# ParaBit: Processing Parallel Bitwise Operations in NAND Flash Memory based SSDs

Congming Gao[†] Xin Xin[*] Youyou Lu[†] Youtao Zhang[§] Jun Yang[*] Jiwu Shu[†]
Department of Computer Science and Technology / BNRist, Tsinghua University[†], China
Department of Electrical and Computer Engineering, University of Pittsburgh[*], USA
Department of Computer Science, University of Pittsburgh[§], USA
shujw@tsinghua.edu.cn

## ABSTRACT

Processing-in-memory (PIM) and in-storage-computing (ISC) architectures have been constructed to implement computation inside memory and near storage, respectively. While effectively mitigating the overhead of data movement from memory and storage to the processor, due to the limited bandwidth of existing systems, these architectures still suffer from the large data movement overhead between storage and memory, in particular, if the amount of required data is large. It has become a major constraint for further improving the computation efficiency in PIM and ISC architectures.

In this paper, we propose ParaBit, a scheme that enables Parallel Bitwise operations in NAND flash storage where data reside. By adjusting the latching circuit control and the sequence of sensing operations, ParaBit enables in-flash bitwise operation with no or little extra hardware, which effectively reduces the overhead of data movement between storage and memory. We exploit the massive parallelism in NAND flash based SSDs to mitigate the long latency of flash operations. Our experimental results show that the proposed ParaBit design achieves significant performance improvements over the state-of-the-art PIM and ISC architectures.

## CCS CONCEPTS

• **Hardware** → **External storage**; • **Computer systems organization** → *Architectures.*

## KEYWORDS

flash memory, in-storage computing, near data processing, bitwise operation

**ACM Reference Format:**
Congming Gao[†] Xin Xin[*] Youyou Lu[†] Youtao Zhang[§] Jun Yang[*] Jiwu Shu[†]. 2021. ParaBit: Processing Parallel Bitwise Operations in NAND Flash Memory based SSDs. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '21), October 18–22, 2021, Virtual Event, Greece.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3466752.3480078


*MICRO '21, October 18–22, 2021, Virtual Event, Greece*

ACM ISBN 978-1-4503-8557-2/21/10…$15.00
https://doi.org/10.1145/3466752.3480078

## 1 INTRODUCTION

Memory wall remains a major performance bottleneck of modern computing-centric systems [36]. Recent studies have proposed two types of solutions — processing-in-memory (PIM) and in-storage computing (ISC), to address the large data movement overhead between memory/storage and processor. By exploiting the computation capability and high bandwidth of DRAM and NVM memory technologies, PIM architectures integrate basic logic computation in memory and offload computation to memory, which achieves great bandwidth and performance improvement with low design complexity and overhead [2, 47, 49, 56]. Alternatively, ISC architectures [43, 48] also offload computation to processors inside SSD controller, near storage, which mitigates the data movement overhead from storage to the main processor.

Most existing PIM and ISC techniques assume the data has been loaded into memory before processing such that the data swapping between memory and storage is negligible. However, this assumption is too optimistic: due to the limited memory size, data-intensive applications often have to store data in storage initially, and move data from storage to memory before and during the processing. Our preliminary study shows that PIM and ISC schemes exhibit 30× and 60× data movement cost over bitwise operation cost while the interconnection interface exhibits limited bandwidth, respectively. Therefore, it is beneficial to enable computation inside storage (e.g., SSD) where data resides, thus reducing or avoiding significant data movement from storage to memory.

To mitigate the data movement overhead between memory and storage, we propose ParaBit, a scheme that conducts Parallel Bitwise operations in NAND flash memory based SSDs. ParaBit exploits the intrinsic computation capability of the control circuits in commodity SSD so that it can be implemented with firmware upgrade, i.e., **no extra hardware**. ParaBit adjusts the sensing and transistor controls to perform bitwise operations in latching circuit during read operation. ParaBit supports seven common bitwise operations (including `AND`, `OR`, `XNOR`, `NAND`, `NOR`, `XOR` and `NOT`) in NAND flash based SSDs. While each bitwise operation in ParaBit takes longer latency than that in PIM and ISC architectures, ParaBit exploits massive parallelism of SSD or high scalability in all-flash array and strives to achieve the overall efficiency. We then propose location-free ParaBit to further improve the performance. Location-free ParaBit facilitates bitwise operations with operands resided in flash cells from different wordlines by integrating an extra inverter and two control transistors to assist `XOR` and `XNOR` operations. To the best of our knowledge, ParaBit is the first work that leverages the intrinsic computation capability of flash memory's circuits to realize in-SSD data processing. We implement the ParaBit schemes

and evaluate them in three case studies. Our experimental results show that ParaBit schemes, by performing computations inside SSDs, significantly reduce the data movement overhead and thus improve the system performance.

We summarized our contributions as follows. 1) We proposed an in-flash-computing scheme, referred to as ParaBit, to realize bitwise operations in NAND flash memory with no or little hardware modification; 2) We constructed a ParaBit based SSD design, which includes several additional components for implementing ParaBit inside SSDs; 3) We evaluated the performance of ParaBit in MLC SSD using a modified SSD simulator and compared it with state-of-the-art PIM and ISC architectures. The reliability of ParaBit also is verified in real flash chips.

In the rest of the paper, we briefly discuss the background and related works in Section 2. We present the motivation in Section 3 and elaborate our design in Section 4. We analyze the experimental results in Section 5 and get the conclusion in Section 6.

## 2 BACKGROUND AND RELATED WORK

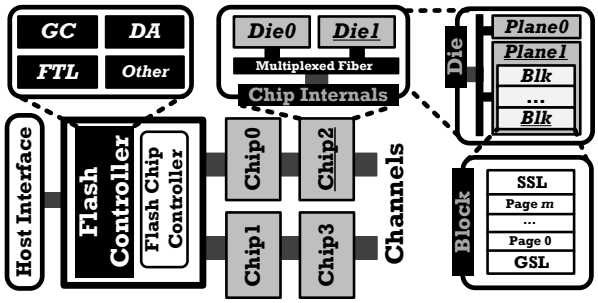
### 2.1 NAND Flash Memory based SSDs

Figure 1(a) shows the architecture of a typical planar MLC based SSD. Inside SSD, there is a host interface, which is used to connect SSD and host system. The flash controller contains several important components, which are in charge of Garbage Collection (GC), Data Allocation (DA), address mapping (Flash Translation Layer, FTL), and some other functions, such as Wear Leveling (WL) and cache [1]. Apart from these components, a flash chip controller is equipped to connect flash chips and flash controller. To boost the performance of SSDs, internal architecture is organized in four levels of parallelism, from channel to chip, to die, and to plane [15, 16]. Inside each die, multiple planes are maintained and each plane contains a series of blocks. Each block contains several pages.

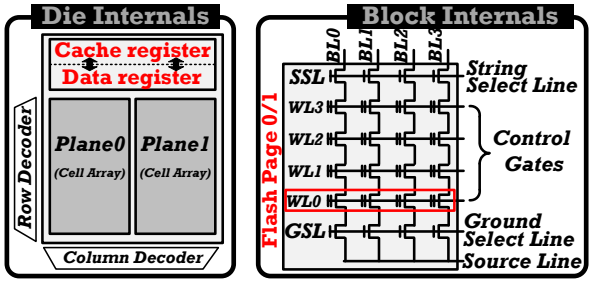
Figure 1(b) show the internal details of die and block. Inside a die, two planes are engineered. Around planes, there are two address decoders, determining the data access location. On the top of planes, two registers, termed cache register and data register, are engineered to transfer data to or from flash array. To speed up read operation, data can be transferred from cache register to host while concurrently moving data from flash array to data register. This process is termed Cache Read, which has been widely supported in modern flash memory [23, 39]. Inside the block, all flash cells are organized as cell array. In the right of Figure 1(b), all cells in horizontal direction are connected through a wordline (WL) and all cells in vertical direction are stringed and selected by bitline (BL). According to the number of bits stored in a cell, flash cells can be divided into several types: Single Level Cell (SLC), Multi Level Cell (MLC), Triple Level Cell (TLC) and Quad Level Cell (QLC). Take Figure 1(b) as an example, where each MLC cell stores a 2-bit value, each bit of which belongs to different page. All MLC cells reside in the same WL are grouped and used as two physical pages. Inside SSDs, read and write are performed in page-based operation while erase is processed in block-based operation.

### 2.2 Latching Circuit in Flash Memory

Given ParaBit relies on adjusting the control sequence of latching circuit, we next elaborate the baseline implementation in this




(a) The architecture of SSD.




(b) The internal details about die and block.

**Figure 1: The architecture of SSD and its internals.**

section. As shown in Figure 2, data register and cache register are implemented via two latches, which can buffer data derived from sense amplifiers while outputting the cached data. In the discussion, **we represent the logic value at a location $X$ as $L(X)$=$x_1x_2x_3x_4$, where $x_1$, $x_2$, $x_3$, and $x_4$ indicate the logic value when the MLC cell being sensed is in state** E, S1, S2, **and** S3, **respectively**. Each $x_i$(1<=i<=4) takes 0 or 1 indicating if location $X$ has low or high voltage, respectively.

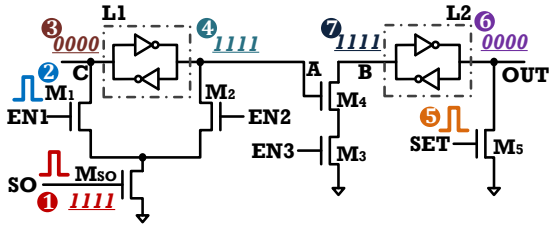



**Figure 2: Initialization of latching circuit.**

Figure 2 illustrates how we initialize the latch circuit before read operation. SO and EN1 signals are initially set to "high" (step 1 and step 2) and transistors $M_{SO}$ and $M_1$ are enabled to force the voltage at C to ground so that $L(C)$=0000 (step 3). Consequently, $L(A)$=$\overline{L(C)}$=1111 (step 4). Then, SET signal is set to "high" (step 5) so that $L(OUT)$=0000 (step 6) and $L(B)$=$\overline{L(OUT)}$=1111 (step 7).

To differentiate four states in an MLC cell (E, S1, S2 and S3), three reference voltages ($V_{READ1}$, $V_{READ2}$ and $V_{READ3}$) are used as threshold voltages, as shown in the middle of Figure 3. By applying different sensing voltages in a sequence of Single Read Operations (SROs), we get the outputs by comparing the sensing voltages and the current cell voltage. The comparison result is transferred to SO node where the value may be 0 (if the cell voltage is lower than sensing voltage) or 1 (if the cell voltage is higher). The value at SO is then moved and stored in the latches (L1 and L2). Since the sensing circuit is independent of L2, SROs can be performed to

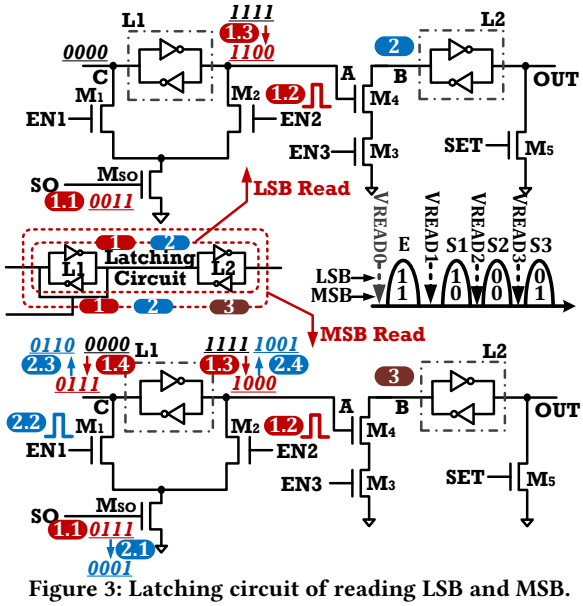



**Figure 3: Latching circuit of reading LSB and MSB.**

sense and store new value in L1 while previous sensed value has been buffered in L2 and then be transferred out from node OUT (referred to as "Cache Read").

To read the LSB bit (least-significant-bit) from an MLC cell, the controller applies a sensing voltage $V_{READ2}$, denoted as step 1 at the top of Figure 3, which includes three sub steps. Depending on the state of the cell, all possible outputs at SO are represented as $L(SO)$=0011 (step 1.1) while the voltage of cell in E and S1 states is lower than $V_{READ2}$ and voltage of cell in S2 and S3 states is higher. The transistor $M_2$ is then enabled (step 1.2) and voltage at A is forced to ground if SO has a sensed high voltage, e.g., $L(A)$ changes from 1 to 0 only when $L(SO)$=1 at step 1.3. As a result, $L(A)$=$L(A)_{old} \wedge \overline{L(SO)}$=1100, i.e., it is the same as the LSB bit value. Step 2 transfers the sensed value to L2 to enable cache read.

The process to read MSB bit (most-significant-bit) (at the bottom of Figure 3) is similar. The difference is that we employ two sensing voltages $V_{READ1}$ and $V_{READ3}$ to sense data to L1. At step 1.1, $L(SO)$=0111 when sensing voltage is applied at $V_{READ1}$. At step 1.2, we then enable $M_2$ and force the voltage at A to ground if $L(SO)$=1. As a result, $L(A)$=$L(A)_{old} \wedge \overline{L(SO)}$=1000 and $L(C)$=$\overline{L(A)}$=0111 at step 1.3 and step 1.4. After finishing step 1, $M_2$ is disabled. At step 2.1, sensing voltage $V_{READ3}$ is applied and $L(SO)$=0001. Similarly, we enable $M_1$ (step 2.2) so that $L(C)$=$L(C)_{old} \wedge \overline{L(SO)}$=0110 (step 2.3) and $L(A)$=$\overline{L(C)}$=1001 (step 2.4), i.e., it is the same as the MSB bit. Step 3 transfers the sensed value to L2 to enable cache read.

## 2.3 PIM and ISC

**PIM:** PIM typically exploits the intrinsic computation capability inside DRAM or NVM technologies, which mitigate the data movement overhead between memory and processor [41, 49, 56]. For example, Ambit is a typical DRAM based PIM architecture that exploits charge sharing for computation [49]. For example, to implement AND operation in DRAM, Ambit activates three cells on the same bitline so that, if at least two cells' voltages are high, the voltage on the bitline is above $V_{dd}/2$ after charge sharing. Otherwise, the voltage is lower than $V_{dd}/2$. The NOT operation in Ambit is realized by attaching a dual-contact cell to the bitline so that the negated value of the bitline's output is sensed. Given the parallel DRAM cells in a row, bulk bitwise operations can be realized.

PIM architectures usually assume that all operands have been loaded into the memory. The data movement between storage and memory is rarely studied in the literature.

**ISC:** ISC architectures were designed to offload computation to the integrated computing units that reside in storage modules. By sending only the results back to the processor, ISC effectively mitigates the data movement between storage and the main processor [11, 31, 43, 48, 50]. Previous works adopt FPGA or embedded ARM to implement computations so as to achieve good tradeoff between programmability and computing capability, which potentially can meet different computation requirements. While ISC architectures can implement bulk bitwise operations, they face two shortcomings: First, processing bitwise operation in an powerful computing unit is not cost-effective. Take FPGA as an example, SRAM-based FPGA requires additional power consumption to maintain operands in SRAM, compromising the energy efficiency. Second, due to the limited memory capacity in SSD controller and low interconnection bandwidth, the data movement overhead remains high.

Instead, this work exploits a new computing paradigm that offloads basic bitwise operations to flash memory and then further reduce data movement overhead to and from the storage.

## 2.4 Other Related Works

In addition to DRAM-based PIMs [18, 49, 56], recent studies enabled bitwise operations in different memory technologies, e.g., SRAM, RRAM and MRAM. These works leverage electrical characteristics [10, 33, 53] or stateful logic operation [5, 55, 58] to conduct simple bitwise operations.

However, due to the poor density, high cost and immature process technology, these new memory medium based PIM architectures are still a long way from widespread use. For NDP (Near Data Processing), Akin *et al.* proposed to place processing units near main memory so that data can be processed without high data movement cost [3]. NDP often introduces higher design overhead, due to the demands for high-speed processor, more memory controllers and I/O circuits [4]. For ISC, powerful computing units are included in high-end SSDs to process the data before being moved to the memory [11, 31, 48]. Although ISC achieves good performance by bringing data near storage, designing a bitwise operation dedicated ISC is not cost-effective.

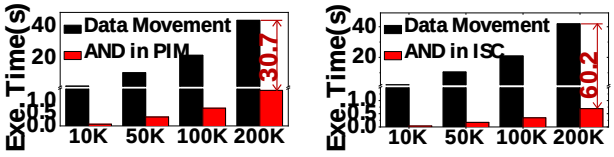
# 3 MOTIVATION

We motivate our design by studying the data movement between SSD and memory in PIM and ISC architectures, respectively. We choose an image segmentation benchmark program [14] as the example, which employs color recognition to realize region segmentation. To determine whether one pixel belongs to a color range, the program uses YUV color recognition and leverages bitwise operations [6]. Assume the YUV color space is discretized to 10 levels in each channel, the "orange" color can then be represented as:

```
Y_Class[ ]={0,1,1,1,1,1,1,1,1,1};
U_Class[ ]={0,0,0,0,0,0,0,1,1,1};
V_Class[ ]={0,0,0,0,0,0,0,1,1,1}.
```

Given a pixel whose YUV value is (2,7,9), it falls into orange's color space if "`Y_Class[2] AND U_Class[7] AND V_Class[9]`" is evaluated to be true. The color recognition can be concurrently performed with multiple pixels and colors. In the experiment, the number of images varies from 10,000 to 200,000, the resolution of each image is set to 800×600, the YUV color space is discretized to 256 levels in each channel, and four types of colors are recognized. Due to the limited memory size, image data need to be stored in storage before processing — it takes 267.5GB space to store 200,000 images with each image occupying 1.37MB space. When writing well-reconstructed images to storage, the program performs a light-weight pre-processing that transforms YUV value into recognized color based YUV classes, which reduces the per image space consumption to 0.72MB (800×600×3 channels×4 bits per channel), or 140GB for 200,000 images. Clearly, this data volume is still beyond the memory capacity of a typical PIM/ISC system.

To evaluate the data movement overhead, we choose the Ambit PIM architecture [49] with its latency of bitwise operation measured in H-spice[51], and Cosmos OpenSSD platform [13] as ISC architecture, respectively. For the storage device, we use a Samsung SSD 970 PRO with MLC flash memory and 512GB capacity [46]. Given current Cosmos OpenSSD platform's bandwidth is out-dated (about 2GB/s while modern SSDs' bandwidth is often greater than 3GB/s), we attach Samsung SSD 970 PRO to the Cosmos OpenSSD platform for moving data from flash memory to computing unit. We remap all images' logical addresses sequentially so as to exploit SSD's peak bandwidth [27]. More details about the evaluation can be found in Section 5. Figure 4 summarizes the results. While it actually triggers data swap between memory and SSD if the size of data operands exceeds the DRAM or SRAM capacity in PIM and ISC, respectively, we ignore result writeback and thus only account for the time spent in moving data from SSD.




(a) The evaluation with PIM. (b) The evaluation with ISC.

**Figure 4: The execution time of data movement and bitwise operations in PIM and ISC architectures.**

From the figure, both PIM and ISC spend most of their execution time on data movement while only a small portion of their execution time on bitwise `AND` operations. PIM and ISC spend 43.9$s$ and 41.8$s$, respectively, on data movement, which are 30.7× and 60.2× longer than the time on `AND` operations, respectively. In the experiment, the PCIe Gen3× 4 is used as the interconnection but remains a performance bottleneck. To summarize, it can greatly improve the system performance if the computation can be offloaded to SSD so that only the computation results, i.e., a small amount of data, need to be transferred from SSD to DRAM and/or FPGA.

# 4 THE DESIGN OF PARABIT

As a solution to address the issue in the preceding section, we propose to offload bitwise operations to SSD. We first present how to perform such operations in NAND flash based SSDs, and then elaborate the implementation details, followed by additional discussion.

## 4.1 Performing Bitwise in Latching Circuit

Intuitively, our latching circuit based bitwise operation is to compute, e.g., $Z_i$ = $X_i$ `AND` $Y_i$ (i=0,1,2...), by first storing $X_i$ and $Y_i$ in the LSB and MSB of the same MLC cell, and then having $Z_i$ computed and buffered in the output latch (i.e., `L2` latch in Figure 2). For clarity, the truth table for the set of bitwise operations supported by ParaBit is shown in Table 1.

**Table 1: Truth table of bitwise operation.**

| *State* | *(LSB/MSB)* | *AND* | *OR* | *XNOR* | *NAND* | *NOR* | *XOR* | *NOT* |
|---|---|---|---|---|---|---|---|---|
| ***E*** | ***(1/1)*** | 1 | 1 | 1 | 0 | 0 | 0 | (0/0) |
| ***S1*** | ***(1/0)*** | 0 | 1 | 0 | 1 | 0 | 1 | (0/1) |
| ***S2*** | ***(0/0)*** | 0 | 0 | 1 | 1 | 1 | 0 | (1/1) |
| ***S3*** | ***(0/1)*** | 0 | 1 | 0 | 1 | 0 | 1 | (1/0) |

**To conduct an `AND` operation,** the output should be 1000 according to the truth table and the state value notion used in Section 2.2, i.e., the output is 1, 0, 0, and 0 when the MLC to be sensed is in `E`, `S1`, `S2`, `S3` state, respectively. Only when the cell is in `E` state indicating both LSB and MSB are 1s, we have the output being 1.

As donated in Figure 5(a), `AND` operation can be achieved by employing two-step latching circuit control. At first step, we determine whether the cell being sensed is in state `E` by applying the sensing voltage at $V_{READ1}$ and having $L(SO)$=0111 at step 1.1. Then, at step 1.2, $M_2$ is enabled to force node `A` to ground if `SO` has a sensed high voltage. We have $L(A)$=$L(A)_{old} \wedge \overline{L(SO)}$=1000 (step 1.3), where $L(A)_{old}$ is the initialized value 1111. For the second step, the value of $L(A)$ is transferred to `L2` by enabling $M_3$ (step 2.1) and then $L(OUT)$=$\overline{L(B)}$=1000 (step 2.3).

The process of performing `AND` operation is the same as reading LSB bit illustrated in Figure 3, except for the applied sensing voltage. That is, `AND` operation can be realized inside flash memory based SSDs by leveraging the intrinsic bit read capability without extra hardware modification.

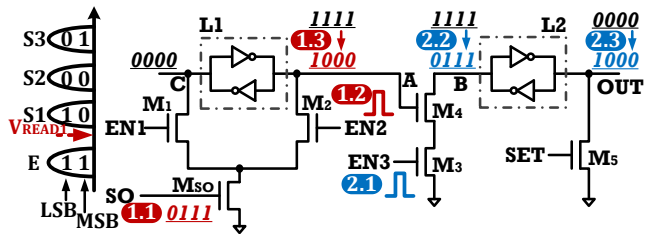
**To conduct an `OR` operation,** the output should be 1101. We take the three-step latching circuit control, which is the same as reading MSB bit except for the applied sensing voltages. As shown in Figure 5(b), at step 1.1, $L(SO)$=0011 while sensing voltage is applied at $V_{READ2}$, and then we have $L(A)$=$L(A)_{old} \wedge \overline{L(SO)}$=1100 (step 1.3) and $L(C)$=$\overline{L(A)}$=0011 (step 1.4). At step 2.1, sensing voltage is applied at $V_{READ3}$ to have $L(SO)$=0001. Then, by enabling $M_1$ (step 2.2), we have $L(C)$=$L(C)_{old} \wedge \overline{L(SO)}$=0010 (step 2.3), where $L(C)_{old}$=0011. Lastly, at step 2.4, $L(A)$=$\overline{L(C)}$=1101, i.e., it is the same as the result of `OR` operation. Again, step 3.1 enables $M_3$ to transfer the value to `L2` for cache read and $L(OUT)$=1101 (step 3.3).

**To conduct an `XNOR` operation,** the output should be 1010 based on the truth table. In Figure 6, we take a new six-step latching circuit control to determine whether the cell being sensed is in state `E` or `S2`. Six steps are elaborated as follows:

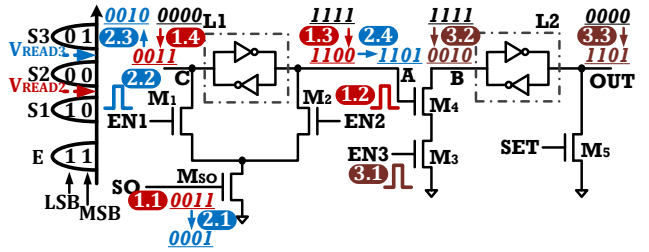
**Step 1.1** employs sensing voltage $V_{READ1}$ (step 1.1) and enables $M_2$ (step 1.2) so that $L(A)$=1000 (step 1.3) and $L(C)$=0111 (step 1.4);

**Step 2.1** enables $M_3$ to transfer the value in the preceding step to `OUT`, i.e., $L(OUT)$=$\overline{L(B)}$=$\overline{L(B)_{old} \wedge \overline{L(A)}}$=1000;

**Step 3.1** employs sensing voltage $V_{READ0}$ and enables $M_2$ (step 3.2) to set $L(A)$=0000 (step 3.3) and have $L(C)$=1111 (step 3.4);




(a) Performing AND operation in latching circuit.



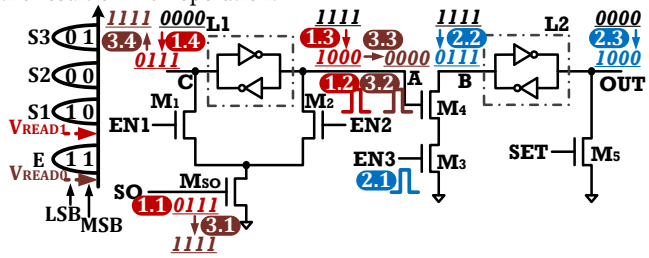

(b) Performing OR operation in latching circuit.

**Figure 5: Latching circuit control for performing AND and OR bitwise operations.**

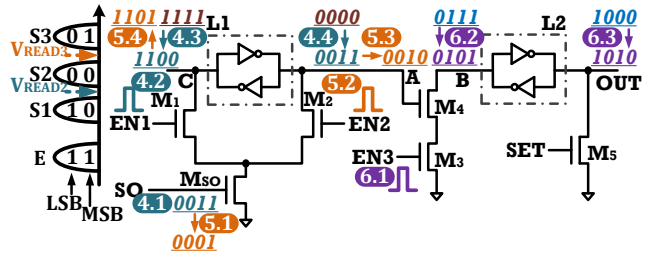
**Step 4.1** employs sensing voltage $V_{READ2}$ and enables $M_1$ (step 4.2) so that $L(C)$=$L(C)_{old} \wedge \overline{L(SO)}$=1111 $\wedge$ $\overline{0011}$=1100 (step 4.3) and $L(A)$=$\overline{L(C)}$ =0011 (step 4.4);

**Step 5.1** employs sensing voltage $V_{READ3}$ and enables $M_2$ (step 5.2) so that $L(SO)$=0001 and $L(A)$=$L(A)_{old} \wedge \overline{L(SO)}$=0011 $\wedge$ $\overline{0001}$ =0010 (step 5.3);

**Step 6.1** enables $M_3$ so that $L(B)$=$L(B)_{old} \wedge \overline{L(A)}$=0111$\wedge\overline{0010}$=0101 (step 6.2) and $L(OUT)$=$\overline{L(B)}$=1010 (step 6.3), e.g,. it is the same as the result of XNOR operation.




(a) The first, second and third steps of XNOR operation.




(b) The fourth, fifth and sixth steps of XNOR operation.

**Figure 6: Latching circuit control for performing XNOR bitwise operation.**

By adjusting the initial values, the sensing voltages, and the control transistors, we may implement other bitwise operations, e.g., NAND, NOR, XOR, and NOT, which are the inverted values of AND, OR, XNOR, and LSB/MSB value. Before performing these bitwise operations, values in latches should be re-initialized, of which the value in L2 is the same as traditional one while the value in L1 is inverted. In detail, as presented in Figure 7, SO and EN2 signals are initially set to "high" and transistors $M_{SO}$ and $M_1$ are enabled to force the voltage at A to ground so that $L(A)$=0000 and $L(C)$=$\overline{L(A)}$=1111. In the following discussion, we use the table representation to elaborate the implementation.

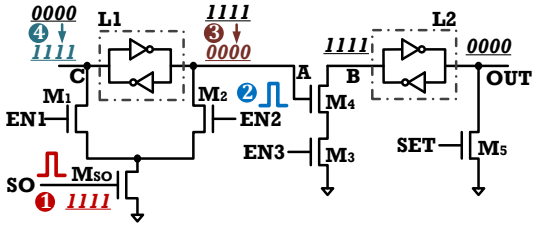



**Figure 7: Initialization of latching circuit for performing NAND, NOR, XOR and NOT bitwise operations.**

**For NAND operation,** the last three rows in Table 2 indicate we need one initialization, one sensing step and one transfer step. The second and third columns of each row elaborate the control details at each step. The second row indicates the initialization of latching circuit. The third row indicates we use sensing voltage $V_{READ1}$ and enable $M_1$, thus the values at node C and A are changed. The fourth row indicates we enable $M_3$ and transfer value from L1 to L2, and then we can have $L(OUT)$=$\overline{L(B)}$=0111, e.g., it is the same as the result of NAND operation.

**Table 2: Performing NAND operation in latching circuit.**

| *Rows* | NAND | *L(SO)* | $M_x$ | *L(C)* | *L(A)* | *L(B)* | *L(OUT)* |
|---|---|---|---|---|---|---|---|
| *1* | ***Initialization*** | - | - | 1111 | 0000 | 1111 | 0000 |
| *2* | $V_{READ1}$ | 0111 | $M_1$ | 1000 | 0111 | 1111 | 0000 |
| *3* | ***L1 to L2*** | - | $M_3$ | 1000 | 0111 | 1000 | 0111 |

**For NOR operation,** after initialization, we need two sensing steps and one transfer step as shown in Table 3. The third row indicates we use sensing voltage $V_{READ2}$ and enable $M_1$ to have $L(A)$=$\overline{L(C)}$=0011. The fourth row indicates we use sensing voltage $V_{READ3}$ and enable $M_2$ to conduct $L(A)$=0010. The last row indicates we enable $M_3$ and transfer value from L1 to L2, and then we have $L(OUT)$=$\overline{L(B)}$=0010, e.g., it is the same as the result of NOR operation.

**Table 3: Performing NOR operation in latching circuit.**

| *Rows* | NOR | *L(SO)* | $M_x$ | *L(C)* | *L(A)* | *L(B)* | *L(OUT)* |
|---|---|---|---|---|---|---|---|
| *1* | ***Initialization*** | - | - | 1111 | 0000 | 1111 | 0000 |
| *2* | $V_{READ2}$ | 0011 | $M_1$ | 1100 | 0011 | 1111 | 0000 |
| *3* | $V_{READ3}$ | 0001 | $M_2$ | 1101 | 0010 | 1111 | 0000 |
| *4* | ***L1 to L2*** | - | $M_3$ | 1101 | 0010 | 1101 | 0010 |

**For XOR operation,** we have $M \oplus N = \overline{M}N + M\overline{N}$. The control sequence is illustrated in Table 4. For $\overline{M}N$, it is true only when the cell being sensed is in state S3. Thus, the third row implements $\overline{M}N$ by using sensing voltage $V_{READ3}$. And then, sensed value is transferred and buffered at node OUT by enabling $M_3$. For $M\overline{N}$, we use sensing voltage $V_{READ1}$ and $V_{READ2}$ to determine whether the cell is in state S1. But before implementing $M\overline{N}$, the values at A and C should be re-initialized at the fifth row. And then, sixth and seventh rows indicate the implementation of $M\overline{N}$. After seventh row's control, we have $L(A)$=$M\overline{N}$. At the last step, we enable $M_3$ to transfer value from L1 to L2 so that we have $L(OUT)$= $\overline{L(B)}$=$L(B)_{old} \wedge \overline{L(A)_{old}}$ =$\overline{L(B)_{old}}$ + $L(A)_{old}$ =$L(OUT)_{old}$ + $L(A)_{old}$ =$\overline{M}N$ + $M\overline{N}$.

**Table 4: Performing XOR operation in latching circuit.**

| *Rows* | XOR | *L(SO)* | $M_x$ | *L(C)* | *L(A)* | *L(B)* | *L(OUT)* |
|---|---|---|---|---|---|---|---|
| *1* | ***Initialization*** | - | - | 1111 | 0000 | 1111 | 0000 |
| *2* | $V_{READ3}$ | 0001 | $M_1$ | 1110 | 0001 | 1111 | 0000 |
| *3* | ***L1 to L2*** | - | $M_3$ | 1110 | 0001 | 1110 | 0001 |
| *4* | $V_{READ0}$ | 1111 | $M_2$ | 1111 | 0000 | 1110 | 0001 |
| *5* | $V_{READ1}$ | 0111 | $M_1$ | 1000 | 0111 | 1110 | 0001 |
| *6* | $V_{READ2}$ | 0011 | $M_2$ | 1011 | 0100 | 1110 | 0001 |
| *7* | ***L1 to L2*** | - | $M_3$ | 1011 | 0100 | 1010 | 0101 |

**For NOT operation,** Table 5 lists the controls to invert the LSB and the MSB bits, respectively. The inversion is mainly achieved by setting different initial values at A and C.

**Table 5: Performing NOT operation in latching circuit.**

| *Rows* | NOT-LSB | *L(SO)* | $M_x$ | *L(C)* | *L(A)* | *L(B)* | *L(OUT)* |
|---|---|---|---|---|---|---|---|
| *1* | ***Initialization*** | - | - | 1111 | 0000 | 1111 | 0000 |
| *2* | $V_{READ2}$ | 0011 | $M_1$ | 1100 | 0011 | 1111 | 0000 |
| *3* | ***L1 to L2*** | - | $M_3$ | 1100 | 0011 | 1100 | 0011 |
| *Rows* | NOT-MSB | *L(SO)* | $M_x$ | *L(C)* | *L(A)* | *L(B)* | *L(OUT)* |
| *1* | ***Initialization*** | - | - | 1111 | 0000 | 1111 | 0000 |
| *2* | $V_{READ1}$ | 0111 | $M_1$ | 1000 | 0111 | 1111 | 0000 |
| *3* | $V_{READ3}$ | 0001 | $M_2$ | 1001 | 0110 | 1111 | 0000 |
| *4* | ***L1 to L2*** | - | $M_3$ | 1001 | 0110 | 1001 | 0110 |

## 4.2 Location-free ParaBit

The basic ParaBit works when two operand bits share the same MLC. When the operands are from different cells, ParaBit reallocates them to the same cell, which incurs high reallocation overhead and compromises the benefits of bitwise operations.

Given the aligned cells from the same bitline but different wordlines share the same latch circuit, it is possible to devise two step sensing strategy that senses one cell in the first step and an aligned cell in the second step. By adjusting the sensing voltage and latch circuit controls, we may have the bitwise operation of corresponding bits generated and buffered at the output latch of the latching circuit. This is referred to as location-free ParaBit in this paper. Location-free ParaBit eliminates bit reallocation such that it can effectively improve the in-storage computation performance.

To support location-free ParaBit, the intrinsic CACHE READ RANDOM command in NAND flash memory is leveraged, which can employ sensing voltages at random wordlines [23, 39]. In the following discussion, for two operand bits to be computed, we assume that one operand bit is stored in the LSB of one MLC cell while the other bit in the MSB of an aligned MLC cell (i.e., from the same bitline but different wordline).

**Table 6: AND operation in location-free ParaBit.**

| *LSB* | *L(SO)* | *L(A)* | *L(B)* | *L(OUT)* |
|---|---|---|---|---|
| *1* | *0* | 1001->1001 | 1111->0110 | 0000->1001 |
| *0* | *1* | 1001->0000 | 1111->1111 | 0000->0000 |

**To conduct an AND operation,** there are two steps after normal initialization. First, $V_{READ1}$ and $V_{READ3}$ are employed to perform MSB read operation, thus we can have $L(A)$=1001. Second, $V_{READ2}$ is employed to sense LSB bit. As elaborated in Table 6, if $L(SO)$=0, i.e., the value in LSB is 1, the output is determined by the sensed value at A, i.e., the value in MSB. Thus, $L(A)$=$\overline{L(SO)} \wedge L(A)_{old}$=$\overline{0} \wedge$ 1001=1001, which is the result of AND operation. After that the output is transferred to OUT. Otherwise, the output at OUT should be 0 while the value in LSB is 0. Thus, $L(A)$=$L(A)_{old} \wedge \overline{L(SO)}$=1001 $\wedge$ $\overline{1}$=0000 and $L(OUT)$=0000.

The AND operation takes place at A while the second operand is sensed and stored at SO. But if the second operand is from MSB cell and requires two sensing operations, the value in A will be override. Therefore, AND operation in location-free ParaBit works only when the second operand is stored in LSB. To be completely location-free, one more latch and some transistors should be engineered to store the second operand from MSB. Such a solution incurs more hardware overhead, will be discussed in future work.

**Table 7: OR operation in location-free ParaBit.**

| *LSB* | *L(SO)* | *L(A)* | *L(B)* | *L(OUT)* |
|---|---|---|---|---|
| *1* | *0* | 1111->1111 | 0110->0000 | 1001->1111 |
| *0* | *1* | 1111->0000 | 0110->0110 | 1001->1001 |

**To conduct an OR operation,** after normal initialization, the value in MSB is sensed by employing $V_{READ1}$ and $V_{READ3}$, and the sensed value is transferred to B so as to have $L(B)$=0110. Then, LSB read operation is performed to have $L(A)$=1100 after re-initializing the latching circuit. As shown in Table 7, if $L(A)$=1, the output should be 1 no matter what the value of MSB is, i.e., $L(B)$=$L(B)_{old} \wedge \overline{L(A)}$=0000 and $L(OUT)$=$\overline{L(B)}$=1111. Otherwise, the output is determined by the value in MSB that has been stored in L2. Thus we have $L(OUT)$=$\overline{L(B)}$=1001.

The OR operation takes place at B while the second operand stored at A is transferred to L2. Since L1 is independent with L2, L1 can be reset and then used to store the second operand from MSB by employing two sensing operations.

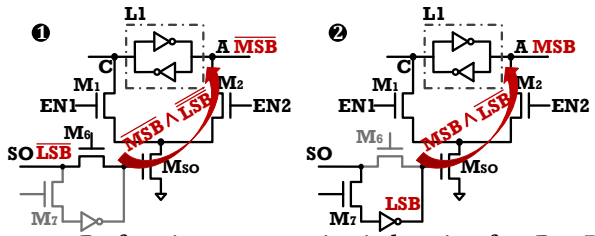



**Figure 8: Performing XOR operation in location-free ParaBit.**

**To conduct an XOR operation,** it can be expressed as $M \oplus N = \overline{M}N + M\overline{N}$, which can be performed by combining AND, OR and NOT operations. Assume that $M$ and $N$ indicate the operands in MSB and LSB bits, respectively. The XOR operation is performed by two steps, as presented in Figure 8. At step 1, to compute $\overline{M}N$, L1 is initialized as Figure 7 to perform NOT operation and followed by reading and storing MSB value at A, i.e., $\overline{M}$. Then, the LSB bit is sensed and its inverted value is stored at SO, i.e., $\overline{N}$. By enabling $M_2$, we can have $L(A)$=$\overline{L(SO)} \wedge L(A)_{old}$, where $\overline{L(SO)}$ is $\overline{\overline{N}}$ and $L(A)_{old}$ is $\overline{M}$. Then, the output of $\overline{M}N$ is transferred to B. At step 2, to compute $M\overline{N}$, operand in MSB is read and stored in A firstly, and then the LSB bit is sensed. To get the original value stored in LSB, additional inverter is engineered between SO and $M_{SO}$. As shown in the right of Figure 8, the original value is got by enabling $M_7$ and disabling $M_6$. Then, $M_2$ is enabled to compute $M\overline{N}$. As a result, we have the result of $M\overline{N}$ at A, and then OR operation is performed by transferring value from A to B. Similar to AND operation, location-free ParaBit performs XOR operation only when the second operand is from LSB.

**For NAND, NOR and XNOR operations,** they can be realized by inverting the initialized status of latching circuit firstly, and then implementing similar sensing and transistor controls.

If there are multiple XOR operations with more than two operands, the buffer inside SSD is used to store temporary result and then reload the result and inverted result as the operands to A in latching circuit. This process is the same as the data-load process in performing write operation [37]. But for AND and OR operations, since their temporary results have been stored at A and B, following bitwise operations can be performed by sensing operand to SO instead of storing previous result to the buffer.

## 4.3 Implementation

Commodity SSDs already support these actions and allow selection of sensing voltages for error tolerance purpose, e.g., read retry or LDPC [42, 60]. Therefore, ParaBit is compatible with existing technologies, which requires firmware upgrade rather than (or with little) extra hardware modification. The following discussion will reveal a sequence of host and device level actions.

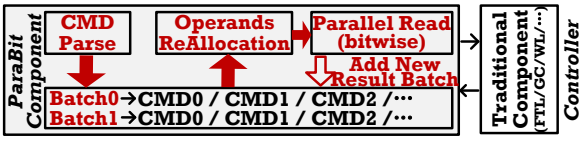



**Figure 9: ParaBit implementation in SSD controller.**

Figure 9 presents an overview of the ParaBit implementation in the SSD controller. The left sub-figure shows the components that deal with the implementation of ParaBit, and traditional components are presented in the right sub-figure. First, after receiving the commands for bitwise operation from the host system, CMD Parse module parses the host commands, constructs device commands and organizes them in a batch structure. Each batch is in charge of one bitwise operation with two operands. Then, Operands ReAllocation module prepares the data for each batch by reallocating operands if necessary. Lastly, Parallel Read module performs different control sequences to accomplish the bitwise operations. If there are multiple bitwise operations, after finishing two batches, a new batch is added to further compute the result of previous two batches. After finishing all bitwise operations, the controller returns the result to the host system.

*4.3.1 Command Reconstruction at Host and Device.* The reserved free bytes in NVMe read command is used to store the bitwise operation semantic, including free bytes in DWord 2, 3 and 13, each of which contains 4 reserved bytes [30, 35]. Assume that there is a formula for calculating $(M\ ?\ N)\ !\ (M\ ?\ N)\ !\ (M\ ?\ N)$, where $(M\ ?\ N)$ is defined as a bitwise batch, $M$ and $N$ indicate the first and second operands in the batch, ? and ! indicate the types of intra- and extra-batch bitwise operation. Thus, at least five semantics, including **intra bitwise type**, **operand tag**, **pointer**, **extra bitwise type** and **batch order**, are required while parsing the formula.

First, to distinguish whether the current command maintains the first or second operand, the first reserved bit in DWord 13 of both operand commands is used to store operand tag, denoted as "0" or "1" in Figure 10; Second, intra-bitwise type takes three reserved bits in DWord 13 of the first operand command, denoted as "i-t", while there are 8 types of bitwise operations; Third, to bind two operands inside a batch, the reserved space (DWord 2 and DWord 3) in first operand command is used to store the logical address of second operand; Fourth, if there are multiple batches that are performed sequentially, the reserved bits in DWord 13 are used to determine the batch's computation order; Fifth, 3 reserved bits in DWord 13 of the second operand command is used to record extra bitwise type, denoted as "e-t". Note that, if operand size exceeds flash page size, the batch is partitioned into several sub-operations. To bind sub-operations, the reserved DWord 2 and DWord 3 of the second operand command in current sub-operation are used to record the logical address of next sub-operation's first operand. If operand (or partitioned operand) size is smaller than flash page size, 8 reserved bits in DWord 13 is used to store the offset address of operand in flash page and operand's size at the granularity of sector.

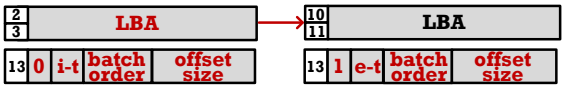



**Figure 10: NVMe ParaBit command reconstruction.**

After receiving commands from system side, CMD Parse module parses them and constructs device commands. For example, we assume that host system implements three bitwise operations with four operands and the size of each operand is twice of flash page size. Thus, after CMD Parse module, each operand is partitioned into two commands and totally eight device commands (denoted as CMD in Figure 11) are constructed and organized in a batch structure. Inside each batch, two sub-operations are contained and each sub-operation includes two device commands at the granularity of flash page. The commands belonging to different sub-operations are used to read one operand, e.g., CMD0 in first sub-operation and CMD2 in second sub-operation. To bind two operands in the same sub-operation, CMD0 and CMD2 stores the physical addresses of CMD1 and CMD3 (e.g., denoted as PBA1 in CMD0), which can be obtained after FTL. To bind two sub-operations, CMD1 stores the physical address of CMD2 (denoted as PBA2 in CMD1). To distinguish two operands, one bit is used to record the operand tag. Also, the intra- and extra- batch bitwise types are recorded in CMD0 and CMD1, respectively. Note that, if operand (or partitioned operand) size is smaller than flash page, the address offset and operand size are recorded as well.

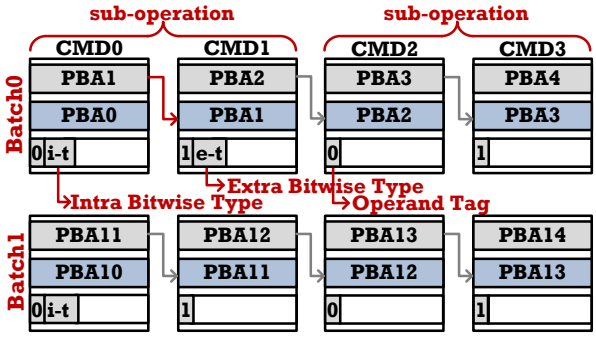



**Figure 11: The structure of batch list.**

*4.3.2 Reallocating Operands.* For ParaBit, after constructing device commands, Operands ReAllocation module is implemented if necessary. Operands ReAllocation module is skipped only when

two operands reside in two aligned pages sharing the same MLCs. Otherwise, it needs to reallocate operands to the same MLCs.

If there is more than one bitwise operation, the result of reallocated operands should be set as one of the operands for the subsequent bitwise operation. Thus, a new device command is constructed to maintain its information. For example, in Figure 12, after performing operand reallocation, data in PBA0 and PBA1 are reallocated to the same MLCs, of which the physical addresses are PBA21 and PBA22. Since reallocated share the same wordline, only the physical address of the first operand page (i.e. PBA21) is recorded in new command. Totally, the computations in Figure 11 deliver four new device commands, constructing a new batch (Batch2 in the bottom of Figure 12). To implement bitwise operation in the new batch, the intra-bitwise type of Batch0 is recorded as previous-bitwise type, denoted as p-t in Batch2. To perform bitwise operation in the new batch, ParaBit is used to read the results of reallocated operands firstly, and then Operands ReAllocation reallocates them followed by Parallel Read module to get the result of new batch.

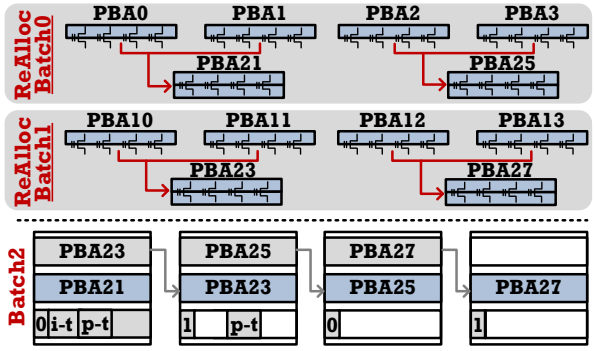



**Figure 12: The structure of new batch.**

Inside SSDs, the scrambling function that mutates the original data for reliability purpose [59] would complicate the use of ParaBit. To solve this problem, scrambling function is disabled when operands are allocated or reallocated, and enabled when the results are restored. Similarly, encryption function is also disabled when operands are reallocated.

*4.3.3 Operands Alignment.* Given ParaBit operates on bits saved in the same MLC cell, it requires operand alignment before operation. The operands may be pre-allocated to aligned locations if the computation semantics can be predicted in advance, e.g., through profiling. Otherwise, the operands get aligned at runtime, which is achieved by reading data from old pages and writing to new pages with offsets. The buffer inside SSD can be exploited to temporarily store the first operand before alignment.

## 4.4 Discussion

*4.4.1 Extending MLC to TLC/QLC.* To simplify the discussion, we choose MLC to elaborate how ParaBit works. MLC-based ParaBit exhibits three advantages: (1) Choosing MLC introduces less write amplification from pre-computation reallocation, which helps to achieve better endurance. (2) MLC has better performance and thus improves the operation efficiency. (3) Most bitwise operations are binary, making it natural to adopt MLC.

In practice, the design principles of ParaBit are applicable to TLC or QLC as well. For example, TLC encodes its eight states (from E, S1 to S7) as 111, 110, 100, 101, 001, 000, 010, and 011, respectively. When conducting the AND operation, sensing voltage at $V_{READ1}$ is applied to determine whether the TLC being sensed is in state E.

*4.4.2 Scalability.* Comparing to other storage medium, e.g., DRAM or 3D XPoint, NAND flash memory based SSD is more cost-efficient, has better parallelism and scalability. ParaBit can achieve better computation efficiency for all-flash storage system that consists of hundreds or thousands of SSDs [25]. Moreover, NAND flash memory based ParaBit also can be applied in NVDIMM based system [45]. The latching circuit is independent of memory cells, thus it can also be extended to other NVMs, such as PCM, STT-MRAM or ReRAM, in which operand bits also can share the same cell. Comparing to NAND flash memory, low-latency NVMs exhibit lower read and write latencies, thus achieving better performance.

*4.4.3 Reliability and Errors.* This is a common problem faced by all bitwise-operation based PIMs (no matter using what kind of memory medium, such as SRAM, DRAM and NVM [12, 19, 22, 33].), because the traditional ECC is not compatible with bitwise logic operation, except for XOR or XNOR. ParaBit accomplishes bitwise operations after the sensing operations, which prevents the ECC modules from detecting and correcting sensing errors. If the application is less error-tolerant, we move two operand pages to new MLCs before operation (even if the operand pages already share the same physical page) to minimize the impact from error sources. We will evaluate the impact of errors in the experimental section.

*4.4.4 Overhead Analysis.* ParaBit introduces space overhead due to storing commands at the device side, which is around 5MB when exploiting the maximal parallelism at the plane level, Location-free ParaBit requires hardware modifications including extra transistors and inverters. The latching circuit and column decoder account for 9.9%~11.9% of the die area [24, 26, 32] while the latching circuit takes up about 35% [37]. To summarize, the hardware overhead is around 1.2% of the die area.

# 5 EVALUATION

## 5.1 Evaluation Setup

**The PIM architecture:** We adopt the Ambit [49] PIM architecture as the baseline. We also construct Ambit with powerful configuration, including 2 ranks, 16 banks, and 256 subarrays with 16KB row buffer size (compared to 4~8KB in regular DRAM). The timing parameters of $t_{RCD}/t_{RAS}/t_{RP}/t_{FAW}$ are set to be 13.75/35/13.75/30 *ns*. The latency of bitwise operations is evaluated using H-spice[51] while the maximum size of operands that can be processed in parallel is set to 16KB with the limitation of power constraint.

**The ISC architecture:** We adopt Cosmos OpenSSD platform [13] as the ISC architecture, which implements SSD controller inside Zynq-7000 FPGA, and then we attache a 512GB SSD [46] to the platform in this work. The Zynq-7000 FPGA with 19.2 Mb BRAM, 437,200 Flip-Flops and 218,600 LUTs is used and its frequency is 100MHz. The LUT in the FPGA is configured as 6-input LUT so that at most five bitwise operations can be processed at once.

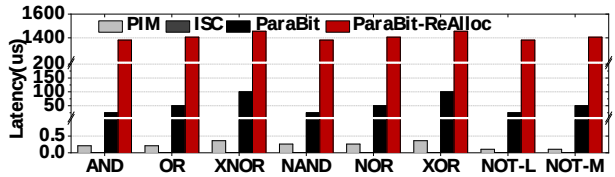
**The ParaBit architecture:** The evaluated SSD is equipped with 128 chips and 4 planes per chip, and the size of flash page is set to 8KB. We used a 512GB SSD. The write latency is set to 640 $\mu$s and

each sensing latency is set to 25 $\mu$s, which are typical for MLC based flash memory [46]. Such a configuration enables SSD to perform a parallel bitwise operation with two 8MB operands. In this work, a widely studied SSD simulator [21] is used to track the duration of bitwise operation in SSD, while Samsung 970 Pro SSD [46] is used to collect the duration of moving results from SSD to DRAM. The size of used DRAM is set to 64GB.

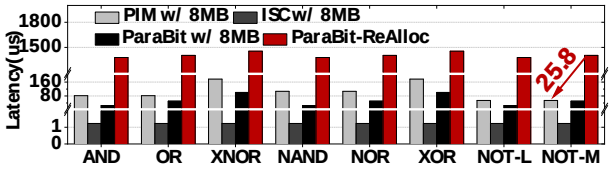
## 5.2 Latency of Bitwise Operation

We implemented six schemes: Ambit based PIM (denoted as PIM), ISC (denoted as ISC), PIM with two 8MB operands (denoted as PIM w/ 8MB), ISC with two 8MB operands (denoted as ISC w/ 8MB) and ParaBit with and without pre-computation reallocation (denoted as PIM and ParaBit-ReAlloc), respectively. Figure 13 compares the latencies of bitwise operations in different schemes.

In Figure 13(a), the latencies of performing one bitwise operation are presented. For PIM, the operation completes in *ns* level. The XNOR and XOR operations are the most time-consuming ones, because these operations are logical combinations of simpler AND, OR and NOT operations. For ISC, bitwise operation is also performed at *ns* level while only one process cycle is required. For ParaBit, since there is no pre-computation reallocation, the XNOR and XOR operations take 100 $\mu$s to sense the result, which accounts for the overall latency. For ParaBit-ReAlloc, the reallocation time before computation dominates the overall latency of bitwise operation. Note that, NOT operation can be performed without operand reallocation, but in ParaBit-ReAlloc, operand reallocation time cost is considered.




(a) The latencies of bitwise operation in PIM, ISC, ParaBit and ParaBit-ReAllocation.




(b) The latencies of bitwise operation in PIM w/ 8MB, ISC w/ 8MB, ParaBit w/ 8MB and ParaBit-ReAllocation.

**Figure 13: The latencies of bitwise operation.**

To demonstrate the efficiency of ParaBit, we extended the operand size to 8MB, the maximum size in ParaBit in the evaluated SSD, and summarized its results in Figure 13(b) as well. For large operands, comparing to other schemes, ISC w/ 8MB achieves the best performance while five bitwise operations are computed in a LUT simultaneously and there are hundreds of thousands of LUTs. PIM partitions each pair of operands to 16KB-sized sub-operands and sequentialize the computation. Therefore, PIM w/ 8MB is always slower than ParaBit w/ 8MB but faster than ParaBit-ReAlloc. For example, NOT-MSB in ParaBit-ReAlloc is 25.8 × slower than that of PIM w/ 8MB. Such a latency gap can be filled by increasing the parallelism of SSDs. While the size of each operand in ParaBit is larger than 206.4MB, it can outperform PIM. For location-free ParaBit, since the access latency on low-latency NVM cells is at *ns* level, location-free ParaBit with two 8MB operands achieves the similar results with ParaBit w/ 8MB, always outperforming PIM architecture.

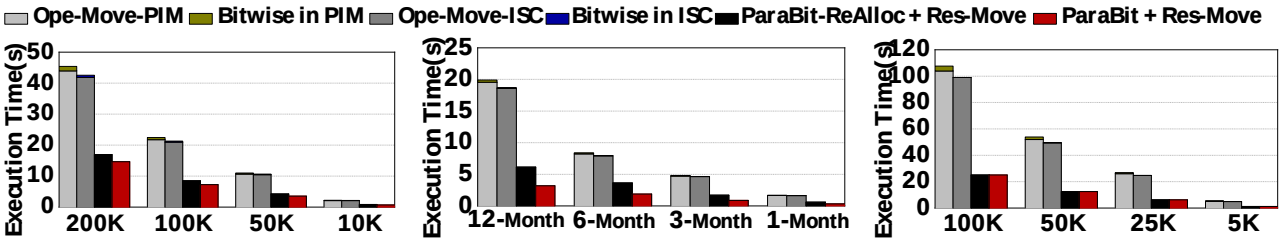
## 5.3 Case Studies

We studied three applications when moving different amount of data from SSD to memory. For PIM and ISC, we moved all data to memory and triggered bitwise operations in memory and FPGA. For ParaBit, we processed data in SSD and moved results to memory.

*5.3.1 Image Segmentation.* In the evaluation, the configuration is the same as that in Section 3. In the discussion, we represent the color recognition result at pixel $m$ as $Re(m) = (C1_a, C1_b, C1_c, C1_d)$ AND $(C2_a, C2_b, C2_c, C2_d)$ AND $(C3_a, C3_b, C3_c, C3_d)$, where $a$, $b$, $c$ and $d$ indicate four types of color, $CX_a$, $CX_b$, $CX_c$ and $CX_d$ indicate whether the value of $X$ channel falls into the recognized color space. To exploit the parallelism of bitwise operations, multiple pixels and images can be grouped and computed at once.

Figure 14(a) summarizes the results, which include the total execution time and its breakdown. We need 140GB space cost to store 200,000 images. After performing AND operations among three channels, we reduce the output size to one third of the original size. As a result, comparing to PIM and ISC, ParaBit (with and without pre-computation allocation) can significantly reduce the data movement cost (Ope-Move-PIM/ISC indicates the time cost of moving operands from SSD to memory in PIM and ISC) to 33.3% and 35.0%, respectively. For the time cost of performing AND operation, ParaBit-ReAlloc reallocates operands before performing each bitwise operation, increasing AND operation time cost by 11.8 and 24.4 times over PIM and ISC, respectively. For ParaBit-ReAlloc, the process of performing AND operation and results movement are pipelined (denoted as ParaBit-ReAlloc+Res-Move), thus it can further reduce the total execution time. Comparing to PIM and ISC, ParaBit-ReAlloc+Res-Move reduces the total execution time to 37.3% and 39.8%, respectively. Furthermore, to reduce bitwise time cost in ParaBit architecture, we pre-allocate operands for the first bitwise operation, and then the result is reallocated with another operand to accomplish the second bitwise operation. Therefore, ParaBit reduces AND operation cost by 51.7% comparing to ParaBit-ReAlloc. To summarize, the total execution time of ParaBit+Res-Move is reduced to 32.3% and 34.4% of PIM and ISC. Also, when total data volume fits in memory, such as image segmentation with 10,000 images, ParaBit and ParaBit-ReAlloc still outperform PIM while data movement time cost is largely reduced.

*5.3.2 Bitmap Indices.* Bitmap index is a program designed as a analytic method for specific databases [8]. We assume to count the number of users who were active every day for the past $m$ months. Thus, two types of operations are required: the AND operation and the bitcount function. The former can be performed by either PIM, ISC or ParaBit while the latter is done by the processor. In the discussion, we represent the computation result of user $y$ as $Res(y) = (V_{1,y})$ AND $(V_{2,y})$ AND ... AND $(V_{x,y})$ ..., where $x$ indicates the numbers of days, $V_{x,y}$ indicates whether user $y$ at day $x$ is active. We compute the results of multiple users simultaneously to fully exploit the parallelism of bitwise operation. In the evaluation, we set the number of users to 800 millions, and varied $m$ from 12 to 1.




(a) Image segmentation. (b) Bitmap indices. (c) Image encryption.

**Figure 14: Execution time breakdown of PIM, ISC and ParaBit architectures in three case studies.**

Figure 14(b) summarizes the evaluation results. From the figure, PIM and ISC spend most of their execution time on moving data from SSD to DRAM and FPGA and less than 2% time on computation. ParaBit effectively reduces the data movement overhead by transferring only the computation results back to DRAM. When $m$=12, the amount of operands reaches 33.99GB. Comparing to PIM and ISC, ParaBit and ParaBit-ReAlloc reduce data movement cost to around 0.3% as only 800 millions bits of results are moved to DRAM. However, when comparing the AND operation performance, PIM, ISC, ParaBit-ReAlloc and ParaBit take 353$ms$, 41$ms$, 6137$ms$ and 3179$ms$, respectively. Therefore, performing bitwise operations in ParaBit and ParaBit-ReAlloc dominates the total execution time. To summarize, the total execution time of ParaBit-ReAlloc+Res-Move was reduced to 30.8% and 32.8% of PIM and ISC while ParaBit+Res-Move reduces total execution time to 15.9% and 17.0%.

*5.3.3 Image Encryption.* XOR operation is widely applied in image encryption[20] and image feature extraction [57]. Take image encryption as an example, where XOR operation is used to encrypt or decrypt image data. The number of original images varies from 100,000 to 5,000 and the resolution of each image is set to 800×600, each channel requires 8 bits. At most, we need 140GB space cost to store 100,000 images. In the discussion, the encryption process can be presented as *Cipher(x) = Ori(x)* XOR *Key(x)*, where $x$ indicates the pixel's location, *Cipher(x)* indicates the encrypted pixel and *Key(x)* indicates the pixel of key image. By adopting ParaBit, images can be encrypted inside SSD without data movement to memory.

Figure 14(c) summarizes the results. Take 100,000 images as an example. For PIM and ISC, the data movement time cost still dominates the total execution time while XOR operations take less than 3.5% and 0.21%, respectively. To summarize, comparing to PIM and ISC, ParaBit-ReAlloc reduces the execution time cost to 23.3% and 25.3%, respectively. Since only one XOR operation is performed to encrypt an image, ParaBit and ParaBit-ReAlloc implement the same process: reading original image and writing it to the same cells with the accompany of key image, thus two ParaBit schemes achieve the same result.

*5.3.4 Overall Impact on Application Performance.* From the preceding case studies, we find that the overall performance impact of ParaBit on modern applications depends on two factors: one is the amount of bitwise operations while the other is the data preparation cost. Clearly, the more bitwise operations the application has, the better performance improvement it may achieve. Given modern applications becoming increasingly data intensive, there are a number of applications that contain frequent bitwise operation, making them particularly suitable for ParaBit acceleration.

- *Deduplication Application.* Modern deduplication applications widely adopt XOR operation to differentiate two pages [9, 29]. Adopting ParaBit not only frees the system from allocating a dedicated processor for deduplication computation, but also eliminates the data movement overhead. The latter may occupy 80% or more of the system offchip memory bandwidth [54].
- *Binary Neural Network.* Modern CNNs demand a large number of weights, e.g., 150GB in ImageNet [44]. While binary weight based CNN algorithms [34, 52] achieve 32× space reduction, i.e., saving one-bit instead of 32-bit weight parameters, the required storage may still be more than the capacity of the memory of many systems. ParaBit acceleration is suitable for binary weighted neural networks due to their predominant bitwise operations involved in the computation.
- *Fast Data Scanning.* Data-intensive applications often involve frequent data scanning processing [11, 28]. Given the huge size of database system [40], by leveraging XOR operation in scanning process, ParaBit can improve the scanning efficiency while the operations are performed inside SSDs.

The cost to prepare data for ParaBit operations also has a large impact on the overall application performance. For example, it is usually not beneficial to adopt ParaBit to accelerate dynamically generated data as it would require writing the dynamic data into flash before operation, making the data preparation cost too high. Instead, adopting PIM or pipelined based PIM might achieve better performance improvement. In summary, ParaBit is more suitable for applications that apply bitwise operations on in-storage static data, e.g., files that exist in SSDs before deduplication.

## 5.4 Endurance Impact

The pre-computation allocation in ParaBit tends to trigger more GCs, shortening the SSD lifetime. We conducted the experiment to evaluate this impact. For bitmap, image segmentation and image encryption, when the tracking duration is set to 12 months and the numbers of images are set to 200,000 and 100,000, their reallocated operands' sizes reach 67.79GB, 186.67GB and 140GB, respectively. We set the maximal value of MLC based SSDs' Terabytes Written (TBW) to 600 [46], ParaBit reduces the TBW to 200.67, 257.51 and 300 assuming the SSD is dedicated to these applications exclusively. However, a more applicable usage is to employ the SSD as both storage and computing medium, the TBW value would be larger than above values.

## 5.5 Performance of Location-Free ParaBit

To evaluate the efficiency of location-free ParaBit, we compared the performance of three types of ParaBit schemes and summarized the results in Figure 15.

The left part of Figure 15 compares the latencies when performing bitwise operations on two 8MB operands. As a result, `ParaBit-ReAlloc` takes the longest latency due to high operand reallocation cost. By adopting pre-computation allocation, `ParaBit` achieves the smallest latency. As a tradeoff, `ParaBit-LocFree` reduces the time spent in operand reallocation as more sensing operations are employed comparing to `ParaBit` with pre-computation allocation. But if there are multiple bitwise operations, `ParaBit` with pre-computation allocation only works for the first bitwise operation, thus suffers from reallocation overhead during performing following bitwise operations.

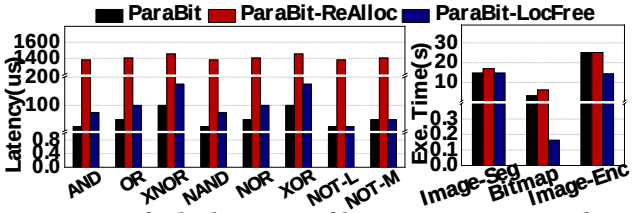



**Figure 15: *Left:* the latencies of bitwise operation; *Right:* total execution time.**

The right part of Figure 15 summaries the execution time for three case studies. We store all data in LSB bits of MLCs. For Image Segmentation, since result movement overhead is the dominant factor, `ParaBit-LocFree` and `ParaBit` achieve the similar performance. For Bitmap, while the size of moved data is significantly reduced, the performance of `ParaBit` depends mainly on the efficiency of bitwise operations. Therefore, `ParaBit-LocFree` has the smallest execution time while there is no operand reallocation — `ParaBit-LocFree` reduces the execution time to 5.23% and 10.1% of `ParaBit-ReAlloc` and `ParaBit`, respectively. For Image Encryption, ParaBit-LocFree performs `XOR` operation without operand reallocation, thus reducing the execution time cost to 57.1% of `ParaBit-ReAlloc` and `ParaBit`.

## 5.6 Energy Consumption of ParaBit

We next evaluate the read/write energy consumption in ParaBit and summarize the results in Figure 16. We use the NAND system power calculator from Micron [38] to compare the per read/write operation energy consumption in the baseline and ParaBit schemes. We normalize the results to the read and write operations of MSB pages, denoted as dashed lines, respectively.

In the figure, while `ParaBit-ReAlloc` is the scheme that consumes the most energy, it consumes up to 2.65% more energy than that of baseline write operation. `ParaBit` achieves the lowest energy consumption among all ParaBit variants — its energy consumption is mostly comparable to that of the baseline MSB read. In the worst case, it is about 2× of that of the baseline MSB read.

## 5.7 Compression in ParaBit

By default, PIM and ISC could have data stored in compressed format in storage while ParaBit stores the data in uncompressed format. For image segmentation with 200,000 images, `ParaBit-LocFree` breaks even with PIM when the data can be compressed to 30.1% or lower. For bitmap, the total execution time taken by `ParaBit-LocFree` is less than computation time cost of PIM, thus it always outperforms PIM while there still exists data movement overhead for compression based PIM.

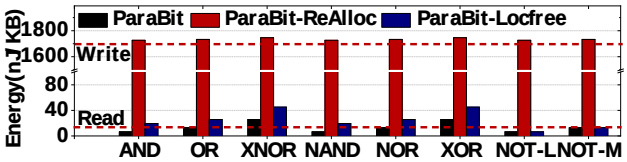



**Figure 16: Energy consumption of ParaBit schemes.**

## 5.8 Error Tolerance

To alleviate the impact from bit errors, ECC module is triggered during operand reallocation, thus only the impact from P/E cycle is counted. In the evaluation, customer grade Intel MLC flash chips with cache read are evaluated [23]. When the number of P/E cycles is 5K, the average and maximal numbers of bit errors per WL after seventh ( for `XOR` operation) sensings are 0.945 and 5 while the page size is 8KB, as shown in the left of Figure 17. Such a good reliability comes from two aspects: First, MLC flash cells are engineered with large state margin so as to tolerate significant voltage state shifts; Second, voltage calibration read that can minimize the number of bit errors is adopted to optimize the reliability of flash memory [7, 17]. In the right part of Figure 17, we evaluate the percentages of bit errors at application level. In worst case, the percentage of bit errors only is 0.00149% for `XOR`-based image encryption.

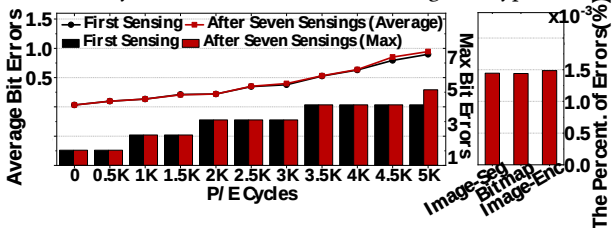



**Figure 17: The number of bit errors with the increase of P/E cycles.**

## 6 CONCLUSION

In this work, we proposed ParaBit, a processing-in-flash architecture, to perform massive parallel bitwise operations in flash memory. ParaBit can be implemented within commodity SSDs without (or with little) extra hardware, which can process data inside SSDs and dramatically reduces the amount of data moved from SSDs to DRAM or processor. ParaBit exhibits good scalability, good cost-effectiveness, and great potentials for modern data-intensive applications.

## ACKNOWLEDGMENTS

We sincerely thank anonymous reviewers for their constructive feedback. This work is partially supported by the National Key Research and Development Program of China (No. 2018YFB1003301), the National Natural Science Foundation of China (No. 61832011, 62022051, 61772300, 61877035), Zhejiang Lab (No. 2020KC0AB03), the National Science Foundation (No. 2011146, 1738783, 1910413, 1718080) and the China Postdoctoral Science Foundation (No. 2020M 680568, 2021T140376).

# REFERENCES


[1] Nitin Agrawal, Vijayan Prabhakaran, Ted Wobber, John D Davis, Mark S Manasse, and Rina Panigrahy. 2008. Design tradeoffs for SSD performance.. In *ATC*. USENIX.

[2] Junwhan Ahn, Sungpack Hong, Sungjoo Yoo, and et al. 2015. A scalable processing-in-memory accelerator for parallel graph processing. In *ISCA*. ACM.

[3] Berkin Akin, Franz Franchetti, and James C Hoe. 2015. Data reorganization in memory using 3D-stacked DRAM. In *CAN*. ACM.

[4] Hadi Asghari-Moghaddam, Young Hoon Son, Jung Ho Ahn, and Nam Sung Kim. 2016. Chameleon: Versatile and practical near-DRAM acceleration architecture for large memory systems. In *MICRO*. IEEE.

[5] Julien Borghetti, Gregory S Snider, Philip J Kuekes, J Joshua Yang, Duncan R Stewart, and R Stanley Williams. 2010. 'Memristive'switches enable 'stateful'logic operations via material implication. In *Nature*. Nature Publishing Group.

[6] James Bruce, Tucker Balch, and Manuela Veloso. 2000. Fast and inexpensive color image segmentation for interactive robots. In *IROS*. IEEE.

[7] Yu Cai, Saugata Ghose, Erich F Haratsch, Yixin Luo, and Onur Mutlu. 2017. Error characterization, mitigation, and recovery in flash-memory-based solid-state drives. In *Proceedings of the IEEE*. IEEE.

[8] Chee-Yong Chan and Yannis E Ioannidis. 1998. Bitmap index design and evaluation. In *SIGMOD*. ACM.

[9] Feng Chen, Tian Luo, and Xiaodong Zhang. 2011. CAFTL: A Content-Aware Flash Translation Layer Enhancing the Lifespan of Flash Memory based Solid State Drives. *Fast*.

[10] Ping Chi, Shuangchen Li, Cong Xu, Tao Zhang, Jishen Zhao, Yongpan Liu, Yu Wang, and Yuan Xie. 2016. Prime: A novel processing-in-memory architecture for neural network computation in reram-based main memory. In *ISCA*. ACM.

[11] Jaeyoung Do, Yang-Suk Kee, Jignesh M Patel, Chanik Park, Kwanghyun Park, and David J DeWitt. 2013. Query processing on smart ssds: Opportunities and challenges. In *SIGMOD*. ACM.

[12] Charles Eckert, Xiaowei Wang, Jingcheng Wang, Arun Subramaniyan, Ravi Iyer, Dennis Sylvester, David Blaaauw, and Reetuparna Das. 2018. Neural cache: Bit-serial in-cache acceleration of deep neural networks. In *ISCA*. IEEE.

[13] ENC. [n.d.]. Cosmos OpenSSD Platform. http://www.openssd-project.org/wiki/Cosmos_OpenSSD_Platform.

[14] Chiou-Shann Fuh, Shun-Wen Cho, and Kai Essig. 2000. Hierarchical color image region segmentation for content-based image retrieval system. In *TIP*. IEEE.

[15] Congming Gao, Liang Shi, Chun Jason Xue, Cheng Ji, Jun Yang, and Youtao Zhang. 2019. Parallel all the time: Plane level parallelism exploration for high performance SSDs. In *MSST*. IEEE.

[16] Congming Gao, Liang Shi, Mengying Zhao, Chun Jason Xue, Kaijie Wu, and Edwin H-M Sha. 2014. Exploiting parallelism in I/O scheduling for access conflict minimization in flash-based solid state drives. In *MSST*. IEEE.

[17] Congming Gao, Min Ye, Qiao Li, Chun Jason Xue, Youtao Zhang, Liang Shi, and Jun Yang. 2019. Constructing large, durable and fast SSD system via reprogramming 3D TLC flash memory. In *MICRO*. IEEE.

[18] Fei Gao, Georgios Tziantzioulis, and David Wentzlaff. 2019. Computedram: In-memory compute using off-the-shelf drams. In *MICRO*. IEEE.

[19] Saransh Gupta, Mohsen Imani, and Tajana Rosing. 2018. Felix: Fast and energy-efficient logic in memory. In *ICCAD*. IEEE.

[20] JongWook Han, Choon-Sik Park, Dae-Hyun Ryu, and Eun-Soo Kim. 1999. Optical image encryption based on XOR operations. In *Optical Engineering*. International Society for Optics and Photonics.

[21] Yang Hu, Hong Jiang, Dan Feng, and et al. 2012. Exploring and exploiting the multilevel parallelism inside SSDs for improved performance and endurance. In *TC*. IEEE.

[22] Mohsen Imani, Saransh Gupta, Yeseong Kim, and Tajana Rosing. 2019. Floatpim: In-memory acceleration of deep neural network training with high precision. In *ISCA*. IEEE.

[23] Intel 2015. *Intel 64M25 Compute NAND Flash Memory Datasheet*. Intel.

[24] Woopyo Jeong, Jae-woo Im, Doo-Hyun Kim, and et al. 2015. A 128 Gb 3b/cell V-NAND flash memory with 1 Gb/s I/O rate. In *JSSC*. IEEE.

[25] Young Tack Jin, Sungjoon Ahn, and Sungjin Lee. 2018. Performance analysis of nvme ssd-based all-flash array systems. In *ISPASS*. IEEE.

[26] Dongku Kang, Woopyo Jeong, Chulbum Kim, and et al. 2016. 256 Gb 3 b/cell V-NAND flash memory with 48 stacked WL layers. *JSSC*.

[27] Hyukjoong Kim, Dongkun Shin, Yun Ho Jeong, and Kyung Ho Kim. 2017. SHRD: Improving Spatial Locality in Flash Storage Accesses by Sequentializing in Host and Randomizing in Device. In *FAST*. USENIX, 271–284.

[28] Sungchan Kim, Hyunok Oh, Chanik Park, and et al. 2011. Fast, energy efficient scan inside flash memory SSDs. *ADMS*.

[29] Ricardo Koller and Raju Rangaswami. 2010. I/O deduplication: Utilizing content similarity to improve I/O performance. In *TOS*. ACM.

[30] Gunjae Koo, Kiran Kumar Matam, I Te, and et al. 2017. Summarizer: trading communication with computing near storage. In *MICRO*. IEEE.

[31] Joo Hwan Lee, Hui Zhang, Veronica Lagrange, and et al. 2020. SmartSSD: FPGA Accelerated Near-Storage Data Analytics on SSD. *CAL*.

[32] Seungjae Lee, Jin-yub Lee, Il-han Park, and et al. 2016. 7.5 A 128Gb 2b/cell NAND flash memory in 14nm technology with tPROG= 640$\mu$s and 800MB/s I/O rate. In *ISSCC*. IEEE.

[33] 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 *DAC*. ACM.

[34] Xiaofan Lin, Cong Zhao, and Wei Pan. 2017. Towards accurate binary convolutional neural network. *arXiv preprint arXiv:1711.11294*.

[35] Kevin Marks. 2013. An nvm express tutorial. In *Flash Memory Summit*.

[36] Sally A McKee. 2004. Reflections on the memory wall. In *CF*.

[37] Rino Micheloni, Luca Crippa, and Alessia Marelli. 2010. *Inside NAND flash memories*. Springer Science & Business Media.

[38] Micron. [n.d.]. Parallel NAND System Power Calculator. https://www.micron.com/support/tools-and-utilities/nand-system-power-calculator.

[39] Micron 2018. *NAND MLC Flash Memory Datasheet*. Micron.

[40] Kimberly Mlitz. [n.d.]. Data center storage capacity worldwide from 2016 to 2021. https://www.statista.com/statistics/638593/worldwide-data-center-storage-capacity-cloud-vs-traditional//.

[41] Onur Mutlu, Saugata Ghose, Juan Gómez-Luna, and Rachata Ausavarungnirun. 2020. A Modern Primer on Processing in Memory. In *arXiv preprint arXiv:2012.03112*.

[42] Jisung Park, Myungsuk Kim, Myoungjun Chun, Lois Orosa, Jihong Kim, and Onur Mutlu. 2020. Reducing Solid State Drive Read Latency by Optimizing Read-Retry. In *ASPLOS*. ACM.

[43] Zhenyuan Ruan, Tong He, and Jason Cong. 2019. INSIDER: Designing In-Storage Computing System for Emerging High-Performance Drive. In *ATC*. USENIX.

[44] Olga Russakovsky, Jia Deng, Hao Su, and et al. 2015. Imagenet large scale visual recognition challenge. In *IJCV*. Springer.

[45] Arthur Sainio. 2016. NVDIMM: changes are here so what's next. In *Memory Computing Summit*.

[46] Samsung. [n.d.]. Samsung 970 Pro. https://www.samsung.com/semiconductor/minisite/ssd/product/consumer/970pro/.

[47] Abu Sebastian, Manuel Le Gallo, Riduan Khaddam-Aljameh, and Evangelos Eleftheriou. 2020. Memory devices and applications for in-memory computing. In *Nature nanotechnology*. Nature Publishing Group.

[48] Sudharsan Seshadri, Mark Gahagan, Sundaram Bhaskaran, and et al. 2014. Willow: A User-Programmable SSD. In *OSDI*. USENIX.

[49] Vivek Seshadri, Donghyuk Lee, Thomas Mullins, and et al. 2017. Ambit: In-memory accelerator for bulk bitwise operations using commodity DRAM technology. In *MICRO*. IEEE.

[50] Yong Ho Song, Sanghyuk Jung, Sang-Won Lee, and Jin-Soo Kim. 2014. Cosmos openSSD: A PCIe-based open source SSD platform. In *Flash Memory Summit*.

[51] Synopsys. [n.d.]. H-spice. https://www.synopsys.com/.

[52] Wei Tang, Gang Hua, and Liang Wang. 2017. How to train a compact binary neural network with high accuracy? *AAAI*.

[53] Hossein Valavi, Peter J Ramadge, Eric Nestler, and Naveen Verma. 2019. A 64-tile 2.4-Mb in-memory-computing CNN accelerator employing charge-domain compute. In *JSSC*. IEEE.

[54] Grant Wallace, Fred Douglis, Hangwei Qian, and et al. 2012. Characteristics of backup workloads in production systems.. In *FAST*. USENIX.

[55] Zhuo-Rui Wang, Yu-Ting Su, Yi Li, Ya-Xiong Zhou, Tian-Jian Chu, Kuan-Chang Chang, Ting-Chang Chang, Tsung-Ming Tsai, Simon M Sze, and Xiang-Shui Miao. 2016. Functionally complete Boolean logic in 1T1R resistive random access memory. In *EDL*. IEEE.

[56] Xin Xin, Youtao Zhang, and Jun Yang. 2020. ELP2IM: Efficient and Low Power Bitwise Operation Processing in DRAM. In *HPCA*. IEEE.

[57] Ching-Nung Yang and Dao-Shun Wang. 2013. Property analysis of XOR-based visual cryptography. In *TCSVT*. IEEE.

[58] He Zhang, Wang Kang, Bi Wu, Peng Ouyang, Erya Deng, Youguang Zhang, and Weisheng Zhao. 2019. Spintronic processing unit within voltage-gated spin hall effect MRAMs. In *TN*. IEEE.

[59] Li Zhang, Shen gang Hao, Jun Zheng, Yu an Tan, Quan xin Zhang, and Yuan zhang Li. 2015. Descrambling data on solid-state disks by reverse-engineering the firmware. In *Digital Investigation*.

[60] Kai Zhao, Wenzhe Zhao, Hongbin Sun, and et al. 2013. LDPC-in-SSD: Making advanced error correction codes work effectively in solid state drives. In *FAST*. USENIX.