Search-in-Memory: Reliable, Versatile, and Efficient Data Matching in SSD's NAND Flash Memory Chip for Data Indexing Acceleration

Yun-Chih Chen, *Member, IEEE*, Yuan-Hao Chang, *Fellow, IEEE*, and Tei-Wei Kuo, *Fellow, IEEE*

***Abstract*—To index the increasing volume of data, modern data indexes are typically stored on solid-state drives and cached in DRAM. However, searching such an index has resulted in significant I/O traffic due to limited access locality and inefficient cache utilization. At the heart of index searching is the operation of filtering through vast data spans to isolate a small, relevant subset, which involves basic equality tests rather than the complex arithmetic provided by modern CPUs. This article demonstrates the feasibility of performing data filtering directly within a NAND flash memory chip, transmitting only relevant search results rather than complete pages. Instead of adding complex circuits, we propose repurposing existing circuitry for efficient and accurate bitwise parallel matching. We demonstrate how different data structures can use our flexible SIMD command interface to offload index searches. This strategy not only frees up the CPU for more computationally demanding tasks, but it also optimizes DRAM usage for write buffering, significantly lowering energy consumption associated with I/O transmission between the CPU and DRAM. Extensive testing across a wide range of workloads reveals up to a 9× speedup in write-heavy workloads and up to 45% energy savings due to reduced read and write I/O. Furthermore, we achieve significant reductions in median and tail read latencies of up to 89% and 85%, respectively.**

***Index Terms*—Database systems, databases, flash memories, indexes systems, memory, systems.**

I. Introduction

Challenges *of Indexing Vast Amount of Data:* Data indexes, such as hash tables and trees, are fundamental for quickly retrieving relevant data from vast datasets. As the volume of data to be indexed explodes, the size of the index is growing significantly large. In many user-facing databases that execute complex queries, index size can even surpass the data being indexed [1]. Given that accessing an index invariably precedes any data retrieval, indexes are commonly pinned in-memory to boost performance. With the introduction of high-speed solid-state drives (SSDs), even systems sensitive to latency—those interfacing directly with users—resort to storing indexes on SSDs and loading them into DRAM on-demand. Upon loading an index block (for instance, a B-Tree's leaf node or a hash table's bucket) into DRAM, a subsequent scan through the memory page that contain arrays of candidate entry is necessary to find the matching one. Such a parallel equality test is often accelerated with SIMD instructions.

As I/O can easily become the bottleneck, compression and data prefetching are common techniques employed to reduce I/O and hide latency. However, in many workloads indexes exhibit low compressibility, and decompression incurs overhead [1]. Moreover, prefetching can accelerate the replacement of loaded index blocks. In large-scale data systems, where the working set size far exceeds DRAM capacity and the accesses scatter widely, index blocks can be repetitively loaded and evicted from DRAM. Even if all index blocks fit entirely in DRAM, they can still be evicted after context-switching to other processes that might also allocate memory. Another pressing issue is the management of index updates. These updates not only require considerable buffering to mitigate the SSD's high write costs but also introduce multiple data versions that compete for the limited DRAM cache space with index reads, leading to increased I/O due to more frequent read cache misses.

To solve the I/O bottleneck, one can either increase DRAM capacity or I/O bandwidth. However, both approaches bring substantial costs and power consumption. In environments where cost efficiency is as crucial as performance, the focus should not solely be on maximizing index retrieval's throughput but on enhancing the utility of the retrieved indexes. Perhaps the best way is to fundamentally cut the amount of indexes that need to be transferred from the storage system.

There have been numerous innovations in data structures aimed at optimizing data indexing and system-level optimizations, such as kernel bypassing, to maximize I/O bus utilization. This article takes a different approach, focusing on the core operation of data indexing: matching a query against a vast array of candidate entries. Within the constraints of today's von Neumann architecture, this equality test operation occurs in the CPU only after transferring all candidate entries from storage. Yet, this operation, predominantly data-bound,

Manuscript received 5 August 2024; accepted 10 August 2024. This work was supported in part by the German Federal Ministry of Education and Research (BMBF) in the Course of the 6GEM Research Hub under Grant 16KISK038, and in part by the Deutsche Forschungsgemeinschaft (DFG), as part of the Project OneMemory under Grant 405422836. This article was presented at the International Conference on Hardware/Software Codesign and System Synthesis (CODES + ISSS) 2024 and appeared as part of the ESWEEK-TCAD Special Issue. This article was recommended by Associate Editor S. Dailey. *(Corresponding author: Yun-Chih Chen.)*

Yun-Chih Chen is with the Department of Computer Science, Technische Universität Dortmund Fakultät für Informatik, 44227 Dortmund, Germany (e-mail: yunchih.chen@tu-dortmund.de).

Yuan-Hao Chang is with the Institute of Information Science, Academia Sinica, Taipei 115, Taiwan (e-mail: johnson@iis.sinica.edu.tw).

Tei-Wei Kuo is with the Delta Electronics, Department of Computer Science and Information Engineering, High Performance and Scientific Computing Center, and the Center of Data Intelligence: Technologies, Applications, and Systems, National Taiwan University, Taipei 10617, Taiwan (e-mail: ktw@csie.ntu.edu.tw).

Digital Object Identifier 10.1109/TCAD.2024.3443702

does not require the complex arithmetic or control flows modern CPUs offer and could be executed by simpler hardware circuits.

This leads us to question whether equality tests could be integrated deeper into the storage system. While processing in memory (PiM) has been explored as a solution to the bottleneck between the CPU and DRAM, it does not address DRAM's capacity scaling challenges. Conversely, a NAND-flash-memory-based solution offers higher energy efficiency and capacity. In this article, we explore this direction by introducing the search-in-memory (SiM) chip.

SiM is based on the architecture of existing triple-level cell (TLC) flash memory chip. Instead of introducing a full-fledged hardware-based indexing solution, we aim to minimize hardware changes and use software to decompose complex indexing operations into simple hardware instructions, similar to the design philosophy of RISC CPUs. We demonstrate how to minimally modify an existing flash memory chip to conduct equality tests directly in itself and send only the relevant results in response to a search request rather than the entire page to fundamentally reduce the I/O traffic.

In our experiment, we also demonstrate the performance characteristics of index search under various workloads, query distribution, and system constraints, as well as how SiM can improve system efficiency by reducing I/O transmission and increasing cache utilization. We make the following contributions.

1) We introduce the SiM chip, a standalone flash memory chip minimally adopted from existing chips to realize on-chip equality tests. SiM features a versatile SIMD interface with two primitives: a) search and b) gather command (Section III). This interface makes SiM adaptable for various data-bound operations, offering flexibility and applicability to different scenarios.
2) Maintaining data integrity is a significant challenge for NAND-flash-based on-chip computing. To address this, we propose the "Optimistic Error Correction," which optimizes the common case of no errors in single-level cell (SLC) pages, while providing a fallback solution for rare corner cases (Section IV).
3) We introduce several system integrations, from general data structures like B+Tree, which is used in many systems, to supporting database analytical queries, to demonstrate SiM's generalizability and flexibility (Section V).

II. Background and Motivations

A. SSD's Parallelism

As shown in Fig. 1, an SSD is made up of multiple flash memory chips that communicate with a central controller via high-speed data channels. A chip has several dies, each can simultaneously conduct memory operations. Modern SSDs' impressive I/O bandwidth is the result of parallel operations across multiple chips (i.e., *interchip parallelism*) and the activation of multiple components within a single chip (i.e., *intrachip parallelism*). However, the degree of parallelism has a physical limit. Heat dissipation is becoming increasingly difficult, even in data centers, as the density of modern flash memory chips increases. Too many parallel operations can result in electric currents that exceed the hardware power budget.

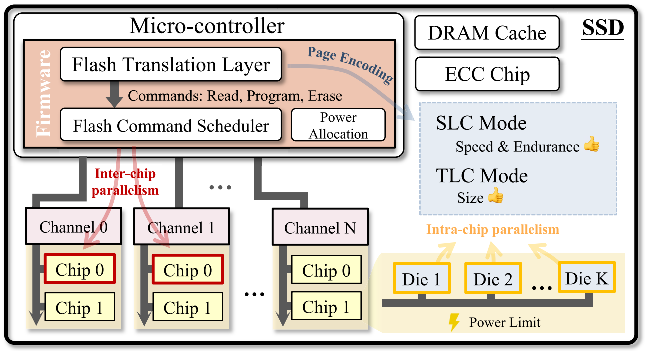


Fig. 1. SSD architecture.

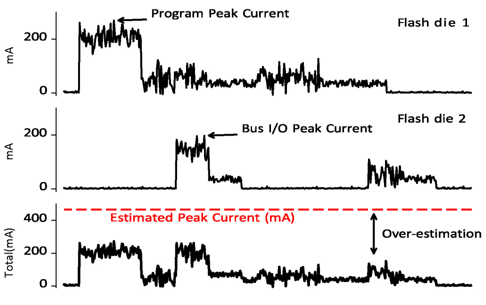


Fig. 2. Conceptual illustration of current consumption in a NAND flash chip.

B. Bus I/O Can Limit SSD's Parallelism

As shown in Fig. 2, a flash memory command consumes varying amount of current throughout various phases (I/O transfer phase, the read/program phase, and the status phase). To simplify power management, many controllers represent the peak current consumption of a command as its overall current usage [2], [3], [4]. This ensures that the total current consumption of the entire chip does not exceed the power budget when multiple commands are executed concurrently. On the other hand, if the aggregate peak currents is anticipated to exceed the power budget, the controller must restrain from dispatching further commands even if the target flash die is idle. Lowering the peak current of a flash command is therefore critical for ensuring efficient power allocation and parallelism.

As SSDs' capacity increases, more data must be moved in and out, increasing the demand for higher I/O bandwidth [5]. The increased bandwidth requirement is often fulfilled by increasing the I/O clock rate, but such an approach can easily make the I/O phase to become the phase in a flash command that draws the peak current. For instance, transferring a 16-KiB page at a clock frequency of 1.6 GHz can consume up to 50% of a chip's maximum power budget [2].

Performance scaling through continually increasing the I/O clock rate is not sustainable and there is a need to fundamentally reduce the bandwidth demand. In fact, as we will

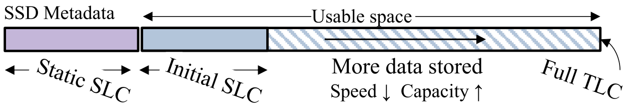


Fig. 3. Commercial SSD.

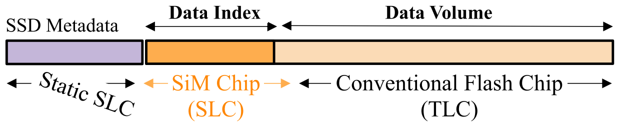


Fig. 4. SiM-enhanced SSD.

show in this article, a decrease in I/O bandwidth does not always result in lower application performance. By filtering out unnecessary data transfer at its source, it is possible to operate I/O buses at a reduced clock rate while preserving the application-perceivable throughput. This article aims to enable such a filtering at low cost.

C. Capacity and Metadata Scaling Must Go Hand-in-Hand

Recently, improvements in 3D-NAND Flash memory technology have made it possible to stack more than 300 layers of memory cells [5], each cell storing multiple bits. This increases SSD's capacity to unprecedented levels. However, without proportionate scaling of metadata storage, the efficiency of retrieving the increased volume of data will be seriously compromised.

SSDs use SLC and TLC modes to encode data and metadata differently in order to meet the specific needs of data and metadata storage. The speed and durability of SLC mode—which stores one bit per cell—make it the preferred method for storing metadata. TLC mode—which stores three bits per cell—is used for data storage because it has a higher storage density.

Fig. 3 depicts the architecture of a typical commercial SSD. A small section of the memory cell encoded in SLC is used to store internal metadata or a write buffer, while the remaining memory cell encoded in TLC is used to store user data. The user data section can transition between SLC and TLC depends on capacity usage. Data are stored in SLC if user utilizes less than advertised capacity. As more capacity is used, the SSD controller transparently converts the SLC-encoded data into TLC. However, such an implicit hybrid model does not guarantee that the user's metadata will be accessed optimally.

In this article, we propose allocating a portion of the user-visible capacity to store data indexes, as depicted in Fig. 4. We implement the index storage with the SiM chip in SLC mode. Although our model has a lower total capacity than using TLC mode for the entire user visible capacity, it provides better metadata access performance and endurance.

D. Case for New Chip Optimized for Data Indexing

There must be a compelling case for designing a new hardware solution because it might bring a huge engineering cost. Data indexing is frequently the first step in querying large data systems, such as file systems, databases, and search engines for narrowing down the search space. The process of executing a key query on a typical database index is as follows. First, an in-memory index structure is queried to locate the leaf index pages. These leaf index pages can be, for example, the leaf node of a B-Tree or a bucket in a hash table. Then, the leaf index page is searched to locate the corresponding entry. These indexes are so large that they must be stored on SSDs and loaded into host memory on demand before the CPU can search the query key in the array of candidate entries in the index pages. The search is usually performed using either SIMD or binary search. However, transferring a large number of index pages between SSD and host memory for matching is usually the performance bottleneck. It also consumes significant amount of I/O bandwidth and power.

The I/O bottleneck in data indexing between the SSD and host has led to the development of various near-storage processing solutions, which conduct data matching in the SSD controller's CPU [6], [7]. However, we argue that instead of loading the vast amount of candidate entries into general-purpose processors to match with a small query key; we should reverse the I/O direction by shipping the query key to where the candidate entries are stored. Several PiM proposals have used this approach [8], [9], [10]. However, many proposals incorporate a processing element (PE) into the memory array or a specialized pattern matching accelerator [11] in the peripheral circuit, increasing design complexity and manufacturing costs.

This article demonstrates the feasibility of adapting the existing design of NAND flash chips to enable on-chip index search. We find that index searches can utilize the existing logic gates within a flash memory chip's peripheral circuits, reducing the need for substantial additional hardware investments. This approach repurposes hardware initially intended for core data storage functionalities. For instance, the registers and logic gates within each page buffer (PB), originally designed for the encoding and decoding of multiple bits within a memory cell, can be repurposed to execute bit-serial matches. Similarly, the page-wide counter, initially devised for verifying data programming, can be adapted for the aggregation of match results. This strategic repurposing of existing circuits introduces new indexing capabilities while maintaining the original functionalities and without significantly affecting the chip's area or power budget.

III. SEARCH-IN-MEMORY

We introduce the SiM chip, which integrates vectorized data matching into NAND flash memory. This allows data-bound operations to be executed directly within the SSD, eliminating the need to transfer index pages to the CPU. Rather than viewing index pages as opaque data, SiM treats the page content as an array of fixed-width data.

SiM offers a generic SIMD interface, featuring two primary commands: 1) *search* and 2) *gather*. The `search` command compares an input key with the data array in the index page, generating a matching bitmap. Subsequently, the `gather` command uses this bitmap to extract specific data chunks within an index page, bypassing nonmatching data. This

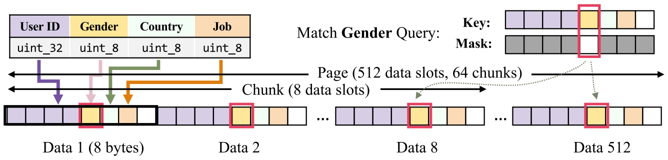


Fig. 5. Page format and data encoding.

targeted approach reduces the bandwidth waste and excessive energy often linked with full page-sized I/O transfers.

A. SiM Page Format

As shown in Fig. 5, SiM recognizes a data page as an array of 8-byte data slots, a format central to many index structures, like the leaf node of a B+Tree or an external hash table's bucket. Thus, a 4-KiB page corresponds to an array with 512 data slots.[1] When a search command is performed, the chip matches the 8-byte query key with these slots, returning a 512-bit bitmap as the match result.

To reduce wiring overhead in the gather command implementation, we group every eight data slots into a *chunk*. This chunk serves as the minimal data transfer unit. Optionally, users can treat the first chunk as the page header, using it to store metadata, a practice common in many B+Tree implementations.

B. SiM Command Format

SiM's search command consists of the target page address and two 64-bit arguments: 1) a query key and 2) a mask. The mask facilitates the comparison of specific bit ranges, ignoring other positions as "don't care." In SiM-indexed relational database tables, where each row corresponds to an 8-byte key and data columns are encoded at specific bit ranges, the mask aids in isolating a specific column for matching. Fig. 5 demonstrates this by encoding rows into 8-byte data and querying based on the gender value, while masking unrelated columns. This command format flexibility enables SiM to support diverse queries through the *BitWeaving* technique [12], which is widely used in database systems to enable high parallelism.

SiM's gather command resembles the gather SIMD instruction for the CPU: it uses a 64-bit index bitmap to indicate the desired chunks within an index page to read (a page contains 64 chunks). Compared to transmitting the entire page, the gather command can significantly reduce the volume of I/O transmission.

C. Storage and Match Mode

SiM ensures compatibility with existing flash memory chips and preserves their high-density storage functionality by introducing minimal additional hardware. It operates in two modes: 1) *Match Mode* and 2) *Storage Mode*. A flash memory page can function in both modes, but their interpretations differ.

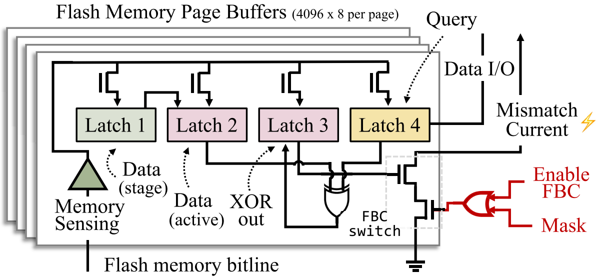


Fig. 6. PB (extension to the existing structure marked in red).

In *Storage Mode*, the flash memory chip is solely responsible for storing data. It does not interpret the page content. This mode emphasizes high storage density and I/O bandwidth. Consequently, it typically stores multiple bits per memory cell, and the I/O bus operates at a high clock rate.

In contrast, *Match Mode* prioritizes efficient data retrieval. It stores only one bit per cell (SLC) to ensure data reliability, and the I/O bus operates at a lower clock rate. This mode does not compromise latency because on-chip matching significantly reduces the amount of data transfer required.

SiM dynamically switches between the two modes based on operational requirements. It utilizes *Match Mode* for foreground indexing operations, taking advantage of its efficient data retrieval capabilities. On the other hand, it employs *Storage Mode* when writing new data and performing background maintenance, leveraging its high storage density and I/O bandwidth.

IV. Implementation

A. Extending Existing Circuit

Each NAND flash memory plane contains a set of PBs, each associated with a memory bitline, for reading a page from the memory array. Fig. 6 illustrates the typical structure of a NAND flash memory PB, equipped with multiple data latches[2] and an XOR gate.

SiM utilizes the XOR gate for bit matching, in conjunction with the failed bit counting (FBC) circuitry.[3] Query key is loaded into Latch 4 and XORed with the memory content stored in Latch 2. The XOR result is stored in Latch 3, where a one-bit signifies a mismatch. SiM's core data unit, including its query key size and mask size, is 8 bytes (or 64 bits) to align with the FBC's PB group structure, where every 64 bitlines form a match group. A nonzero count in a PB group indicates a mismatch. Moreover, we add an OR gate to each PB. This allows reading from Latch 2 either when FBC is activated during data programming in *Storage Mode* or when the current query's mask bitmap has an active bit in the respective bit

[1] Throughout the rest of this article, we use 4 KiB as the logical page size.

[2] A data latch stores a single bit. Encoding and decoding 3-bit storage require three latches and an XOR gate.

[3] SSDs store data by injecting electric charges into flash memory cells until they reach a predetermined charge level. After every program operation, the cell states are verified and recorded in Latch 3. A one-bit denotes a mismatch, releasing a small current. The FBC sums these currents, determining if the misprogrammed cells exceed a set limit. Every 64 PBs are grouped and all currents from the group's PBs are combined using an analog counter, with the current magnitude indicating the count value [13], [14].

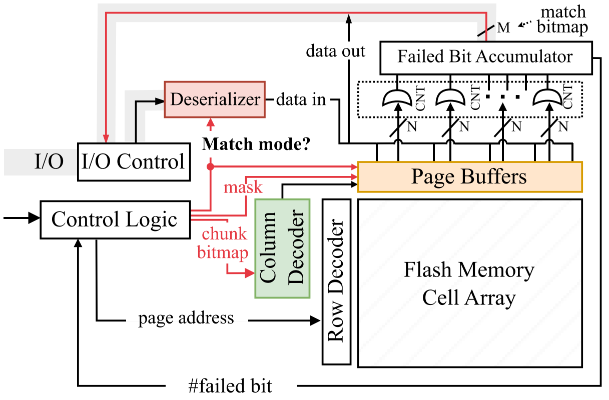


Fig. 7. SiM's chip-level design.

position in *Match Mode*. Fig. 7 shows SiM's chip design, incorporating a new signal, `match mode`, to switch between *Storage Mode* and *Match Mode*.

B. On-Chip Matching Workflow

The controller initiates on-chip matching using the `page-open` command, which specifies the target page address. Upon receiving this, SiM loads the target page from flash memory into Latch 1. Since SiM permits simultaneous memory reading and bit matching from a previous round, active data from the previous round might still be in Latch 2. If so, the newly read data is held in Latch 1 until the `page-close` command moves it to Latch 2. The next round then begins with *Optimistic Error Correction* for data integrity (refer to Section IV-C2).

After the initialization, SiM can receive multiple `search` command for batch matching. Each `search` command activates the deserializer to duplicate and forward the 64-bit query key to Latch 4 of each PB. In the next clock cycle, Latch 2 and Latch 4 contents are XORed, and the result is stored in Latch 3. A replicated 64-way mask signal, representing the 64-bit mask of the query, is linked to every PB. If both the XOR result in Latch 3 and the mask signal are active, a small current flows through the FBC switch, signaling a *mismatch*. The FBC's analog counter then aggregates the 64 match signals. If there is a mismatch, it emits a nonzero value, which is identified using a 1-bit voltage comparator. A bitmap of size $M = 512$, denoting match results from M PB groups (with each group having 64 PBs), is generated. These results are stored in latches for synchronization and later transferred to the I/O bus.[4]

SiM performs a `gather` command as follows. First, the target page is loaded from the flash memory into the L1 latch. Next, the column decoder deserializes the 64-bit index bitmap, converting it into the entire page, and then sequentially transmits the selected chunks onto the I/O bus. It is common for a `search` command to be immediately followed by a `gather` command. In such cases, since the page content is already loaded into the PBs, the `gather` command can initiate data transmission without delay.

C. Data Integrity

1) Data Randomization: In modern SSDs, it is a common practice to randomize the stored data to ensure data reliability. This randomization process involves XORing the data bits with a deterministically generated random bit stream, which is derived from a seed determined by the page address. When reading a page, the data is de-randomized using the same procedure to recover the original data values. In SiM, the query key is randomized within the *deserializer* using the same seed that was used to randomize the target page. Since the random stream is canceled out when XORed twice, we can perform bit matching in the PB without de-randomizing the target data page. Unlike conventional randomization, we initialize the seed for each chunk using the chunk address. This enables us to de-randomize noncontiguous chunks in the `gather` command.

2) Optimistic Error Correction: In order to perform on-chip matching without transmitting the full page to the SSD controller, we adopt an optimistic approach of sampling a few bytes at the beginning of the page for errors. This approach is based on two rationales. First, a recent work in in-flash computing [16] has characterized real chips and found that the SLC pages we adopt, which store one bit per cell, exhibit no errors for extended periods of time. Second, another recent work has demonstrated the feasibility of sampling a portion of a page to determine its overall stability [17].

Our optimistic approach is as follows. Before writing a logical page to the flash memory, we prepend a verification header to verify data integrity during subsequent page reads. This verification header includes the current timestamp and an 8-byte predetermined magic number. Additionally, we prepend an 8-byte CRC checksum calculated over the first chunk and the two aforementioned fields.

When the `page-open` command loads the page content from the flash memory, both the verification header and the first chunk are transmitted to the controller. The controller verifies the chunk using the CRC checksum. If a mismatch is detected, the controller initiates a full page read to retrieve the entire page from the PB. The page is then processed by a dedicated ECC chip, similar to a normal page read. If an uncorrectable error is detected, the controller adjusts the sensing voltage using the magic number and performs read-retries up to a specified maximum number of times [17].

Our optimistic error correction approach optimizes the common case of error absence in SLC pages while providing a fallback solution for corner cases. Additionally, if the age of the page, indicated by the write timestamp in the verification header, exceeds a safety margin, the page is also read out for error correction and placed in a refresh queue to be rewritten at a later time, ensuring data reliability.

3) Concatenated Error Correction: In addition to the verification header, we also assign a 4-byte ECC parity to each

[4] Unlike normal data transfer, which sends approximately equal numbers of zero and one bits, the bitmap from the SiM chip mostly comprises zero bits due to the typically low number of matches. This sparsity reduces power consumption during data transmission over modern I/O bus protocols operating in *Low-Tapped Termination*, like NV-LPDDR4 [15], which consumes power only when transmitting one bits.

chunk, which is checked in the controller upon loading. The chunk-level ECC is stored alongside the page-level ECC parity. This arrangement forms a *concatenated code*, a classic technique for enhancing data reliability [18]. In our case, this arrangement enables the `gather` command to perform fine-grained error correction without the need to load the entire page to the controller.

D. Hardware Overhead

We add the mask signal to each PB to control the FBC switch in match mode and an `OR` gate to enable the FBC in data programming in storage mode. We also modify the column decoder to transmit specific chunks within a page and adjust the deserializer to distribute the input data across all page bits. Given that modern NAND flash chips support reading specific portions of a page (i.e., *Random Data Out*) and generate test data patterns for reliability tests [19], our modifications to the column decoder and deserializer are minimal. Considering the PB and decoder account for under 9% of the total chip area [11], we estimate that SiM adds around 3% to the overall area overhead.

E. Batch Matching

SiM offers the capability of batch matching to maximize the utility of a page read from the flash memory (the page read latency accounts for the largest portion in the overall on-chip matching process, so conducting multiple matching can amortize the page read latency). We implement a deadline-based command scheduler to evaluate the effectiveness of this approach. Each command is associated with a deadline upon submission. The scheduler holds the submitted commands in a queue until their respective deadlines expire. At that point, the scheduler searches for other commands in the queue that target the same page and submits them together as a batch. We evaluate the scheduler in Section VII-E.

V. System Integrations

This section demonstrates how SiM's versatile interface makes it possible to integrate it into various data-intensive systems.

A. Database Primary Index

The *Primary index* in a relational database maps the primary key of a table to a pointer indicating the storage location of the corresponding data row. It is usually implemented with a B+Tree, as shown in Fig. 8. The internal nodes of the B+Tree can usually fit within the DRAM, while the leaf nodes often require on-demand reading from disk [20]. A leaf node page typically begins with a header that stores metadata, including a validity bitmap, counters for empty slots, compression information, and sibling pointers. Following the header is a compact array of keys and values.

The arrangement of keys and values within the leaf nodes is essential to efficient search of query key on the CPU. For instance, keys can be stored contiguously or sorted to facilitate SIMD parallel search and binary search, respectively. The leaf nodes can be directly stored in SiM, effectively replacing the process of on-demand disk I/O and in-CPU search with a search command to SiM. A leaf node can span multiple SiM pages. For example, an 8-KiB leaf node can store its key array in one SiM page and its value array in another SiM page. A leaf node search involves a search command that targets the first page, followed by a gather command that targets the second page. These two commands can be internally pipelined to reduce latency. Storing keys and values separately, as opposed to storing them in the same page, increases the parallelism of key search and prevents unnecessary loading of the value when a key is not found.

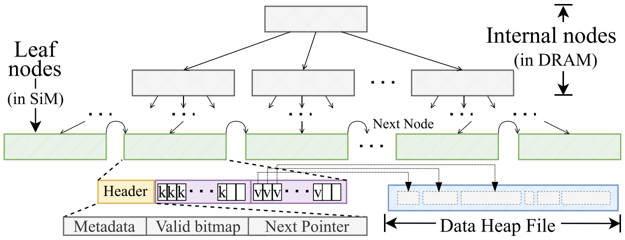


Fig. 8. SiM-enhanced database primary index.

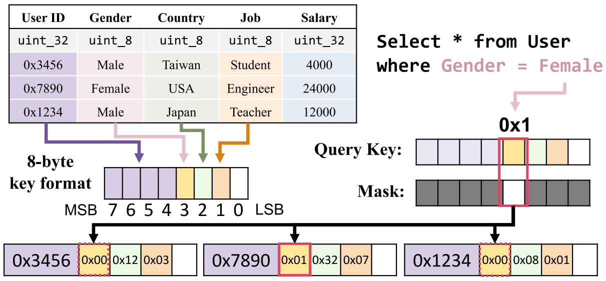
TABLE I
SiM- Versus Non-SiM-Based Primary Index

	SiM	Without SiM
Total I/O	128 B	8192 B
Bus Freq	40 MHz	1600 MHz [21]
Current	11 mA [22]	152 mA [2] (13x)
Energy	63 nJ	1400 nJ (22x)
Latency	3.2 μs	5.1 μs (1.6x)

Table I presents a back-of-the-envelopment comparison of the worst-case energy consumption and latency in data transfer between a conventional disk-based B-Tree and an SiM-based B-Tree. The comparison focuses solely on the data transfer from the flash memory chip's PB to the SSD controller, excluding transfer to the host OS. In the absence of SiM, the entire key and value pages must be read, resulting in an I/O size of 8 KiB. However, with SiM, the `search` command sends a 64-byte bitmap, while the `gather` command sends a 64-byte chunk. The flash chip operates in *Match Mode* with a bus clock frequency of 40 MHz, whereas without SiM, the clock frequency defaults to 1600 MHz for higher bandwidth. Consequently, the peak current of the high-speed bus is thirteen times greater than that of the low-speed bus. Energy consumption is 22 times higher without SiM. However, the latencies of the two approaches are comparable. This comparison demonstrates that SiM's data reduction improves energy efficiency and performance due to enhanced *goodput*.

B. Database Secondary Index

A secondary index is a data structure in a database that maps the values of one or multiple columns to the primary key of a table, enabling efficient retrieval of rows based on specific column values without the need for a full table scan. Fig. 9 illustrates an example user table with its secondary index stored on SiM. Following the key encoding scheme used in MySQL [23], each row is transformed into an 8-byte key,



User ID	Gender	Country	Job	Salary
uint_32	uint_8	uint_8	uint_8	uint_32
0x3456	Male	Taiwan	Student	4000
0x7890	Female	USA	Engineer	24000
0x1234	Male	Japan	Teacher	12000

```
Select * from User
where Gender = Female
```

Fig. 9. Example encoding of a table for SiM.

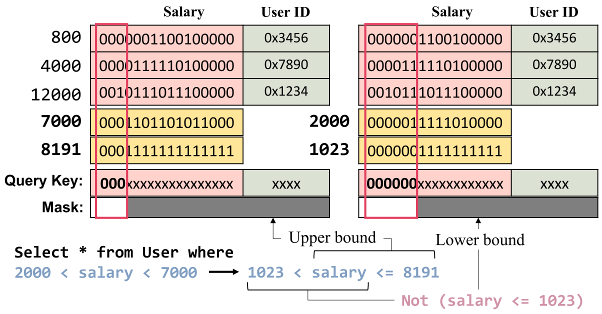


Fig. 10. Example usage of SiM: filtering a secondary index.

and the encoded keys are stored compactly in a SiM page. To perform a query that retrieves all female users, the target value (e.g., female represented by `0x01`) is encoded into the query key, and a mask is constructed based on the position of the target column. SiM produces a match bitmap, allowing us to retrieve the user IDs of the matched keys using a `gather` commandover the same page.

C. Database Range Queries

A range query in a database table aims to find all k such that $U > k \geq L$. SiM narrows the search space in two steps. First, it decomposes the range query into upper-bound and lower-bound queries. The upper-bound query $U > k$ is transformed into $2^{\lceil \log_2(U) \rceil} - 1 > k$, where $2^{\lceil \log_2(U) \rceil}$ corresponds to the smallest value larger than U that is a power of two. The lower-bound query $k \geq L$ is processed by transforming it into an upper-bound query of "$k < L$" and then applying a bitwise NOT operation to the obtained bitmap. The final result of the range query is obtained by performing a bitwise AND operation between the two subqueries.

Fig. 10 illustrates a secondary index for finding users with a specific range of the salary column in the same table. The salary and user ID are encoded in big endian into an 8-byte key, resembling the bit-sliced index used in analytical databases [24]. The range query "`select * from User where 2000 < salary < 7000`" is decomposed into upper-bound and lower-bound queries. In Fig. 10, the upper-bound query is transformed into "`salary <= 8191`." By examining the most significant bits, we determine that the 0th to 2nd bits of both 7000 and 8191 are zero. Any integer with the 0th to 2nd bits being zero is guaranteed to be smaller than 8191. The 0th to 2nd bits in the query key are set to zero while masking out the rest of the bit positions. The search returns the bitmap `110` because 800 and 4000 satisfy the conditions. The lower-bound query is transformed into "`Not (salary <= 1023)`," which returns the bitmap `011`. Combining the two subqueries give the final result: `010`.

Although the result encompasses more elements than the actual range, it effectively reduces the search space for subsequent fine-grained filtering. We make the design choice of only implementing exact equality matching over exact range search because this allows us to repurpose existing circuitry in the hardware implementation without the need for additional circuits, as further discussed in Section IV-B. There is no bit dependencies in exact equality matching. Thus, the matching can be finished in one pass, and the interbit wiring cost can be saved.

Nevertheless, users have the flexibility to conduct multipass comparisons to achieve their desired level of confidence. This can be achieved by masking out the previously compared MSB bit region and recursively compare the masked-out number. Furthermore, in the field of data analytics, precise results are often unnecessary. When the keys are uniformly distributed, our approximate range query can have low error rates [24].

D. Redistributing Data

Many data structures used in disk-based systems partition and redistribute data to improve performance. For instance, when a B-Tree or extendible hash table becomes full, it splits a full node or bucket into two. LSM-Trees perform compaction when a level reaches its capacity. Log-structured data structures require periodic garbage collection to free up space. In database systems, the join operation combines data from multiple tables using a hash table, which necessitates partitioning the dataset to ensure efficient data access during queries. These operations involve reading data from disk and rearranging them in memory. Data redistribution can result in high temporary memory usage and CPU spikes, especially in log-structured storage where data for the same partition can be scattered across multiple files. This can cause significant performance issues for frontend user services. Because of its significance, there have been calls for specialized hardware acceleration [25]. However, with SiM, data redistribution can be performed incrementally by keyspace partitioning. Partitioning the key space using a specific bit slice from the key, similar to a radix tree, allows us to locate a particular partition using the `search` command and collect the data using the `gather` command. By gathering one partition at a time, we can avoid loading data that do not belong to the specific partition, effectively reducing the I/O and memory overhead.

VI. Experiments

A. Experimental Setup

1) Hardware: We implement the `search` command and `gather` command by defining two new NVMe commands. We encode the SiM-specific payloads in NVMe's vendor-specific dataset management (DSM) opcode and extend NVMe's kernel driver to parse the new command formats.

TABLE II
HARDWARE PARAMETERS

3D NAND flash chip parameters	
(Channel, Package)	(8, 1)
(Die, Plane, Block, Page)	(2, 1, 32, 128)
(Read, Program, Erase)	(16 µs, 80 µs, 1 ms)
(Flash Page Size, Cell type)	(4 KiB, SLC)
SiM Clock Cycle	10
SiM Clock Frequency	33 MHz
External I/O	
Interface	PCIe Gen 3
Bus Width	128
Bus Clock	250MHz
Internal I/O Bus	
Interface	NV-DDR3 (ONFi 4.x)
Bandwidth (Match mode)	80 MT/s
Bandwidth (Storage mode)	800 MT/s
Bus Width	8
Power Settings	
(Bus Voltage, NAND Voltage)	(1.2V, 3.3V)
Bus Active / Idle Current	5mA / 10uA
NAND Read/Program Current	25mA, 25mA
SiM Current	2.5mA

TABLE III
QUERY CONCENTRATION IN DIFFERENT DISTRIBUTIONS

	1st	2nd	3rd	4th
Uniform	0.03%	0.03%	0.03%	0.03%
Skewed ($\alpha = 0.5$)	0.23%	0.12%	0.11%	0.07%
Very Skewed ($\alpha = 0.9$)	17.00%	2.54%	1.53%	1.08%

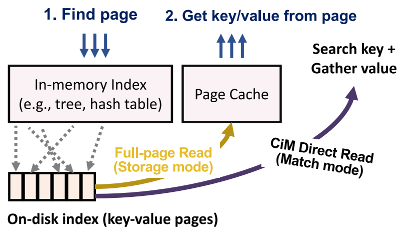

1. Find page
2. Get key/value from page
Search key + Gather value
In-memory Index (e.g., tree, hash table)
Page Cache
Full-page Read (Storage mode)
CiM Direct Read (Match mode)
On-disk index (key-value pages)

Fig. 11. SiM and baseline setup in the experiment.

We prototyped and simulated SiM on Amber [26], a high-fidelity SSD emulator. Table II shows the hardware parameters we used. The I/O bandwidth for SiM's match command is configured 80 MT/s (NV-DDR3's timing mode 1), which is 10% of the typical bandwidth for full-page I/O. This setting can lower the I/O bus's operation current[5] and the peak power. Thanks to the reduced I/O volume, the latency and the energy is minimally effected. I/O requests are scheduled in a first-come–first-serve manner, but a deadline-based scheduler is also evaluated in Section VII-E.

2) Data Structure: We create a generic index that consists of an in-memory top-level index and a collection of disk pages, each containing a compact array of key–value pairs. The top-level index maps a key to its on-disk page, as shown in Fig. 11. If it is implemented as a B-Tree, the on-disk pages correspond to the leaf nodes. If implemented as a hash table, on-disk pages correspond to hash buckets. The on-disk page is then loaded into the operating system's page cache, from which the value can be searched. The on-disk index is set to 650 MiB, taking 65% of the simulated SSD's capacity. We ensure that there is enough spare space to prevent SSD space reclamation, allowing for a more focused evaluation.

3) Baseline Setup: Our baseline is the traditional CPU-centric architecture, which reads entire pages from disk and stores frequently accessed pages in the page cache. SiM, on the other hand, bypasses the page cache by sending a `search` command to the on-disk page address of a key specified by the in-memory index, determining the key's position in the page, and retrieving the desired values using a `gather` command. As we will see later, bypassing the page cache effectively frees it up for other uses, such as write buffering. Thanks to the small transmission size, SiM communicates with the host OS entirely through NVMe's command interface (i.e., MMIO) and bypass the conventional DMA procedures. Note that this article lacks direct comparison with ParaBit [27] and CoX-PM [11] due to differing application scenarios and difficulties in accurately reproducing their proprietary environments.

4) Workloads: We customize the Yahoo! Cloud Serving Benchmark (YCSB) [28], [29] and subject the index to various query distributions and read/write patterns to evaluate it across various application scenarios. Using Linux's CGroup, we downscale the page cache size to various ratios of the on-disk index size. The term *Cache Coverage* refers to this ratio. For example, a Cache Coverage of 50% indicates that the page cache size is 325 MiB, which is 50% of the on-disk index size (650 MiB). A Cache Coverage of 0% indicates that caching is disabled.

We begin collecting statistics only after the initial data has been loaded into the SSD and the workload has run for 30% of its designated length to ensure the system reaches steady state. We disable periodic cache flushing of dirty pages to better understand the systems' sensitivity to varying cache sizes.

5) Query Distribution: Table III illustrates query concentration across different distributions. In many online services, it is common for a small number of queries to dominate the workload. This phenomenon is modeled using Zipf's distribution for both skewed ($\alpha = 0.5$) and very skewed ($\alpha = 0.9$) scenarios, alongside a uniform distribution for comparison. The uniform distribution shows an even spread, whereas the very skewed distribution ($\alpha = 0.9$) shows a significant dominance of the top queries, with the most frequent accounting for 17% of the total workload.

VII. RESULTS

A. Overall Speedup

Fig. 12 displays SiM's overall speedup in terms of query per second (QPS)[6] compared to the baseline. The y-axis represents the varying percentage of read requests in the workload. 100% indicates a completely read-only workload, while 20% indicates a write-intensive workload. The x-axis depicts different cache coverage. 0% disables the page cache and directs all I/O to the SSD. 10% and 25% would be the typical configuration for real-world system to balance

[5] While higher timing modes typically incur higher operational current as in Table I, we set the I/O bus current consumption of the baseline also to 5mA assess the inclusion of advanced power optimization [15].

[6] A workload's QPS is a measure of its throughput calculated as the number of queries divided by the completion time (excluding the first 30% of queries designated as warmup period).

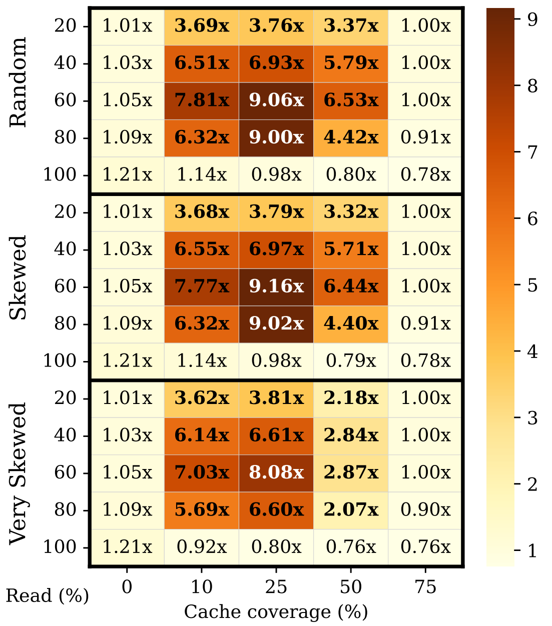


Fig. 12. SiM's query-per-second speedup over baseline.

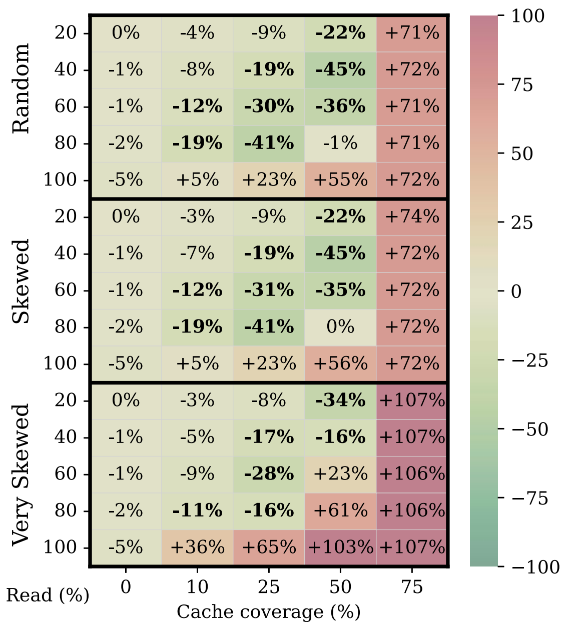


Fig. 13. SiM's energy consumption over baseline.

performance and hardware cost. We make several observations from Fig. 12.

1) The baseline supported with cache performs 8%–20% better than SiM in read-only workloads. This is to be expected, given that SiM bypasses the page cache and requires additional cycles for on-chip matching, whereas the baseline may avoid I/O by searching the pages stored in the page cache directly. One possible solution is to send search commands to the same page in batches to amortize the latency of reading from NAND flash memory—a technique evaluated in Section VII-E. Another option is to cache the retrieved keys. This fine-grained management can make better use of cache space than traditional page-level caching, but it can significantly complicate index designs.
2) SiM outperforms 3×–9× in write-intensive workloads. This is because SiM does not use read caching, so the cache can be used for write buffering. Because writes are significantly more expensive than reads on SSDs, increasing write buffering can improve overall performance and extend SSD lifespan. This is consistent with the design of many modern database engines, such as RocksDB, where read requests bypass page cache to avoid prematurely evicting dirty pages from cache.
3) When cache coverage is zero, all I/O goes directly to the SSD, and the locality difference in query distribution has no effect on performance. When cache coverage is high (75%), SiM has few performance advantages over the baseline because the cache is large enough to absorb page updates that would otherwise be evicted under low cache coverage.

B. Energy Consumption

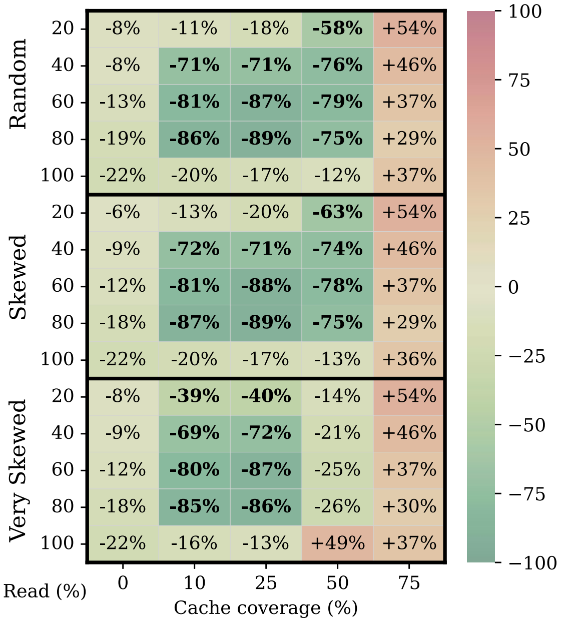
Fig. 13 compares SiM's energy consumption with the baseline. This analysis favors the baseline because it only considers the energy consumption of the NAND flash chip, ignoring the energy consumption of the CPU and DRAM, which are difficult to accurately characterize. It also equalizes the baseline's bus I/O current consumption with SiM's to incorporate recent power optimization for high-frequency I/O bus [15]. Even with these assumptions, SiM still reduces energy consumption by 10%–45% at typical cache coverage levels (10%–50%). A cache coverage of 75% is only a reference as it does not account for the significant DRAM energy consumption required to provide a large memory space. SiM's ability to lower write traffic is what accounts for the lower energy use, as further explored in Fig. 16(a).

SiM's ability to reduce read I/O also contributes to energy savings. In contrast to the baseline, which sends complete key and value pages (each 4 KiB) to the host OS via the PCIe bus, SiM only transmits the result bitmap (64 B) from the key page and the necessary chunk (64 B) from the value page for a random point query, where only one chunk is needed. This strategy decreases data transmission over the PCIe bus by 64 times. In the internal I/O bus, SiM needs to transfer another 256 B for integrity verification upon page open, but this still reduces I/O by 21 times. This is why, despite using a 10-times slower bus timing mode, SiM can reduce I/O transmission delay and bus active time by 2.1 times.

SiM's I/O reduction lowers queuing delays and shortens the SSD's active period. These benefits effectively offset the additional energy consumed by SiM for on-chip matching operations.

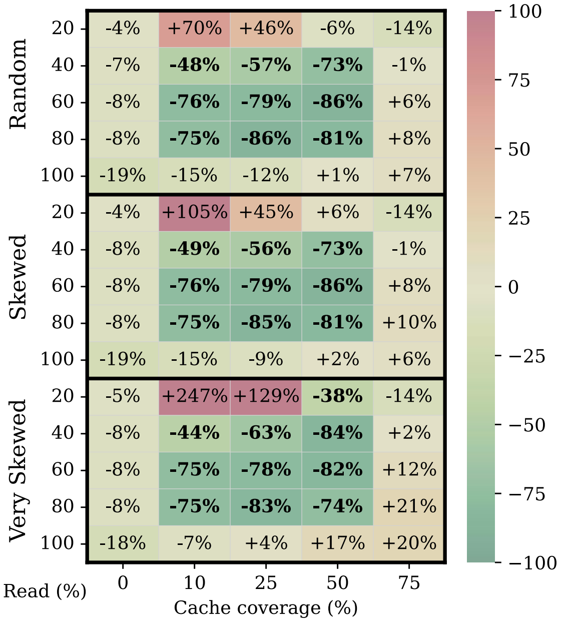
C. Read Latency

Fig. 14 compares SiM's median read latency reduction to the baseline. This reduction varies from 30% to 89% across workloads, whether skewed or uniformly distributed. Note that this analysis inherently favors the CPU-centric baseline, as



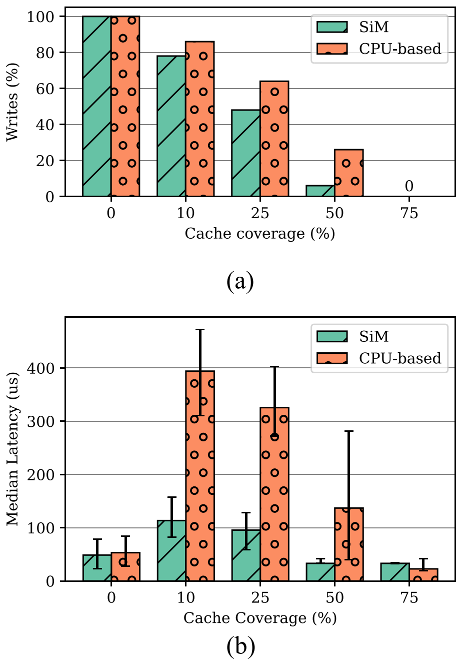
	Read (%)	0	10	25	50	75
Random	20	-8%	-11%	-18%	**-58%**	+54%
	40	-8%	**-71%**	**-71%**	**-76%**	+46%
	60	-13%	**-81%**	**-87%**	**-79%**	+37%
	80	-19%	**-86%**	**-89%**	**-75%**	+29%
	100	-22%	-20%	-17%	-12%	+37%
Skewed	20	-6%	-13%	-20%	**-63%**	+54%
	40	-9%	**-72%**	**-71%**	**-74%**	+46%
	60	-12%	**-81%**	**-88%**	**-78%**	+37%
	80	-18%	**-87%**	**-89%**	**-75%**	+29%
	100	-22%	-20%	-17%	-13%	+36%
Very Skewed	20	-8%	**-39%**	**-40%**	-14%	+54%
	40	-9%	**-69%**	**-72%**	-21%	+46%
	60	-12%	**-80%**	**-87%**	-25%	+37%
	80	-18%	**-85%**	**-86%**	-26%	+30%
	100	-22%	-16%	-13%	+49%	+37%

Fig. 14. SiM's median read latency reduction.



	Read (%)	0	10	25	50	75
Random	20	-4%	+70%	+46%	-6%	-14%
	40	-7%	**-48%**	**-57%**	**-73%**	-1%
	60	-8%	**-76%**	**-79%**	**-86%**	+6%
	80	-8%	**-75%**	**-86%**	**-81%**	+8%
	100	-19%	-15%	-12%	+1%	+7%
Skewed	20	-4%	+105%	+45%	+6%	-14%
	40	-8%	**-49%**	**-56%**	**-73%**	-1%
	60	-8%	**-76%**	**-79%**	**-86%**	+8%
	80	-8%	**-75%**	**-85%**	**-81%**	+10%
	100	-19%	-15%	-9%	+2%	+6%
Very Skewed	20	-5%	+247%	+129%	**-38%**	-14%
	40	-8%	**-44%**	**-63%**	**-84%**	+2%
	60	-8%	**-75%**	**-78%**	**-82%**	+12%
	80	-8%	**-75%**	**-83%**	**-74%**	+21%
	100	-18%	-7%	+4%	+17%	+20%

Fig. 15. SiM's tail read latency reduction.



(a)

(b)

Fig. 16. Detailed comparison at 40% Read, Random Dist. (a) Amount of writes relative to no caching. (b) Median read latency.

we omit the CPU's search time for locating the target key after key pages have been loaded into host OS's memory. In contrast, for SiM, we include the latency incurred by on-chip matching operations. Despite this discrepancy that could advantage the baseline, SiM still demonstrates superior latency improvements.

In read-only workloads, SiM outperforms the baseline particularly when the baseline is allocated less cache. This can be attributed to the longer I/O transmission of the full page transfer. Fig. 16(b) zooms in on the comparison of median read latencies between SiM and the baseline under a random query distribution and a 40–60 read–write workload. Here, error bars denote the 25th and 75th percentiles, with SiM demonstrating narrower error bars. This suggests a more consistent response time, which is crucial for services directly interacting with users.

In write-intensive workloads, SiM has lower read latency than the baseline in mid-range cache coverage where the write set size exceeds the cache capacity. In this case, new writes can evict both clean and dirty pages. Clean page eviction degrades read performance due to cache misses, whereas dirty page eviction causes lengthy queueing delays for read operations. SiM's cache bypass strategy, as discussed in Section VII-A, alleviates this effect.

D. Tail Read Latency

Fig. 15 presents the tail read latency (99th percentile) improvements SiM achieves over the baseline. Although the variability between the 25th and 75th percentile latencies is less for SiM, in rare cases, SiM may exhibit significantly higher latency compared to the baseline, particularly in workloads where read requests are infrequent and highly skewed. Closer examination reveals differing write patterns: the baseline experiences consistent write activity, whereas SiM may face sporadic peaks in write demand. This is attributed to SiM's page cache being primarily composed of dirty pages from data writes. Consequently, initiating a new write could trigger a chain reaction of writing back dirty pages, potentially delaying read requests substantially. In contrast, the baseline system's page cache contains some clean pages fetched from the SSD, which can be evicted immediately to buffer data writes, avoiding such corner cases.

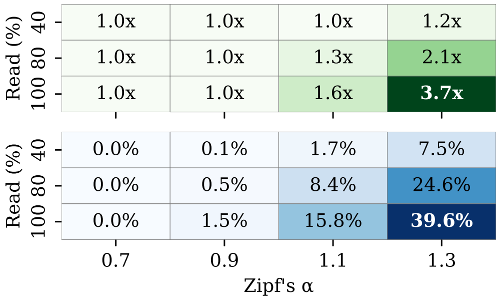


Fig. 17. QPS speedup of batch submission (top) and merge probability (bottom).

To mitigate this issue, implementing an I/O scheduler that gives priority to reads over writes could prevent read starvation. Alternatively, preempting writes in favor of reads—a strategy proposed for ultralow-latency SSDs [30]—could also be effective. Future research should explore replacing our current first-come–first-serve I/O scheduling with more sophisticated strategies to assess their impact on reducing tail latency.

E. Batch CiM Submission

Section IV-E introduces a deadline scheduler aiming to reduce NAND flash memory's read latency by batching `search` command for identical pages. Each `search` command is assigned a deadline of 4 μs, which constitutes 25% of the 16-μs read latency for SLC memory. The upper heatmap of Fig. 17 presents the query-per-second improvement when using the deadline scheduler, compared to SiM's performance without it. The lower heatmap indicates the probability that a query will target the same page as another unexpired query in the scheduler. As the concentration of queries increases, indicated by a rising Zipf's α value, the probability of multiple queries targeting the same page increases, resulting in a 3.7-fold boost in throughput at $\alpha = 1.3$ for purely read-only workloads. However, such an α value is way beyond what a normal workload would exhibit. Setting a longer expiration time can also improve throughput, but at the expense of prolonged latency. We conclude that the deadline scheduler is ineffective for low-latency SSDs because the overhead outweighs the benefits.

F. Sensitive Analysis on Full-Page Read Ratio

While SiM excels in precise data retrieval, the need for full-page reads remains crucial. For instance, indices in read-heavy analytic databases require summing data across entire pages. Similarly, the write-optimized LSM-Tree index, while needing efficient support for random point queries, also necessitates compaction—a garbage collection process that reads indices in full pages for merging. This leads us to assess how variations in the volume of full-page reads affect overall performance across different query distributions and in both read- and write-dominant workloads. Fig. 18 illustrates the relative query-per-second speedup of SiM compared to the baseline where all reads are full-page (on the left-most side of the x-axis). Observe that as the proportion of SiM reads within the workload increases, so does performance. This effect is evident in both read- and write-dominant scenarios, though more markedly in the latter. On the other hand, the influence of varying query distributions on this trend appears minimal.

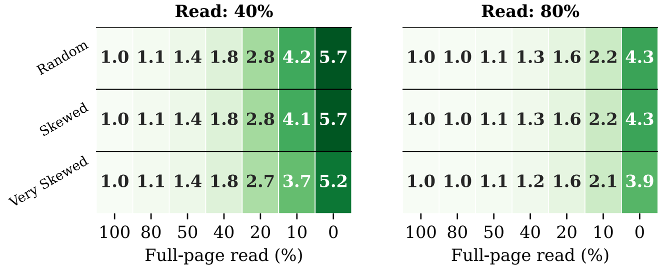


Fig. 18. QPS speedup versus full-page read ratio.

VIII. Related Works

Numerous research efforts have been made on minimizing data transfers through early data filtering, which can be broadly classified into near-storage computing approaches—such as SmartSSD or custom circuits attached to flash memory controllers [31]—and on-chip computing approaches like SiM. Near-storage computing reduces I/O between the host and SSD, whereas on-chip computing reduces data movement from within the SSD itself. On-chip approaches can be analog-based or digital-based. Analog approaches, such as Tseng et al. [9], are well suited for error-tolerant applications like machine learning but fall short for the precise data matching required in indexing. Digital approaches, like Parabit [27] and Flash Cosmos [16], use the existing flash memory sensing mechanisms for bulk bitwise operations, such as AND and OR across flash pages. SiM also utilizes the existing PB circuits but has a different programming model. Unlike Parabit and Flash Cosmos, where both operands are page-sized and must be preprogrammed into the same NAND block prior to the computation, SiM operates with a small query and a page for comparison, making it more efficient to deal with small, dynamically loaded operands.

CoX-PM [11] incorporates error correction and pattern matching circuits into the NAND flash memory. SiM, on the other hand, chooses not to perform on-chip error correction due to its complexity, instead relying on Optimistic Error Correction on SLC pages and the SSD controller's existing ECC chips. SiM also opts not to evaluate complex pattern matching in hardware, instead using software to decompose complex queries into elementary instructions that are cheaper to implement in hardware. ICE [10] integrates 8-bit integer multiplication into the peripheral circuits for on-chip vector matching. Unlike CoX-PM and ICE, SiM strives to repurpose existing circuits and minimize additional circuits to reduce hardware testing costs and accelerate adoption.

IX. Conclusion

This article introduced the SiM chip, a novel solution aimed at overcoming the bottleneck in data indexing through on-chip data matching. SiM introduces simple yet versatile commands for fine-grained data searching and gathering. These commands, despite their simplicity, enable complex, data-intensive operations found in various data structures to be accelerated. SiM's command structure allows for cost-effective implementation with minimal modifications to existing circuit designs. Furthermore, SiM can be combined with readily available high-capacity NAND flash memory chips to create a hybrid SSD that effectively realize the principle of data-metadata separation.

SiM has undergone extensive testing under a variety of workload and system constraints. Evaluation shows up to 9× speedup in write-intensive workloads and up to 45% energy savings due to reduced read and write I/O and better utilization of host's cache space. SiM reduces median and tail read latency by up to 89% and 85%, respectively. As a future work, we aim to integrate SiM technology into actual key–value and relational database systems to enhance their efficiency in garbage collection and range queries. Developing a hardware prototype is also planned.

References

[1] H. Zhang, D. G. Andersen, A. Pavlo, M. Kaminsky, L. Ma, and R. Shen, "Reducing the storage overhead of main-memory OLTP databases with hybrid indexes," in *Proc. Int. Conf. Manag. Data (SIGMOD)*, 2016, pp. 1567–1581.

[2] J. S. P. L. Yu and L. Pilolli, "Peak power management in a memory device," U.S. Patent 1 152 049 7B2, Feb. 2020.

[3] J. H. Yuh et al., "A 1-Tb 4-b/cell 4-plane 162-layer 3-D flash memory with 2.4-Gb/s IO interface," *IEEE J. Solid-State Circuits*, vol. 58, no. 1, pp. 316–328, Jan. 2023.

[4] L. Nubile et al., "Power management architectures for high performance NAND systems," in *Proc. IEEE Workshop Microelectron. Electron Devices (WMED)*, 2023, pp. 1–4.

[5] B. Kim et al., "28.2 A high-performance 1Tb 3b/cell 3D-NAND flash with a 194MB/s write throughput on over 300 layers i," in *Proc. IEEE ISSCC*, 2023, pp. 27–29.

[6] J. Im, J. Bae, C. Chung, and S. Lee, "PinK: High-speed in-storage key-value store with bounded tails," in *Proc. USENIX ATC*, 2020, pp. 173–187.

[7] Y. Kang et al., "Towards building a high-performance, scale-in key-value storage system," in *Proc. 12th ACM Int. Conf. Syst. Storage (SYSTOR)*, 2019, pp. 144–154.

[8] O. Mutlu, S. Ghose, J. Gómez-Luna, and R. Ausavarungnirun, "A modern primer on processing in memory," 2022, *arXiv:2012.03112*.

[9] P.-H. Tseng et al., "In-memory-searching architecture based on 3D-NAND technology with ultra-high parallelism," in *Proc. IEEE IEDM*, 2020, pp. 36.1.1–36.1.4.

[10] H.-W. Hu et al., "ICE: An intelligent cognition engine with 3D NAND-based in-memory computing for vector similarity search acceleration," in *Proc. 55th IEEE/ACM MICRO*, 2022, pp. 763–783.

[11] M. Chun, J. Lee, S. Lee, M. Kim, and J. Kim, "PiF: In-flash acceleration for data-intensive applications," in *Proc. 14th ACM Workshop Hot Topics Storage File Syst. (HotStorage)*, 2022, pp. 106–112.

[12] Y. Li and J. M. Patel, "BitWeaving: Fast scans for main memory data processing," in *Proc. ACM SIGMOD Int. Conf. Manag. Data*, 2013, pp. 289–300.

[13] Y.-h. C. T.-h. Kim, B.-S. Lim, and S.-W. Shim, "Non-volatile memory device and memory system including the same," U.S. Patent 20 180 075 918 A1, Dec. 2018.

[14] N. Choi and J. Kim, "Modeling and simulation of NAND flash memory sensing systems with cell-to-cell Vth variations," in *Proc. IEEE/ACM ICCAD*, 2020, pp. 1–8.

[15] J. Cho et al., "30.3 A 512Gb 3b/cell 7th-generation 3D-NAND flash memory with 184MB/s write throughput and 2.0Gb/s interface," in *Proc. IEEE ISSCC*, 2021, pp. 426–428.

[16] J. Park et al., "Flash-Cosmos: In-flash bulk bitwise operations using inherent computation capability of NAND flash memory," in *Proc. 55th IEEE/ACM MICRO*, 2022, pp. 937–955.

[17] Q. Li et al., "Shaving retries with sentinels for fast read over high-density 3D flash," in *Proc. 53rd Annu. IEEE/ACM MICRO*, 2020, pp. 483–495.

[18] G. D. Forney, *Concatenated Codes*. Cambridge, MA, USA: MIT Press, 1965.

[19] H. M. Cao, F. Liu, Q. Wang, Z. C. Du, L. Jin, and Z. L. Huo, "An efficient built-in error detection methodology with fast page-oriented data comparison in 3D NAND flash memories," *Electron. Lett.*, vol. 58, no. 12, pp. 483–485, 2022.

[20] G. Graefe, "Modern B-tree techniques," *Found. Trends® Databases*, vol. 3, no. 4, pp. 203–402, 2011.

[21] Open NAND Flash Interface (ONFI) Working Group, *ONFI 4.1 Gold Specification*, ONFI, Hillsboro, OR, USA, 2017.

[22] "YMTC Gen2 256Gb TLC Datasheet (Client rev0.2)," Data Sheet, YMTC Corp., Wuhan, China, 2019.

[23] Y. Matsunobu, S. Dong, and H. Lee, "MyRocks: LSM-tree database storage engine serving Facebook's social graph," *Proc. VLDB Endowm.*, vol. 13, no. 12, pp. 3217–3230, Aug. 2020.

[24] N. Potti and J. M. Patel, "DAQ: A new paradigm for approximate query processing," *Proc. VLDB Endow.*, vol. 8, no. 9, pp. 898–909, May 2015.

[25] T. Zhang et al., "FPGA-Accelerated compactions for LSM-based key-value store," in *Proc. 18th USENIX FAST*, 2020, pp. 225–237.

[26] D. Gouk et al., "Amber: Enabling precise full-system simulation with detailed modeling of all SSD resources," in *Proc. 51st Annu. IEEE/ACM MICRO*, 2018, pp. 469–481.

[27] C. Gao, X. Xin, Y. Lu, Y. Zhang, J. Yang, and J. Shu, "ParaBit: Processing parallel bitwise operations in nand flash memory based ssds," in *Proc. 54th Annu. IEEE/ACM MICRO*, 2021, pp. 59–70.

[28] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with YCSB," in *Proc. 1st ACM SoCC*, 2010, pp. 143–154.

[29] Y.-C. Chen, C.-F. Wu, Y.-H. Chang, and T.-W. Kuo, "ZoneLife: How to utilize data lifetime semantics to make SSDs smarter," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 42, no. 8, pp. 2488–2499, Aug. 2023.

[30] W. Cheong et al., "A flash memory controller for 15us ultra-low-latency SSD using high-speed 3D NAND flash with 3us read time," in *Proc. IEEE ISSCC*, 2018, pp. 338–340.

[31] D. Fakhry, M. Abdelsalam, M. W. El-Kharashi, and M. Safar, "A review on computational storage devices and near memory computing for high performance applications," *Memories Mater., Devices, Circuits Syst.*, vol. 4, Jul. 2023, Art. no. 100051.