are generated in parallel. The same operation is performed for subsequent bits, until we have processed all bits. Finally, the carry bit indicates if any of the items in S have overflowed.

4.4 Copy Across Sub-arrays

As computation progresses and the variable space grows, we may be required to move data across bank sub-arrays. However, we cannot achieve that using the in-memory row copy operation, as rows in different sub-arrays don't share the same bit-lines. In such cases, we propose to read the whole row from the source sub-array, store the data in the MC, and then write it into the destination sub-array. Although the MC would require multiple `READ`/`WRITE` commands to go through the columns of the entire row, this approach is preferable to the alternative of moving the data between the memory and processor, as it reduces the distance that data has to travel.

4.5 Error Table

Finally, an essential part of our proposed framework is to address the reliability problems brought up in Section 3.3. We resolve the problem of "bad" rows and columns by excluding them from computations in software. This requires that a thorough scan be performed on the entire module before it is utilized. The scanning process would generate an address record of the discovered "bad" columns and "bad" rows; we call the address record table "error table". The error table is utilized on a custom address translation layer between the software library and the memory command scheduler.

From the perspective of the software library, the items in one vector are stored in consecutive columns and rows, indexed by virtual addresses. As the software invokes the provided API functions to perform in-memory computations, the translation layer while translating virtual to physical addresses skips "bad" columns and "bad" rows. After the appropriate remapping, the command sequence is sent from the memory command scheduler. A re-scan of the DRAM module may be performed periodically to update the error table with columns and rows that became "bad" as a result of natural wear out or changes in environmental variables (e.g. temperature). We evaluate the effect of environmental factors on our framework in Section 6.3.

[1]There is a total of 65536 bits in a row throughout the module: 8 bits per column, 1K columns per chip, and eight chips per rank.

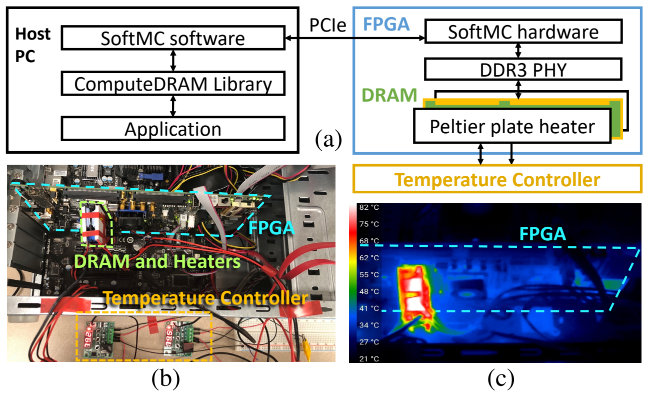

Figure 9: (a) Schematic diagram of our testing framework. (b) Picture of our testbed. (c) Thermal picture when the DRAM is heated to 80 °C**.**

5 EXPERIMENTAL METHODOLOGY

We evaluated our work using a set of custom testbeds, each composed of a host system and a Xilinx ML605 FPGA board [64] connected through a PCIe bus. Figure 9 presents a schematic diagram and photos of our hardware testbed.

We leveraged SoftMC v1.0 [30], an open-source programmable memory controller, to directly control the DRAM module attached to the FPGA. For our work, we extended SoftMC to support dual-rank modules and to allow reading and writing non-repetitive bytes in a burst. SoftMC enables us to experiment with different command sequences and timing intervals, but also imposes limitations. The dependence of SoftMC on the Xilinx Vertex-6 MIG module [63] constrains the operating frequency of the memory command bus and data bus at 400MHz and 800MHz, respectively, regardless of the maximum attainable rate of the DRAM module. As a result, the timing intervals in our experiments are quantized to multiples of 2.5ns. SoftMC can only address a fixed portion of DRAM: a single rank, containing eight banks, each having 1K columns and 32K rows. Thus, the overall addressable memory is constrained to 2GB. The limitations of SoftMC do not influence our design as the provided functionality enables a representative space for exploration. As, currently, there is no open source command-level memory controller for DDR4, we used DDR3 chips for our evaluation. Although DDR4 has a higher working frequency and a lower V_{dd}, its basic commands are exactly the same as DDR3. Thus, we believe that DDR4 also presents opportunity for performing in-memory operations.

For assessing the effect of different supply voltages on the operation's robustness, we re-programed the power management module of our FPGA to control the supply voltage of the DRAM. For accessing the effect of temperature on the reliability, we created a custom setup using peltier devices to heat up both sides of the DRAM modules; temperature was controlled through temperature controller boards.[2]

[2]HiLetgo W1209 12V DC Digital Temperature Controller Board; resolution, measurement, and control accuracy were at a granularity of 0.1 °C and the refresh rate was 0.5 seconds.

Table 1: Evaluated DRAM modules

Group ID: Vendor_Size_Freq(MHz)	Part Num	# Modules
SKhynix_2G_1333	HMT325S6BFR8C-H9	6
SKhynix_4G_1066	HMT451S6MMR8C-G7	2
SKhynix_4G_1333B	HMT351S6BFR8C-H9	2
SKhynix_4G_1333C	HMT351S6CFR8C-H9	4
SKhynix_4G_1600	HMT451S6AFR8A-PB	2
Samsung_4G_1333	M471B5273DH0-CH9	2
Samsung_4G_1600	M471B5273DH0-CK0	2
Micron_2G_1066	CT2G3S1067M.C8FKD	2
Micron_2G_1333	CT2G3S1339M.M8FKD	2
Elpida_2G_1333	EBJ21UE8BDS0-DJ-F	2
Nanya_4G_1333	NT4GC64B8HG0NS-CG	2
TimeTec_4G_1333	78AP10NUS2R2-4G	2
Corsair_4G_1333	CMSA8GX3M2A1333C9	2

We implement the software framework described in Section 4 by wrapping the software API of SoftMC in functions that implement basic (row copy, logical AND/OR) and complex (XOR, ADD) operations, thus providing a higher-level of abstraction to the user. For all basic operations, the sequence of DRAM commands is first buffered on the FPGA and then sent to memory as a carefully timed batch by the hardware part of SoftMC. This ensures that the latency of the PCIe bus does not affect the timing in operations.

6 EVALUATION

In this section we evaluate our proposed techniques and characterize their behavior across a wide range of DRAM chips. We evaluate across a total of 32 DDR3 DRAM modules from seven different manufacturers, using at least two modules from each configuration.[3] Each module is clustered in one of 13 groups based on its configuration and part number. Table 1 provides a detailed breakdown of the different DRAM configurations we used. The frequency in the group ID refers to the maximum attainable data bus frequency. As we mentioned in Section 5, our testing infrastructure issues memory commands at a fixed frequency of 400MHz to all the modules, and the data bus frequency is always 800MHz.

6.1 Proof of Concept

To validate the feasibility of computing with memory, we built a test benchmark based on the command sequence in Figure 3. Our test benchmark selects a random sub-array from each bank of a DRAM module and performs an exploratory scan to identify pairs of timing intervals that produce successful computations.

At the beginning of the test, the sub-array is initialized to predetermined values. Following the execution of the command sequence with a given pair of timing intervals, we read out the target sub-array. Based on the comparison with the predetermined values, we check whether any computation is performed or not. If row copy or logical AND/OR is found, we record the type of computation and the row-wise success ratio. The row-wise success ratio is defined

[3]TimeTec and Corsair are not DRAM IC manufacturers *per se*, but use chips from other companies to build modules. The printed serial number on the chips are T3D2568HT-10 and HYE0001831, respectively.

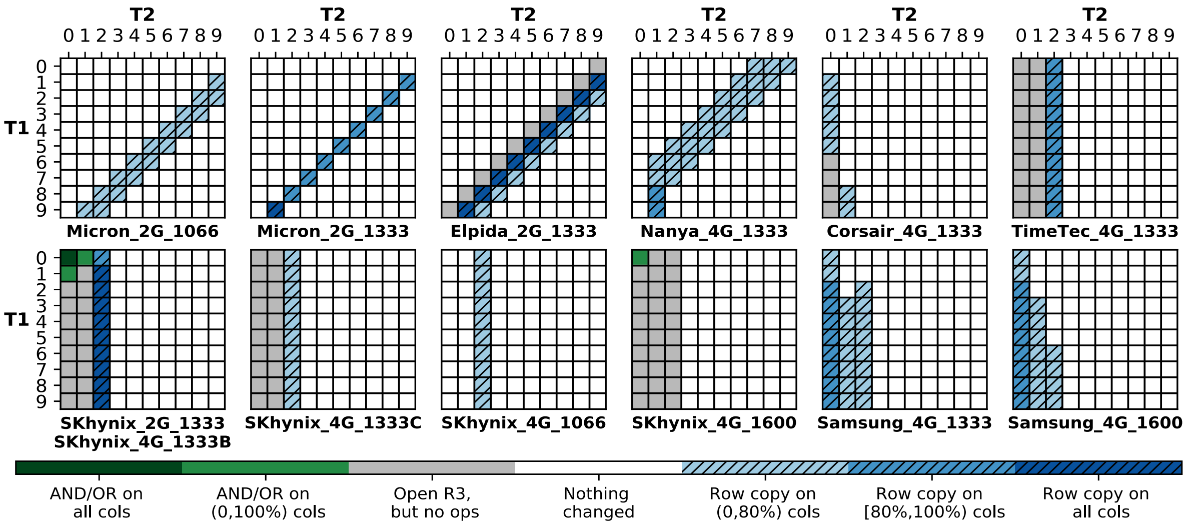

Figure 10: Heatmap of successfully performed operations for each DRAM group with different timing intervals. The integer timing intervals represent the amount of idle cycles between memory commands. Each idle cycle takes 2.5ns.

as the ratio of columns with successful computation over the total number of columns in a row. We ran the test on all 32 modules.

Figure 10 presents the results of our exploratory scan for the 13 different DRAM groups. Due to similarity of results across modules of the same group, we opted to demonstrate the result of a single module from each group. Two of the groups, SKhynix_2G_1333 and SKhynix_4G_1333B, have exactly the same results, so we use a single heatmap for them.

Each colored box in the heatmap provides the row-wise success ratio for a given pair of timing intervals in command cycles. Thus, a timing interval $T_1 = 2$ will introduce two idle cycles ($5ns$) between the first ACTIVATE and the PRECHARGE commands in Figure 3. As we are more interested in capturing the feasibility of an operation, we plot each heatmap using the results from the bank that provided the highest row success ratio for each module.

Blue(hatched) boxes identify timing interval pairs that resulted in a row copy operation, whereas green boxes identify timing interval pairs that resulted in AND/OR operations. Gray boxes indicate that a third row was modified, but the result did not match any interesting operation. White boxes indicate that all the data in other rows remained the same, and there was no meaningful result in the opened two rows. The shade of each color indicates the row-wise success ratio in an operation. Darker shades signifies higher success ratios. Thus, the darkest blue and green colors indicate timing interval pairs that lead to fully functional in-memory operations. That is, using the selected timing interval pair, we were able to produce, across all bits of a row, correct results in at least one sub-array. The existence of at least one heatmap with both dark blue and dark green boxes acts as a proof of concept that we can perform row copy and logical AND/OR using off-the-shelf, unmodified, commercial, DRAM.

In more detail, from Figure 10, we observe that nearly all configuration groups present the capability of performing row copy in at least a portion of the columns in a sub-array. Furthermore, we observe that successful timing interval configurations for row copy follow two patterns: the vertical line pattern and the diagonal line pattern, with most groups exhibiting a vertical line pattern. The timing interval pairs in the vertical line pattern support the explanation in Section 3, where we argue that row copy can be implemented using a small T_2 timing interval. In contrast, DRAMs from Micron, Elpida, and Nanya exhibit a diagonal pattern. We speculate that in these cases, it is the sum of T_1 and T_2 that determines whether the row copy operation is performed. Thus, the success of the operation depends on the timing interval between the two ACTIVATE commands, with the timing of the intermediate PRECHARGE command not affecting the operation. More specifically, we speculate that DRAM modules in those groups perform a check on the timing interval between the first ACTIVATE and the PRECHARGE command, and if the subsequent PRECHARGE command is scheduled too close to the previous ACTIVATE, it will not be issued immediately. That is, the PRECHARGE command is buffered inside the chip and is scheduled with a delay, meaning that the effective T_2 timing interval is smaller than the one externally defined in our test benchmark. Thus, the operational rationale for row copy operation in those groups can still be explained by Section 3.

In regard to logical AND/OR operations, we observe that only DRAMs in groups SKhynix_2G_1333 and SKhynix_4G_1333B are able to perform the operations across all columns of the sub-array. Modules in group SKhynix_4G_1600 can also perform both operations but not across all columns of the sub-array. Although only limited groups exhibit the capability of performing AND/OR operations, half of the groups are able to open a third row (shown using gray boxes in the heatmaps). To realize the logical operations, charge sharing among three different rows is required, therefore opening a third row is the key prerequisite for logical operations. Since these groups have met this key prerequisite, we speculate that they have the potential to perform AND/OR operations at

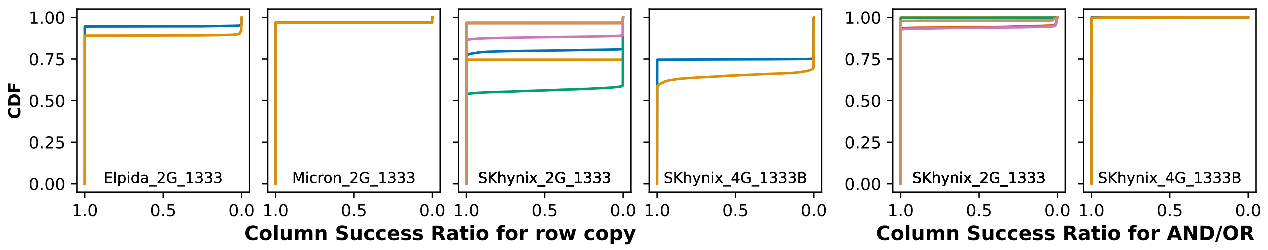

Figure 11: The CDF shows the stability of the row copy and AND/OR operation for each memory group.

a different command frequency. We leave the evaluation of this speculation for future work.

Overall, the most important observation from Figure 10 is the existence of **off-the-shelf, unmodified, commercial, DRAM modules in which we can perform both in-memory row copy and logical AND/OR operations**. Furthermore, **all vendors** have some configurations of DRAM that can perform the row copy operation. The modules in the identified working group, SK hynix DDR3 DRAMs at 1333MHz, are fabricated and distributed as regular DRAM memory, and have a wide adoption in commercial configurations. Based on the above, we speculate that logical AND/OR operations could also become operational in other DRAM modules with minimal hardware modifications and cost. Specifically, for the modules that cannot perform operations, we hypothesize that there exist hardware in them that checks the timing of operations and drops commands that are too tightly timed. Although we don't know the details of that hardware due to the lack of public information, we believe that removing, or making configurable, this portion of the design is feasible without significant overhead. At the same time, such an option will open up more opportunities for research like ComputeDRAM.

6.2 Robustness of Operations

After presenting a proof of concept for in-memory computing using off-the-shelf DRAM modules, we proceed to provide a more thorough exploration of the behavior of each operation. An important consideration for the practical application of our techniques is to find out the portion of the DRAM chip that can be reliably utilized by the framework we described in Section 4. To assess that, we performed a more detailed robustness test for each operation. The robustness test was performed under nominal supply voltage (1.5V) and at room temperature (chip temperature = 25-30 °C).

6.2.1 Row Copy. We ran the row copy reliability benchmark only on groups that exhibit the ability of performing perfect row copy in at least one sub-array – Micron_2G_1333, Elpida_2G_1333, SKhynix_2G_1333, SKhynix_4G_1333B. For each module, we fixed the timing intervals to the pair that generated the best row copy result in the previous exploratory test. For a single bank from each module, we performed 1000 times the row copy reliability test, in which different randomly-generated data are copied back and forth all over 32K rows. Subsequently, we produced a success ratio for each column of a sub-array based on the times that it contained the correct result. Figure 11 presents the Cumulative Distribution Function (CDF) of success ratio for the different DRAM groups, with each module in the group represented by a different line. The data for row copy is shown in the four leftmost sub-plots of Figure 11. The sharp rise at the beginning of the CDF indicates that in all modules, at least half of the columns can reliably perform the row copy operation. Across different modules, 53.9% to 96.9% of the columns present 100% success ratio when performing the row copy operation. The flat horizontal line in the CDF indicates that most of the remaining columns always fail to perform the row copy operation (having a zero success ratio). This result validates our strategy of producing an error table of "bad" columns to our software framework and completely avoid utilizing the offending columns.

6.2.2 AND/OR. We performed a similar test for the logical AND/OR operations. Targeting eight modules from groups SKhynix_2G_1333 and SKhynix_4G_1333B, for all sub-arrays in eight banks from each module, we performed the operations 10000 times, half AND and half OR, on the first three rows. Again, the operands used in the test are randomly generated. Subsequently, we produced a success ratio for each column of a sub-array. In the two rightmost sub-plots of Figure 11, we can see a sharp rise in the start of those CDFs. Among our tested modules, 92.5% to 99.98% percent of columns exhibit 100% success ratio when performing the logical AND/OR operations.

6.3 Effect of Supply Voltage and Temperature

Having as a reference the robustness study at nominal conditions, we expanded our study to supply voltages from 1.2V to 1.6V, with a step of 0.05V, and chip temperatures from 30 °C to 80 °C, with a step of 5 °C. The range of our study was dictated by practical considerations. For supply voltage, the upper limit is imposed by the infrastructure, while the lower limit is imposed by the fact that most of the DDR3 modules we tested could not be read/written with a supply voltage below 1.2V. For temperature, we chose to start from room temperature and study up to 80 °C on the DRAM package, as temperatures outside this range are not common for commodity applications. Timing intervals are fixed to the optimal ones identified on the exploratory scan (see Figure 10). Similar to the initial robustness test, under each environment setup, we ran 10 row copy scans and 100 AND/OR scans, and collected the success ratio for each column.

6.3.1 Supply Voltage. The top part of Figure 12 presents how the amount of the columns that perform row copy and AND/OR operations reliably (100% success ratio) changes as a function of supply

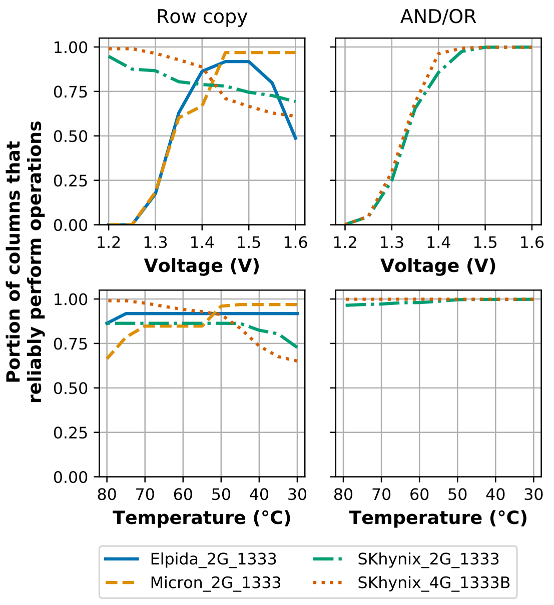

Figure 12: The portion of columns that could reliably perform operations with varied supply voltage and temperature for different DRAM groups.

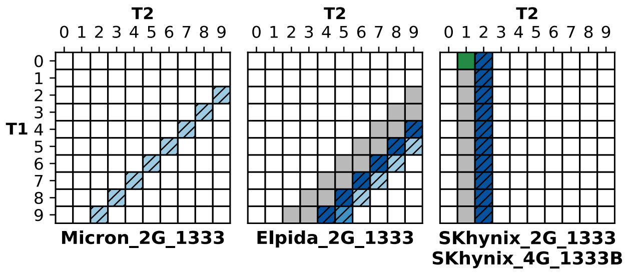

Figure 13: Heatmap of successfully performed operations at a supply voltage of 1.2V – only DRAM groups that reliably performed operations at nominal voltage are included; for interpreting colors, see the colorbar of Figure 10.

voltage for different memory groups. Each line represents the average result among the modules in that group.

In the top part of Figure 12 we observe that: for row copy, Micron modules prefer higher voltage, SK hynix modules prefer lower voltage, and for Elpida modules, the reliability is maximized at nominal supply voltage. Though the effect of supply voltage on the reliability differs across vendors, the pattern stays the same for modules in the same group, thus the behaviour is affected by implementation choices.

Reducing supply voltage increases propagation delay [54] making the circuit "slower". Thus, assuming a fixed frequency, the time period of one cycle at nominal supply voltage is equivalent to more than one cycle at 1.2V supply voltage; that is, the signal needs more time to fully propagate due to increased propagation delay. Therefore, if we alter the supply voltage, we could have a slightly changed "effective" timing interval. The closer this "effective" timing interval is to the "real optimal" timing interval, the higher the success ratio goes. Because of different implementation details, the "real optimal" timing intervals vary among the modules from different manufacturers. That is the reason why the effect of supply voltage on reliability differs across manufacturers. For AND/OR operations, as the voltage drops, the circuit gets so slow that the first ACTIVATE cannot even open the first row yet. Thus the success ratio drops as well.

To verify this timing assumption, we conducted a second timing interval scan using a supply voltage of 1.2V. As we can see in Figure 13, the optimal timing intervals are larger than the previous ones, which proves that the lower voltage makes the circuit slower, and we need more cycles to achieve the same effect.

6.3.2 Temperature. In regard to the effect of temperature on robustness (see the bottom part of Figure 12), we observe that the effect of increasing the temperature is similar to lowering the supply voltage. Essentially, the sub-plots for temperature present a "zoom in" on the top part of Figure 12. This is because at higher temperature (inside our range), the circuit's carrier mobility decreases, thus increasing the propagation delay and making the circuit slower [38, 54]. From the perspective of timing, it has the same effect as a lower supply voltage. Based on the above results, we conclude that our system keeps working within a reasonable variation of supply voltage (±0.1 V) and temperature. Furthermore, we find that the system designer needs to take into consideration the operational temperature of the system when selecting memory vendors.

Overall, we expect that ComputeDRAM enabled DRAMs will require a production step that "bins" modules based on their ability to perform in-memory computations, in what voltage and temperature ranges, and with what level of reliability, similar to the process for CPUs.

7 DISCUSSION

In this section we present a qualitative discussion of the current implementation of ComputeDRAM; we enhance this discussion by including back-of-the-envelop quantitative numbers.

The quantitative calculations are based on an extension of the software framework discussed in Section 4 that implements a series of vector functions: element-wise AND, OR, XOR, SHIFT, and ADD. All extended functionality was constructed using only the three basic in-memory operations we evaluated in previous sections. The added operations follow the convention of bit-serial computing that we described in Section 4.3, allowing us to seamlessly configure the operation's bit-width. Table 2 presents the number of cycles spent on the memory command bus for the computation of a single bit for each of the vector function. The memory command cycles spent on N-bit operations can be calculated by scaling $N\times$ the data of Table 2.

Table 2: Cycles of memory commands for 1-bit operation using SK hynix modules under our hardware setup

row copy	SHIFT	AND	OR	XOR	ADD
18	36	172	172	444	1332

7.1 Computational Throughput

The computational throughput and performance of ComputeDRAM is significantly affected by the characteristics of the targeted computation; such as, the level of parallelism and the scale of computation. Compared against the cost of performing a single scalar operation in a modern CPU, ComputeDRAM will not provide any benefit, as a single ADD operation accounts for over a thousand cycles and the DRAM command frequency is lower than that of a processor core. Furthermore, if the data being computed exhibit high locality, the immense bandwidth out of on-chip SRAMs and Register Files (RF) can significantly increase peak performance for CPUs. To achieve high computational throughput, ComputeDRAM exploits the fact that its computational overhead does not change as we move from scalar to vector operations, up to vectors of 64K elements (for a single module), due to the constant cost of performing row-wise computation across all eight chips of the DRAM. For example, for row copy, our technique requires 18 memory cycles with a peak bandwidth of 182GB/s using a single DDR3 module. On ComputeDRAM, 8-bit AND/OR operations take 1376 cycles providing a peak throughput of 19GOPS, and 8-bit ADD operation takes 10656 cycles for a peak throughput of 2.46GOPS. We can enhance this baseline performance by interleaving the issuing of commands across different banks. Moreover, we can extract further parallelism by building a board with multiple DRAM slots, where a single MC will control all DRAM modules simultaneously through broadcasts in the shared memory command bus. Thus, ComputeDRAM's ideal targets are massive computations that exhibits a large amount of main memory accesses.

The main overhead of ComputeDRAM is the duplicated computation and storage required by our proposed framework. This inefficiency can be addressed at the compiler level. As more complex computation is offloaded to ComputeDRAM, the compiler which is responsible for translating high-level code to a sequence of DRAM commands can perform optimization to reduce the number of commands, by minimizing both the nominal and complementary (negated version) intermediate computations. We have already manually performed such optimization in the implementation of the ADD operation in our software framework.

As computation in CPU and DRAM is decoupled, the overall computational capability of the system is increased. Furthermore, as computation is moved from the CPU to the main memory the interconnect's utilization is reduced. This can lead to reduced latency and runtime.

7.2 Energy Efficiency

The main benefit of ComputeDRAM is its high energy efficiency, as it eliminates the high energy overhead of transferring data between CPU and main memory. If data is fetched from memory to the CPU, computed on, and stored back in memory, ComputeDRAM is 347× more energy efficient than using a vector unit for row copy, and 48× and 9.3× for 8-bit AND/OR and ADD, respectively. [4] Our baseline is the energy used to read the operands from DRAM and write the result into DRAM, as memory access takes the main portion of the total energy consumption in our target scenario. We assume that the data and its compliment is resident in the DRAM at the beginning of the computation.

Notice that in our work many memory commands are interrupted before their nominal end time. As we did not reduce the energy cost for these command in the power model, the previous energy numbers present a conservative estimate.

[4]We used VAMPIRE [26], a command-trace based DRAM power model, for the energy estimation.

8 RELATED WORK & APPLICATIONS

To our knowledge, this is the first work to demonstrate the feasibility of performing row copy, logical AND, and logical OR in off-the-shelf, unmodified, commercial, DRAM.

Near-memory [3, 11, 29, 35, 55] and in-memory [19, 22, 23, 27, 28, 33, 34, 39–41, 46, 51, 52, 56, 59] computing has been extensively explored over the past 25 years in an attempt to provide a solution to the "Memory Wall" problem. In contrast to our work, all past efforts required hardware modifications.

The concepts of copying data across rows of a DRAM sub-array, and performing logical operations through multi-row activation and charge sharing in DRAM memory were, to our knowledge, first discussed by Seshadri et al. [56, 57]. In contrast to our work, their designs require hardware modification on the DRAM to allow in-memory computations. Modifications to the DRAM control logic were proposed to allow back-to-back `ACTIVATE` commands, a prerequisite of the design for performing a copy between two rows [56]. In contrast, we were able to successfully implement the row copy operation, without any modifications to the DRAM, by interposing and quickly interrupting a `PRECHARGE` command between two `ACTIVATE` commands.

Similarly, Seshadri et al. [57] proposed to modify the row decoder to allow simultaneous activation of three rows, a requirement for performing logical operations based on charge sharing; coincidentally, a similar work [1] that targeted SRAM caches was published the same year. In our work, we show (Section 3.1.2) that, without any modifications to the DRAM, it is feasible to simultaneously activate three rows by interrupting both `ACTIVATE` and `PRECHARGE` in the command sequence shown in Figure 3.

More generally, multiple prior works proposed the integration of processing logic *inside* the DRAM [18, 19, 22, 27, 34, 39, 51, 52, 59], or modifications on the DRAM design and enhancement on the basic memory cell to enable native computational capabilities [4, 24, 42]. None of these solutions have gained widespread industry adoption, largely due to requiring modifications of the existing DRAM design and additional circuits to be added to cost optimized and low-margin RAM implementations.

The introduction of 3D-stacked memory architectures where a logic layer is coupled with a memory scheme (e.g., HMC 2.1 [31]), enabled the space for designs that expand the logic layer to include computational capabilities [25, 50]. Though such schemes increase the available bandwidth, compared to off-chip memory, they fall short of fully exploiting the internal bandwidth of a DRAM.

In parallel, the emergence of new materials and technologies for memories has inspired works that study how non-CMOS memory technologies can be utilized to perform in-memory computations [28, 33, 40, 41, 46].

Bit-serial architectures were extensively used on early massively parallel processing (MPP) machines such as the Goodyear

MPP [6, 7, 9], and Connection Machine CM-1 and CM-2 [61]. More recently, Micron proposed In-Memory Intelligence [24], a bit-serial computing in memory architecture. Its single-bit processing units are able to perform basic operations such as AND and XOR, with more complex operations constructed through sequences of basic primitive operations. The high levels of parallelism and throughput make bit-serial architectures ideal for big-data problems in image [53] and signal [10] processing, Deep Neural Networks (DNN) [21], computer vision [20].

9 CONCLUSION

We have demonstrated for the first time the feasibility of performing in-memory operations in off-the-shelf, unmodified, commercial, DRAM. Utilizing a customized memory controller, we were able to change the timing of standard DRAM memory transactions, operating outside of specification, to perform massively parallel logical AND, logical OR, and row copy operations. We designed an algorithmic technique to compute arbitrary computations based on the three available, non inverting, in-memory operations. We incorporated our algorithmic technique in a software framework to run massively parallel bit-serial computations and demonstrate it working in a real system. Finally, we characterized the capabilities and robustness of the in-memory compute operations across DDR3 DRAM modules from all major DRAM vendors, and under different supply voltages and temperatures.

This work has the potential for large impact on the computing industry with minimal hardware design changes. Currently, DRAM is solely used to store data. But our discovery that off-the-shelf DRAM can be used to perform computations, as-is, or with nominal modifications by DRAM vendors, means that with a small update to the DRAM controller, all new computers can perform massively parallel computations without needing to have data transit the memory bus or memory hierarchy.

ACKNOWLEDGMENTS

This material is based on research sponsored by the National Science Foundation (NSF) under Grant No. CCF-1453112 and the Air Force Research Laboratory (AFRL) and Defense Advanced Research Projects Agency (DARPA) under agreement No. FA8650-18-2-7862. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Air Force Research Laboratory (AFRL), the Defense Advanced Research Projects Agency (DARPA), the NSF, or the U.S. Government.

REFERENCES

[1] Shaizeen Aga, Supreet Jeloka, Arun Subramaniyan, Satish Narayanasamy, David Blaauw, and Reetuparna Das. 2017. Compute Caches. In *High Performance Computer Architecture (HPCA), 2017 IEEE International Symposium on*. IEEE, 481–492.

[2] Amogh Agrawal, Akhilesh Jaiswal, Chankyu Lee, and Kaushik Roy. 2018. X-SRAM: enabling in-memory boolean computations in CMOS static random access memories. *IEEE Transactions on Circuits and Systems I: Regular Papers* 99 (2018), 1–14.

[3] Junwhan Ahn, Sungjoo Yoo, Onur Mutlu, and Kiyoung Choi. 2015. PIM-enabled instructions: a low-overhead, locality-aware processing-in-memory architecture. In *Computer Architecture (ISCA), 2015 ACM/IEEE 42nd Annual International Symposium on*. IEEE, 336–348.

[4] Avidan Akerib, Oren Agam, Eli Ehrman, and Moshe Meyassed. 2012. Using storage cells to perform computation. US Patent 8238173B2.

[5] K. Arndt, C. Narayan, A. Brintzinger, W. Guthrie, D. Lachtrupp, J. Mauger, D. Glimmer, S. Lawn, B. Dinkel, and A. Mitwalsky. 1999. Reliability of laser activated metal fuses in DRAMs. In *Twenty Fourth IEEE/CPMT International Electronics Manufacturing Technology Symposium*. 389–394.

[6] Kenneth E. Batcher. 1980. Architecture of a Massively Parallel Processor. In *Proceedings of the 7th Annual Symposium on Computer Architecture (ISCA '80)*. ACM, New York, NY, USA, 168–173.

[7] Kenneth E. Batcher. 1980. Design of a Massively Parallel Processor. *IEEE Trans. Comput.* C-29, 9 (Sept 1980), 836–840.

[8] Kenneth E. Batcher. 1982. Bit-serial parallel processing systems. *IEEE Trans. Comput.* C-31, 5 (May 1982), 377–384.

[9] Kenneth E. Batcher. 1998. Retrospective: architecture of a massively parallel processor. In *25 Years of the International Symposia on Computer Architecture (Selected Papers) (ISCA '98)*. ACM, New York, NY, USA, 15–16.

[10] K. E. Batcher, E. E. Eddey, R. O. Faiss, and P. A. Gilmore. 1981. *SAR processing on the MPP*. Technical Report NASA-CR-166726 (N82-11801/9). Goodyear Aerospace Corporation.

[11] Adrian M Caulfield, Laura M Grupp, and Steven Swanson. 2009. Gordon: using flash memory to build fast, power-efficient clusters for data-intensive applications. *ACM Sigplan Notices* 44, 3 (2009), 217–228.

[12] Karthik Chandrasekar, Sven Goossens, Christian Weis, Martijn Koedam, Benny Akesson, Norbert Wehn, and Kees Goossens. 2014. Exploiting expendable process-margins in DRAMs for run-time performance optimization. In *Proceedings of the Conference on Design, Automation & Test in Europe (DATE '14)*. European Design and Automation Association, Article 173, 6 pages. http://dl.acm.org/citation.cfm?id=2616606.2616820

[13] Kevin K. Chang, Abhijith Kashyap, Hasan Hassan, Saugata Ghose, Kevin Hsieh, Donghyuk Lee, Tianshi Li, Gennady Pekhimenko, Samira Khan, and Onur Mutlu. 2016. Understanding latency variation in modern DRAM chips: experimental characterization, analysis, and optimization. *SIGMETRICS Perform. Eval. Rev.* 44, 1 (June 2016), 323–336.

[14] Meng-Fan Chang, Shin-Jang Shen, Chia-Chi Liu, Che-Wei Wu, Yu-Fan Lin, Ya-Chin King, Chorng-Jung Lin, Hung-Jen Liao, Yu-Der Chih, and Hiroyuki Yamauchi. 2013. An offset-tolerant fast-random-read current-sampling-based sense amplifier for small-cell-current nonvolatile memory. *IEEE Journal of Solid-State Circuits* 48, 3 (2013), 864–877.

[15] Jan Craninckx and Geert Van der Plas. 2007. A 65fJ/conversion-step 0-to-50MS/s 0-to-0.7 mW 9b charge-sharing SAR ADC in 90nm digital CMOS. In *2007 IEEE International Solid-State Circuits Conference. Digest of Technical Papers*. IEEE, 246–600.

[16] G. De Sandre, L. Bettini, A. Pirola, L. Marmonier, M. Pasotti, M. Borghi, P. Mattavelli, P. Zuliani, L. Scotti, G. Mastracchio, F. Bedeschi, R. Gastaldi, and R. Bez. 2010. A 90nm 4Mb embedded phase-change memory with 1.2V 12ns read access time and 1MB/s write throughput. In *2010 IEEE International Solid-State Circuits Conference - (ISSCC)*. 268–269.

[17] Denyer, Renshaw Peter B., and David. 1985. *VLSI signal processing: a bit-serial approach*. Addison-Wesley Publishing Company, San Francisco, CA, USA.

[18] P. Dlugosch, D. Brown, P. Glendenning, M. Leventhal, and H. Noyes. 2014. An efficient and scalable semiconductor architecture for parallel automata processing. *IEEE Transactions on Parallel and Distributed Systems* 25, 12 (Dec 2014), 3088–3098.

[19] Jeff Draper, Jacqueline Chame, Mary Hall, Craig Steele, Tim Barrett, Jeff LaCoss, John Granacki, Jaewook Shin, Chun Chen, Chang Woo Kang, Ihn Kim, and Gokhan Daglikoca. 2002. The architecture of the DIVA processing-in-memory chip. In *Proceedings of the 16th International Conference on Supercomputing (ICS '02)*. ACM, New York, NY, USA, 14–25.

[20] Michael Drumheller. 1986. Connection Machine Stereomatching. In *Proceedings of the Fifth AAAI National Conference on Artificial Intelligence (AAAI'86)*. AAAI Press, 748–753.

[21] C. Eckert, X. Wang, J. Wang, A. Subramaniyan, R. Iyer, D. Sylvester, D. Blaaauw, and R. Das. 2018. Neural Cache: bit-serial in-cache acceleration of deep neural networks. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. 383–396.

[22] Duncan Elliott, Michael Stumm, W. Martin Snelgrove, Christian Cojocaru, and Robert McKenzie. 1999. Computational RAM: implementing processors in memory. *IEEE Design & Test of Computers* 1 (1999), 32–41.

[23] Duncan G. Elliott, W. Martin Snelgrove, and Michael Stumm. 1992. Computational RAM: a memory-SIMD hybrid and its application to DSP. In *1992 Proceedings of the IEEE Custom Integrated Circuits Conference*. 30.6.1–30.6.4.

[24] T. Finkbeiner, G. Hush, T. Larsen, P. Lea, J. Leidel, and T. Manning. 2017. In-memory intelligence. *IEEE Micro* 37, 4 (2017), 30–38.

[25] M. Gao, G. Ayers, and C. Kozyrakis. 2015. Practical near-data processing for in-memory analytics frameworks. In *2015 International Conference on Parallel Architecture and Compilation (PACT)*. 113–124.

[26] Saugata Ghose, Abdullah Giray Yaglikçi, Raghav Gupta, Donghyuk Lee, Kais Kudrolli, William X. Liu, Hasan Hassan, Kevin K. Chang, Niladrish Chatterjee, Aditya Agrawal, Mike O'Connor, and Onur Mutlu. 2018. What your DRAM power models are not telling you: lessons from a detailed experimental study. In *Abstracts of the 2018 ACM International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '18)*. ACM, New York, NY, USA, 110–110.

[27] M. Gokhale, B. Holmes, and K. Iobst. 1995. Processing in memory: the Terasys massively parallel PIM array. *Computer* 28, 4 (April 1995), 23–31.

[28] Qing Guo, Xiaochen Guo, Ravi Patel, Engin Ipek, and Eby G. Friedman. 2013. AC-DIMM: associative computing with STT-MRAM. *SIGARCH Comput. Archit. News* 41, 3 (June 2013), 189–200.

[29] Anthony Gutierrez, Michael Cieslak, Bharan Giridhar, Ronald G Dreslinski, Luis Ceze, and Trevor Mudge. 2014. Integrated 3D-stacked server designs for increasing physical density of key-value stores. In *ACM SIGPLAN Notices*, Vol. 49. ACM, 485–498.

[30] H. Hassan, N. Vijaykumar, S. Khan, S. Ghose, K. Chang, G. Pekhimenko, D. Lee, O. Ergin, and O. Mutlu. 2017. SoftMC: a flexible and practical open-source infrastructure for enabling experimental DRAM studies. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 241–252.

[31] Hybrid Memory Cube Consortium. 2014. Hybrid Memory Cube specification 2.1.

[32] Bruce Jacob, Spencer Ng, and David Wang. 2007. *Memory systems: cache, DRAM, disk.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[33] S. Jain, A. Ranjan, K. Roy, and A. Raghunathan. 2018. Computing in memory with spin-transfer torque magnetic RAM. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 26, 3 (March 2018), 470–483.

[34] Yi Kang, Wei Huang, Seung-Moon Yoo, D. Keen, Zhenzhou Ge, V. Lam, P. Pattnaik, and J. Torrellas. 1999. FlexRAM: toward an advanced intelligent memory system. In *Proceedings 1999 IEEE International Conference on Computer Design: VLSI in Computers and Processors*. 192–201.

[35] Duckhwan Kim, Jaeha Kung, Sek Chai, Sudhakar Yalamanchili, and Saibal Mukhopadhyay. 2016. Neurocube: a programmable digital neuromorphic architecture with high-density 3D memory. In *Computer Architecture (ISCA), 2016 ACM/IEEE 43rd Annual International Symposium on*. IEEE, 380–392.

[36] Jeremie S Kim, Minesh Patel, Hasan Hassan, and Onur Mutlu. 2018. The DRAM latency PUF: quickly evaluating physical unclonable functions by exploiting the latency-reliability tradeoff in modern commodity DRAM devices. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 194–207.

[37] Yoongu Kim, Vivek Seshadri, Donghyuk Lee, Jamie Liu, and Onur Mutlu. 2012. A case for exploiting subarray-level parallelism (SALP) in DRAM. *ACM SIGARCH Computer Architecture News* 40, 3 (2012), 368–379.

[38] DBM Klaassen. 1992. A unified mobility model for device simulation-II. Temperature dependence of carrier mobility and lifetime. *Solid-State Electronics* 35, 7 (1992), 961–967.

[39] P. M. Kogge. 1994. EXECUBE - A new architecture for scaleable MPPs. In *1994 International Conference on Parallel Processing Vol. 1*, Vol. 1. 77–84.

[40] S. Kvatinsky, D. Belousov, S. Liman, G. Satat, N. Wald, E. G. Friedman, A. Kolodny, and U. C. Weiser. 2014. MAGIC - memristor-aided logic. *IEEE Transactions on Circuits and Systems II: Express Briefs* 61, 11 (Nov 2014), 895–899.

[41] S. Kvatinsky, A. Kolodny, U. C. Weiser, and E. G. Friedman. 2011. Memristor-based IMPLY logic design procedure. In *2011 IEEE 29th International Conference on Computer Design (ICCD)*. 142–147.

[42] Perry V. Lea and Richard C. Murphy. 2018. Apparatuses and methods for in-memory operations. US Patent 10049721B1.

[43] Donghyuk Lee, Samira Khan, Lavanya Subramanian, Saugata Ghose, Rachata Ausavarungnirun, Gennady Pekhimenko, Vivek Seshadri, and Onur Mutlu. 2017. Design-induced latency variation in modern DRAM chips: characterization, analysis, and latency reduction mechanisms. *Proceedings of the ACM on Measurement and Analysis of Computing Systems* 1, 1 (2017), 26.

[44] D. Lee, Y. Kim, G. Pekhimenko, S. Khan, V. Seshadri, K. Chang, and O. Mutlu. 2015. Adaptive-latency DRAM: optimizing DRAM timing for the common-case. In *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*. 489–501.

[45] Shuangchen Li, Dimin Niu, Krishna T Malladi, Hongzhong Zheng, Bob Brennan, and Yuan Xie. 2017. DRISA: A DRAM-based reconfigurable in-situ accelerator. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, 288–301.

[46] Shuangchen Li, Cong Xu, Qiaosha Zou, Jishen Zhao, Yu Lu, and Yuan Xie. 2016. Pinatubo: a processing-in-memory architecture for bulk bitwise operations in emerging non-volatile memories. In *Proceedings of the 53rd Annual Design Automation Conference*. ACM, 173.

[47] Jamie Liu, Ben Jaiyen, Yoongu Kim, Chris Wilkerson, and Onur Mutlu. 2013. An experimental study of data retention behavior in modern DRAM devices: implications for retention time profiling mechanisms. In *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA '13)*. ACM, New York, NY, USA, 60–71.

[48] Sally A. McKee. 2004. Reflections on the Memory Wall. In *Proceedings of the 1st Conference on Computing Frontiers (CF '04)*. ACM, New York, NY, USA, 162–167.

[49] Câncio Monteiro, Yasuhiro Takahashi, and Toshikazu Sekine. 2013. Charge-sharing symmetric adiabatic logic in countermeasure against power analysis attacks at cell level. *Microelectronics Journal* 44, 6 (2013), 496–503.

[50] R. Nair, S. F. Antao, C. Bertolli, P. Bose, J. R. Brunheroto, T. Chen, C. . Cher, C. H. A. Costa, J. Doi, C. Evangelinos, B. M. Fleischer, T. W. Fox, D. S. Gallo, L. Grinberg, J. A. Gunnels, A. C. Jacob, P. Jacob, H. M. Jacobson, T. Karkhanis, C. Kim, J. H. Moreno, J. K. O'Brien, M. Ohmacht, Y. Park, D. A. Prener, B. S. Rosenburg, K. D. Ryu, O. Sallenave, M. J. Serrano, P. D. M. Siegl, K. Sugavanam, and Z. Sura. 2015. Active Memory Cube: a processing-in-memory architecture for exascale systems. *IBM Journal of Research and Development* 59, 2/3 (March 2015), 17:1–17:14.

[51] M. Oskin, F. T. Chong, and T. Sherwood. 1998. Active Pages: a computation model for intelligent memory. In *Proceedings. 25th Annual International Symposium on Computer Architecture*. 192–203.

[52] D. Patterson, T. Anderson, N. Cardwell, R. Fromm, K. Keeton, C. Kozyrakis, R. Thomas, and K. Yelick. 1997. A case for intelligent RAM. *IEEE Micro* 17, 2 (March 1997), 34–44.

[53] JL Potter. 1983. Image processing on the massively parallel processor. *Computer* 16, 1 (Jan 1983), 62–67.

[54] Jan M Rabaey, Anantha P Chandrakasan, and Borivoje Nikolic. 2002. *Digital integrated circuits*. Vol. 2. Prentice Hall Englewood Cliffs.

[55] Parthasarathy Ranganathan. 2011. From microprocessors to nanostores: rethinking data-centric systems. *Computer* 44, 1 (2011), 39–48.

[56] 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. 2013. RowClone: fast and energy-efficient in-DRAM bulk data copy and initialization. In *2013 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 185–197.

[57] 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 the 50th Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, 273–287.

[58] Tadashi Shibata and Tadahiro Ohmi. 1992. A functional MOS transistor featuring gate-level weighted sum and threshold operations. *IEEE Transactions on Electron devices* 39, 6 (1992), 1444–1455.

[59] H. S. Stone. 1970. A logic-in-memory computer. *IEEE Trans. Comput.* C-19, 1 (Jan 1970), 73–78.

[60] K. Tsuchida, T. Inaba, K. Fujita, Y. Ueda, T. Shimizu, Y. Asao, T. Kajiyama, M. Iwayama, K. Sugiura, S. Ikegawa, T. Kishi, T. Kai, M. Amano, N. Shimomura, H. Yoda, and Y. Watanabe. 2010. A 64Mb MRAM with clamped-reference and adequate-reference schemes. In *2010 IEEE International Solid-State Circuits Conference - (ISSCC)*. 258–259.

[61] L. W. Tucker and G. G. Robertson. 1988. Architecture and applications of the Connection Machine. *Computer* 21, 8 (Aug 1988), 26–38.

[62] Wm. A. Wulf and Sally A. McKee. 1995. Hitting the Memory Wall: implications of the obvious. *SIGARCH Comput. Archit. News* 23, 1 (March 1995), 20–24.

[63] Xilinx. 2011. Virtex-6 FPGA Memory Interface Solutions. https://www.xilinx.com/support/documentation/ip_documentation/ug406.pdf. Accessed: 2019-08-30.

[64] Xilinx. 2019. ML605 Hardware User Guide. https://www.xilinx.com/support/documentation/boards_and_kits/ug534.pdf. Accessed: 2019-08-30.

[65] Kewei Yang and Andreas G Andreou. 1994. A multiple input differential amplifier based on charge sharing on a floating-gate MOSFET. *Analog Integrated Circuits and Signal Processing* 6, 3 (1994), 197–208.