FLIXR: Embedding Index Into Flash Translation Layer in SSDs

Gunjae Koo, *Member, IEEE*, Yunho Oh, *Member, IEEE*, Hung-Wei Tseng, *Member, IEEE*, Won Woo Ro, *Senior Member, IEEE*, and Murali Annavaram, *Fellow, IEEE*

Abstract—Flash memory technologies rely on flash translation layer (FTL) to manage no in-place update and garbage collection. Current FTL management schemes do not exploit the semantics of the accessed data. In this paper, we explore how semantic knowledge can be exploited to build and maintain indexes for stored data automatically. Data indexing is a critical enabler to accelerate many database applications and big data analytics. Unlike traditional per-table or per-file indexes that are managed separately from the data, we propose to maintain indexes on a per-flash page basis. Our approach, called FLash IndeXeR (FLIXR), builds and maintains page-level indexes whenever a page is written into the flash. FLIXR updates the indexes alongside any data updates at page granularity. The cost of the index update is hidden in the page write delays. FLIXR stores index data for each page within the FTL entry associated with that page, thereby piggybacking index access on a page access request. FLIXR accesses the index data in each FTL entry to determine whether the associated page stores data with a given key. FLIXR achieves 52.6% performance improvement for TPC-C and TPC-H benchmarks, compared to the conventional host-side indexing mechanism.

Index Terms—Solid-state drives, flash translation layer, database index, in-storage processing

1 Introduction

Big data analytics and database operations rely on efficiently finding records to speedup query processing. Index structures are employed for the purpose of finding records associated with a given key. For instance, without an index, the *select from*, SQL query needs to scan the entire database and select items that match the filter condition specified within the query. However, a database scan can be avoided if an index structure is already built on the keys. Then the *select from* operation can be implemented using a more efficient hashing of the filtering key to access the index table, which will in turn point to the records that match the key.

Index structures are generally created and maintained as separate structures from the data they index. If a server basically keeps the index structures in storage, their accesses need a large amount of I/O operations before reaching the desired data. Even with a multi-level index structure, such as a B+-tree, where the root nodes may be cached on the host DRAM, the remaining levels need to be accessed from storage. Therefore, host systems require additional latency to access the index structures. Also, when data is updated index tables must also be updated. Updating database tables causes significant indirect access to storage data and key comparisons that tax the host processors [1]. Prior work revealed that even if 0.1% of the data is updated, index updates can take 0.3×–5.2× of the original index build time, using B+-tree based index structures [2].

In this work, we make a case for improving storage performance by exploiting the data semantics. The key insight is derived from the fact that SSDs employ flash memory management schemes to support the disparate characteristics of read/write/erase operations of flash memory. Note that flash memory does not allow in-place writes thus the target page has to be erased before the updated data is overwritten. The erase operation in flash memory is significantly slow and may contaminate neighbor pages severely since the erasure requires a very high voltage level. SSDs write the updated page data to one of the empty pages in order to avoid such heavy performance burdens caused by the erase operations. That means every update on the same page creates *new* mapping to a different physical page. SSDs manage this page-level mapping information in a flash translation layer (FTL). As such, *every page access* has to go through FTL lookup in current SSDs. Note that the FTL management is agnostic to the semantics of the data stored in the page. Furthermore, the page/block sizes and the topology of the flash

- *Gunjae Koo is with the Department of Computer Science and Engineering, Korea University, Seoul 02841, South Korea. E-mail: gunjaekoo@korea.ac.kr.*
- *Yunho Oh is with the Department of Electrical and Computer Engineering, Sungkyunkwan University, Suwon 16419, South Korea. E-mail: yunho.oh@skku.edu.*
- *Hung-Wei Tseng is with the Department of Electrical and Computer Engineering, University of California, Riverside, CA 92521 USA. E-mail: htseng@ucr.edu.*
- *Won Woo Ro is with the Department of Electrical and Electronic Engineering, Yonsei University, Seoul 03722, South Korea. E-mail: wro@yonsei.ac.kr.*
- *Murali Annavaram is with the Ming Hsieh Department of Electrical and Computer Engineering, University of Southern California, Los Angeles, CA 90007 USA. E-mail: annavara@usc.edu.*

Manuscript received 12 March 2021; revised 10 January 2022; accepted 12 February 2022. Date of publication 25 February 2022; date of current version 13 December 2022.
This work was supported in part by Defense Advanced Research Projects Agency (DARPA) under Grant HR001117C0053, in part by NSF under Grants 1719074 and 1940048, in part by Samsung under Grant 079856, and in part by the National Research Foundation of Korea (NRF) grants funded by the Korea Government (MSIT) under Grants NRF-2021R1C1C1012172, NRF-2021R1G1A1094978, NRF-2021R1A2B5B01002932.
(Corresponding author: Murali Annavaram.)
Recommended for acceptance by A. Sivasubramaniam.
Digital Object Identifier no. 10.1109/TC.2022.3154602

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/

memory are divergent by flash memory vendors and generations and such information is sealed inside of SSDs. Consequently, the per-page index management that is compliant with SSD's flash management schemes can be offloaded into SSD in order to mitigate the heavy I/O transfers caused by conventional host-side index management methods. With this insight, we demonstrate how per-page indexing information can be embedded within the FTL to speedup database operations.

We propose *FLash IndeXeR (FLIXR)*, which allows the FTL management to parse per-page data to build and manage index information on a per-page basis automatically. Given that FTL is a performance-critical structure, we limit the type of indices that can be stored within each FTL entry to be integer or bitmap structures that occupy several bytes of extra space per FTL entry. This limitation inhibits the creation of complex index structures. However, we show that many database queries can be accelerated using these simple indices. Also, we showcase a co-existence of the proposed per-page index with traditional index structures that host processors manage, thereby allowing flexibility to the database administrators to use complex indices with FLIXR.

FLIXR has the following advantages over conventional index management schemes. First, the storage processors can update the index structures on the fly while the page is being updated. Since flash memory exhibits a long write latency, the index generation and update time can be easily hidden within the flash write process. Second, FLIXR can be applied to the commodity SSD platforms without additional hardware resources or significant firmware modifications. It is because FLIXR's in-SSD operations exploit the native address translation structures and data access processes implemented in the existing SSD firmware. Thus, FLIXR's indexing mechanism works efficiently on the commodity SSD platforms. Third, the data movement burden for reading index structures may also be reduced with FLIXR. To perform index-based filtering typically, the host processors fetch index data structures from the storage devices. As index structure size grows proportionally to database size (5–15% for B+-tree indices), the performance cost for moving index structures from storage to host DRAM cannot be ignored [2]. Previous studies tried to improve the performance of index operations (e.g., indexed scan) exploiting the operational characteristics of SSDs [3], [4]. However, the prior works still incur the overheads caused by I/O operations and index data transfers between SSDs and the host memory. Unlike the prior works, FLIXR can eliminate the I/O overheads and hide the latency caused by the indexed search and the index management.

To summarize, FLIXR utilizes the existing page-mapping tables in the FTL to automatically create and organize indices. FLIXR provides APIs to create programmer-defined indexing rules and a set of APIs to define index lookup operations to be offloaded to the SSD controller, thereby reducing index-related data movement between the host and SSD. This work implements several TPC-C and TPC-H queries that require data filtration and table join operations implemented within FLIXR. In our evaluation, FLIXR shows 52.6% performance improvement on query response time in data analytics workloads, compared to the conventional host-side indexing mechanism.

Followings are our contributions in this paper.

1) We propose FLIXR, an in-SSD indexing structure that enhances an SSD FTL to store page-level indices.
2) FLIXR provides a programming model that enables programmers to specify index creation rules, which are then executed on the SSD controller to create and update indices. The programming model also enables the SSD controller to access the index structure to automatically eliminate flash page accesses that are guaranteed not to contain the search key.
3) We implemented FLIXR on the OpenSSD prototype board and evaluated with critical database operations.

The remainder of this paper is organized as follows: Section 2 introduces the modern SSD platform architecture and FTL functions as background and discusses the related work. Section 3 discloses the criticality of large index structures using exemplar query functions. Section 4 describes the architecture and the programming model of FLIXR. Section 5 represents the evaluation platform and the benchmark applications studied in this paper. Section 6 presents the experimental results. Section 7 concludes this paper.

2 Background and Related Work

2.1 Modern SSD Platforms

Most modern SSDs contain several packages of NAND flash memory as non-volatile storage media. The smallest granularity for accessing flash memory is one page, which is 4-16KB. Multiple pages (64–256 pages) are grouped into a block. The read and write latencies of a NAND flash chip are tens of μs [5]. To communicate with the host system, modern SSDs may use NVM Express (NVMe) protocol [6], which is built on top of PCIe standard.

To process a large number of I/O requests and manage flash memory, modern SSDs equip a general-purpose multicore embedded processor to execute the SSD controller firmware. The firmware handles NVMe commands, data transfers between the host system and the NAND flash memory, manages FTL table, and performs garbage collection (GC) and wear-leveling (WL). Modern SSDs also provision GB-scale DRAM to cache FTL tables and hot pages to support fast accesses. Due to these advanced features, SSDs can finish NVMe operations within tens of microseconds [7], [8].

We measured the utilization of SSD controllers in several commodity NVMe SSDs. Our measurements revealed that the controller is idle for nearly 70% of the time, even with high levels of I/O parallelism. These results also concur with prior work [8], [9], showing that SSD embedded processors have sufficient slack to perform the basic indexing functions that we propose in this study.

2.2 Flash Translation Layer

The read/write process of flash is significantly asymmetrical. The data in flash cells is read or written at the page granularity. However, in the case of writing, it is possible only if the target page is *empty*. If not, the target page should be *erased* before a write. An erasure is performed on the whole block. Thus hundreds of pages in the same block should be erased before any page in the block is rewritten.

This erasure process is extremely slow compared to the read speed since it requires higher voltage and a longer time to reinitialize bit cells in a block. Hence, it is more efficient to read-modify-write a page to an empty page rather than erasing the block to update a page. That means the physical page address (PPA) of the data changes at runtime, and the PPA is not equal to the file system's logical block address (LBA). SSDs maintain the LBA-to-PPA mapping table, called the FTL table.

Other than the page-level FTL, prior work proposed block-level or mixed (hybrid) page-mapping structure to reduce memory space assigned to the FTL table [10]. In this paper, we describe FLIXR assuming a page-level FTL, but the FLIXR architecture can be implemented even in a block-level FTL with minimal modifications.

2.3 Related Work

Prior work revealed the criticality of indexing structures as huge volumes of datasets are maintained by modern database systems. Addressing this challenge, prior work proposed range-based storage page loads/histograms [2] and database systems based on log-structured merge trees (LSMTs) [11], [12], [13]. While the LSMTs support automatically ordering index and database tables, they still require a massive overhead in the host CPUs to manage the index structures. FLIXR maintains per-page indexing information embedded within SSD to reduce index lookup overheads. Since FTL access is required for all page accesses anyway, the cost of index lookup is negligible. Furthermore, depending on major operations in a database, FLIXR may be utilized with the traditional indexing scheme while mitigating the host-side computation overhead. For example, FLIXR can integrate LSMT structures with a simple interface as both employ the range-based indexing scheme. Prior work presented query processing on the distributed database systems as an important challenge [14]. FLIXR's index structures are easily scalable to distributed datasets since FLIXR maintains the indices for each SSD independently.

The concept of computing near storage appeared early, such as *active disk* [15], *active storage* [16], *Smart SSD* [17], [18], and *Willow* platforms [19]. These early in-storage processing ideas proposed the storage device architecture that includes general-purpose processors as powerful as the host processors. While in-storage computation is becoming more feasible as modern SSDs equip general-purpose embedded processors [20], the computation power of the storage processor is still much lower than that of the host CPUs. Another prior work proposed to limit the computation resources for in-storage computation depending on dynamic resource utilization [8]. Unlike prior work, FLIXR minimizes the in-storage computation overhead for index creation and comparison by exploiting the native SSD I/O procedure.

Several researchers exploit hardware logic to accelerate the specific data processing operations in storage. Jun *et al.* presented an FPGA-based solution to perform big data analysis in NAND flash storage [21]. Kim *et al.* presented a hardware engine to execute the scan and join functions in SSD [22]. *Biscuit* framework exploits hardware pattern matching units implemented on the flash channel paths to accelerate query processing [23]. *YourSQL* is a software framework that accelerates query processing by offloading the filtering operations to the SSD that equips hardware *pattern matcher* units [24]. Biscuit and YourSQL focus on accelerating query processing, not considering index update and maintenance. Unlike these approaches, FLIXR provides autonomous index management with minimal software modifications without custom hardware accelerators.

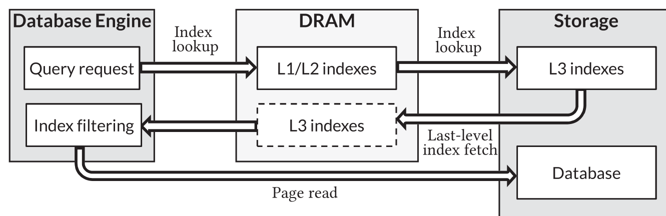


Fig. 1. Hierarchical indexing operation.

There are prior works that improve the performance of indexed scans exploiting the architectural characteristics of SSDs [3], [4], [25], [26]. The proposed ideas focused on that SSDs can exploit internal parallelism to achieve better throughput and page-based data management. While the ideas of these works are promising, they are orthogonal to the direction that FLIXR aims at. The fundamental objective of FLIXR is to show that the database indexes can be automatically managed by SSDs themselves, exploiting the computation power of SSD processors. The proposed new index management using in-storage computing schemes and characteristics of FTL is the main contribution that this paper argues. We believe that the optimization schemes that prior work proposed are applicable to FLIXR. We will investigate this topic in the next stage of this paper.

Providing semantic awareness to storage was explored recently in the context of graph analytics [27]. Prior work replaced FTL with a new graph translation table to enable faster graph accesses. FLIXR does not replace FTL, and instead, it just augments FTL entries with indexing bit vectors.

To summarize, FLIXR is an efficient in-storage indexing mechanism that works well on commodity SSD platforms. FLIXR exploits the native address translation structure in SSDs and lightweight index operations when the storage processor is idle. Furthermore, FLIXR framework does not require significant modification to the SSD firmware, thus it can be easily implemented on existing database software stacks. The detailed FLIXR approach will be presented in the following sections.

3 Criticality of Indexing

Data indexing is critical for accelerating big data analytics. Significant research has been expended on index structures as database management systems (DBMS) have recognized this critical need [2], [28]. However, index structures themselves have storage and latency overheads.

Consider B+-Trees and other multi-level index tables that are popularly employed in DBMS [29]. The operation of this index structure is illustrated in Fig. 1. With the hierarchical index structures, when a query has a filtering condition (for example *where* clause in SQL), the first (and second) level index in the DRAM will point to leaf nodes (third level index pages) that need to be accessed. These lower-level

```
select
  sum(l_extendedprice * l_discount) as revenue      ❸ Computation
from
  lineitem                                          ❶ DB table
where
  l_shipdate >= date ':1'
  and l_shipdate < date ':1' + interval '1' year    ❷ Filtering
  and l_discount between :2-0.01 and :2+0.01
  and l_quantity < :3;
```

Fig. 2. Scan and filtering in TPC-H Query 6.

indices are read from the storage to finally identify the target data pages that meet the filtering criteria. Consequently, even in the presence of a host DRAM to cache some of the levels of index tables, an indexing operation typically requires additional I/O accesses (to reach the lower-level indices) before finally reaching the data page. Note that even with proper caching policies such as storing recently used index pages, the problem of accessing I/O for index structure persists. Finally, the index must also be updated whenever the data is updated. Index updates are computationally expensive due to key comparisons and pointer chasing operations [1], [26].

To concretely demonstrate the above mentioned challenges, we describe how filtering and join processing operations in DBMS suffer from index access and maintenance overheads.

3.1 Scan and Filtering

Fig. 2 shows the SQL code of TPC-H Query 6 [30], which applies filtering conditions while scanning the *lineitem* table. The query defines the filtering conditions (❷) which are defined on multiple columns (*l_shipdate*, *l_discount*, *and l_quantity*). For the purpose of this discussion, let us optimistically assume that the database administrator has already created one primary index on *l_shipdate* and one secondary index on *l_discount* columns. Then, the database engine uses the primary index to winnow the records that fall within a shipping time window. After that, it uses the secondary index to identify the intersecting records that also match the stated discount criteria. Finally, these records are then streamed to the host CPU, which then further filters these records based on *l_quantity* before performing the computation (❸).

In this example, there are multiple challenges faced by the database administrator. First, the administrator must identify the primary and secondary keys for index creation. Second, every time the *lineitem* table is updated, the two indices must be updated as well.

3.2 Join Processing

Join processing is one of the essential functions of database engines that allows queries on multiple database tables that share a key [31]. Join processing usually requires repeated accesses to the index structures by multiple inter-related key values. Hence, the I/O traffic to index structures is significantly higher than simple filtering, as we demonstrate with an example below. While there are multiple approaches to perform join, we illustrate the problem using one approach.

```
select
  sum(CASE when p_type LIKE 'PROMO%'
          then (l_extendedprice * l_discount)       ❹ Computation
          else 0) /
  sum(l_extendedprice * l_discount) as revenue
from
  lineitem                                          ❶ DB tables
  part
where
  l_partkey = p_partkey                             ❸ Join
  l_shipdate >= date ':1'
  and l_shipdate < date ':1' + interval '1' month;  ❷ Filtering
```

Fig. 3. Join processing in TPC-H Query 14.

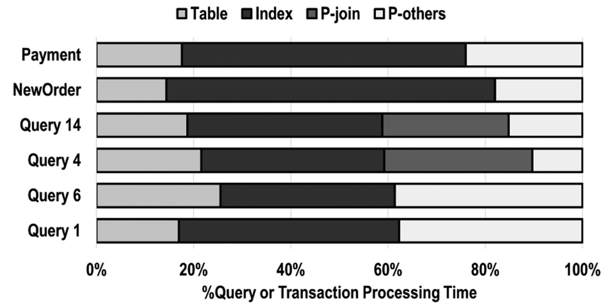


Fig. 4. Fractions of database table access time (Table), index access time (index), and host CPU processing time (P-join: Time spent by join processing, P-other: Time spent by other processing).

Fig. 3 demonstrates an example of a query that uses join processing. TPC-H Query 14 requests the join operation with a common column (*l_partkey = p_partkey*) from *lineitem* and *part* tables. This query also has the filtering conditions (❷) on *lineitem*. Typically, database query optimizers apply to filter first to reduce the size of the table (*lineitem* filtered by the conditions ❷) before initiating join. While filtering the *lineitem* table, the database system also generates a hash table containing *l_partkey* values. Each entry in the hash table contains pointers to all the records in the filtered *lineitem* table for a given unique *l_partkey* value. This hash table is repeatedly accessed during the join process to match the *p_partkey* of the *part* table. Depending on the number of the unique keys, the I/O system is repeatedly accessed to just get the index values for the *part* table. Thus, join requires significant I/O time to access the index structures [28], [31].

3.3 Performance Hurdles

Fig. 4 exhibits the breakdown of the query processing time with TPC-H benchmarks and the transaction processing time with TPC-C benchmarks (Payment and NewOrder, detailed information about the benchmarks in Section 5). We measured the breakdown with a system that equips the SSD platform. Also, we used a B+-tree index for the execution time breakdown analysis. Queries 1 and 6 employ the scan and filtering operation only. Other query processing includes join processing and filtering. We divide the total execution time per query or transaction into database table access time, index access time, join processing time (P-join), and execution time for other functions (P-others).

This execution time graph exhibits the index access occupies a significant fraction (47.5% on average) of the entire execution time for query and transaction processing. Especially the transactions of TPC-C benchmarks spend 63% of the entire execution time for index access on average. It is because the TPC-C benchmarks update the index structure requiring not only storing new index entries but also reordering the index structure. As shown in the analysis in this section, the large index structures can be a significant performance burden in DBMS even if the index structures are partially cached on the host memory.

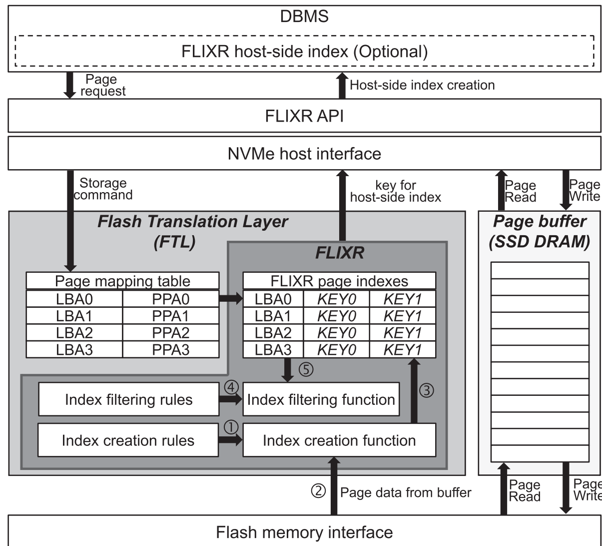


Fig. 5. Overview of FLIXR operation model.

4 Flash Indexer (FLIXR)

As investigated in the previous section, the large index structures can be a significant performance burden in DBMS even if the index structures are partially cached on the host memory. Furthermore, managing these index structures itself is an expensive task in terms of I/O and computation overheads. To reduce these costs, this paper presents FLIXR - an efficient data indexing mechanism in SSDs. A DBMS with a conventional indexing technique fetches the indices from memory or storage with conventional indexing techniques. Even if a DBMS with a conventional indexing scheme uses a hierarchical index structure, the host frequently fetches indices from storage if it uses huge database tables or the host memory size is not enough to keep all the indices. Such a behavior incurs a substantial amount of I/O operations. Unlike the conventional indexing techniques, FLIXR can manage all the indices inside SSDs and perform the in-SSD indexed scan or join processing. Such a scheme eliminates or minimizes page transfers for index access between the host and SSDs.

4.1 Overview of FLIXR Model

FLIXR's in-storage indexing mechanism exploits the native page translation structure in the FTL of modern SSDs. Traditionally, SSDs are block storage devices, and all data appears as pages. FLIXR partially exposes data semantics to the storage controller to automatically build per-(flash) page index information.

Fig. 5 describes the FLIXR operation model. FLIXR can create or update the page-level indices when page data is written to the flash memory. For the index creation process, FLIXR supports the index creation rules (①) that database administrators can specify. Then, FLIXR firmware executes the index creation function on the buffered page data (②) during the flash write process. The created index is stored in FLIXR's page-level index structures (③), which can be simply an enhanced FTL (as depicted in the figure) or a separate table structure indexed by LBA. In addition to the per-page indices, administrators can create host-side indices coupled with the per-page indices. We will explain the detail of each operation from Section 4.2.

Per-Page Index. FLIXR implements per-page index information associated with each LBA. FLIXR's index structure can be implemented as an extended metadata field in each FTL entry as depicted in Fig. 5 (or as a separate data structure pointed by an FTL entry). The basic purpose of this per-page index structure is to indicate whether a particular key is present on that page or not. When the LBA translation entry is searched, FLIXR can concurrently access the index. FLIXR supports an index creation and maintenance API, and an index usage API. While these are programmer visible APIs, one can piggyback them on top of existing NVMe commands using a few reserved bits without any protocol changes. If there is no host-side index associated with the per-page indices, FLIXR searches the indices corresponding to the pages that keep database tables sequentially. We will explain how FLIXR is associated with the host-side index later.

Index Types. FLIXR can provide various types of indices. In this paper, we demonstrate two types of indices that can cover various types of DBMS – a range-based type and a bitmap type. There is no limitation on the number of indices as long as the size of the index is configured at the start of the FLIXR setup appropriately.

The administrator may create a range-based index that indicates the *min* and the *max* values of a key located in a flash page. Such range-based indices are typically used for keys that can be easily ordered, such as integer values.

A bitmap type index is created for keys that may not be easily ordered, such as the unordered sets. For example, keys based on a state name or country name may use a bitmap index [32], [33]. Each bit in the bitmap corresponds to the presence/absence of one member of an unordered set. For example, a key corresponding to a list of states in the USA has a 50-bit bitmap, and each bit in the bitmap is set if a record with that particular state key is present on that page.

Exploiting Both Host-Side and in-SSD Indices. The indices associated with the FTL can significantly reduce the computation and I/O overheads. However, some database operations, such as lookup operations, can still benefit from the host-side indices. Utilizing only the in-SSD per-page indices involves a sequential search of a large number of the index entries. The proposed in-SSD indexing scheme is beneficial for the queries that demand intensive scan operations. On the other hand, its performance gain may not be significant for the operations that require a quick traversal of a small part of indices, such as point queries.

To address the above challenge, we propose an idea of co-locating the in-SSD indices and the host-side index structures (B+-tree in this paper). If a DBMS creates a B+-tree using all the per-page indices, the size of the tree structure becomes large unnecessarily. Then, the host system would suffer from substantial I/O and computation overheads as mentioned in Section 3. To resolve this issue, we implement a cooperative host/in-SSD indexing scheme that creates customized host-side indices using only a part of the per-page indices. We showcase an exemplar design of a host-side B+-tree structure whose leaf nodes contain *min* values and page numbers in SSD. Once an indexed search for a query reaches a leaf node, FLIXR traverses all the records on the corresponding page. The size of the host-side B+-tree

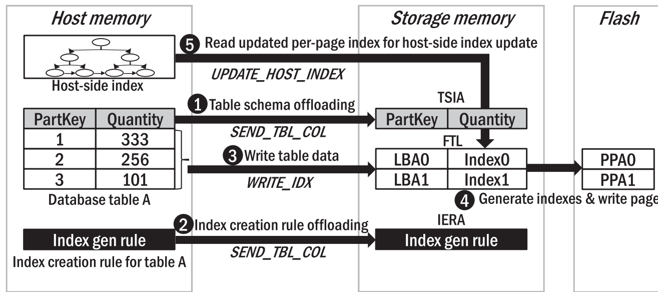


Fig. 6. Index creation and maintenance by FLIXR.

cooperating with the per-page indices becomes much smaller than the traditional host-side B+-tree that uses per-record or per-row information. Note that the performance overhead by a search for all the records on a page is insignificant. Eventually, the cooperative indexing scheme can achieve a significant performance improvement (more details in Section 5). Furthermore, programmers can implement any customized host-side index structures using the per-page indices without a heavy index creation overhead.

In the above example, the host first traverses the host-side index created with the per-page indices while processing a query. Whenever the host reaches a leaf node, it sends an SSD page request with the page number in the current node. Once the SSD receives the page access request, it compares only the *max* value in the FTL entry of the requested page as the DBMS already traversed the host-side index created by the *min* value. After the *max* value comparison, SSD decides whether to send the page to the host or not. Once the page is sent from SSD, the host can search for the rows that meet the query condition within that page. In this manner, FLIXR can boost a wide range of database operations.

If a DBMS updates a database table record, FLIXR first compares the *min* and *max* indices of the page that contains the entry and updates either the *min* value or the *max* value accordingly. If FLIXR does not create the host-side index, FLIXR updates only the in-SSD indices. On the other hand, if FLIXR uses both the host-side index and the in-SSD indices, the SSD firmware sends the logical page numbers whose indices (e.g., the *min* values) are updated to the DBMS. After then, the DBMS updates the B+-tree structure accordingly.

4.2 Index Creation

The first step in using FLIXR is to set up an index creation rule, which the database administrator generates. The rule is then communicated to the SSD controller hardware using a set of FLIXR APIs. Then, the SSD controller automatically generates per-page indices using the programmer-provided table schema information and the index creation rules whenever the host DBMS writes a page. This process flow is depicted in Fig. 6 which we will walk through. We first explain FLIXR's operations and APIs for index creation.

Offloading a Table Schema. Since the creation of the indices is now automatically done by FLIXR SSD controller, the SSD controller needs to know the database schema. As such, the administrator sends the structure of each database table, namely the number of columns, the data type, name, and size of each column, etc., to SSD. For a table, the FLIXR API creates a table id (*tblID*) and the table information along with the *tblID* is sent to the SSD using *SEND_TBL_COL* NVMe command (see ❶ in Fig. 6). Then, the FLIXR firmware on the SSD copies the database schema to a dedicated array structure, called *table structure information array (TSIA)*, and TSIA is indexed by the key *tblID*. The above process is repeated for each table in the database. Thus *TSIA* stores the structure of all tables in the database.

Due to the limited space in the SSD memory, FLIXR firmware may return a *success* or *failure* indication to the host administrator after processing the *SEND_TBL_COL* command. Note that a failure to register a table in FLIXR is a lost opportunity for performance since the database administrator may still create a traditional index structure for any table.

Per-Page Index Creation. FLIXR provides the index creation API, which passes the index generation function to the SSD controller. The index generation function could be simple, such as creating an index on *l_partkey* or could be a more complex function. To provide the most general indexing capability, the API allows the database administrator to pass a pointer to any custom index creation function. Then, the index creation API transfers the custom function code to the SSD using *SEND_IDX_ENC* (❷ in Fig. 6). The FLIXR firmware on SSD then stores the function pointer of the offloaded index creation function in the *index encoding rule array (IERA)*, which is indexed by the *tblID*. To support index creation and index updates concurrent with each page write, FLIXR provides *WRITE_IDX* that initiates the index creation alongside the usual data write command to the SSD (❸ in Fig. 6) and written to the flash memory (❹).

Host-Side Index Creation. Once an in-SSD per-page index is created, administrators can select whether to build the customized host-side index (explained in Section 4.1) or not. FLIXR provides the *UPDATE_HOST_INDEX* (❺ in Fig. 6), which is a simple interface between DBMS and FLIXR. A DBMS can call *UPDATE_HOST_INDEX* after a *WRITE_IDX* is called. This command uses the same page number in the latest *WRITE_IDX* command. Once *UPDATE_HOST_INDEX* is called, the SSD sends the corresponding *min* value to the host. With the metadata sent from SSD, a DBMS can create the customized host-side index by simply exploiting its index creation mechanism.

4.3 Index Maintenance

Once the *WRITE_IDX* command is received, the SSD firmware first buffers data in the SSD DRAM as it does by default. Then the firmware issues the page write command. Concurrently, the FLIXR firmware initiates an index generation process by executing the offloaded index creation function identified by the *tblID*. While scanning the table data in the buffered page, it parses the column entries in the table using the registered table structure identified by the *tblID*.

In the same manner as the per-page index creation, if a host-side index with per-page index is already created, the DBMS issues an *UPDATE_HOST_INDEX* command after the *WRITE_IDX* command. In this case, the *UPDATE_HOST_INDEX* gets the new index value. If the index value is newly updated, the DBMS removes the lastly accessed leaf node of the host-side index, creates a new node with the new index value, and updates the host-side index.

Given the scheme of the per-page indexing, there are two issues: 'how to keep the large-sized index structures' and

'how to provide enough availability while updating the index.' To address them, FLIXR includes an optional feature of separating the index from FTL. The SSD firmware keeps pointers of the separated index structures and reads them once it receives an index read command. The key benefit of the proposed idea is that if the administrators create an additional index (e.g., a secondary index), the proposed idea makes the SSD create it simply. Such flexibility of index maintenance significantly makes FLIXR available to a wide range of database systems. Second, if an index structure accounts for a large size, FLIXR can store the index pages in flash instead of managing entire index structures in the embedded DRAM. Then FLIXR can use a dedicated in-SSD DRAM space to cache indices. As a page is the basic fetch unit for a flash read, indices associated with thousands of pages can be embedded in a single physical page. With this index caching mechanism, we noticed that indices associated with *hot* pages are frequently cached in the embedded DRAM, thereby minimizing the need to access the flash page for index accesses.

4.4 Exploiting FLIXR Indices

FLIXR provides a set of APIs that can be used by the query optimizer (or directly by a programmer as we have done in this implementation) to define the query filtering rules that must be applied when reading a flash page. For instance, FLIXR can perform the *where* clause filtering directly at the page granularity without accessing separate index structures as is typically done in database systems.

Registering Index Comparison Rules. FLIXR provides an API to automatically exploit the per-page index to create a filter request. FLIXR provides a new NVMe command, *SEND_IDX_ICR*, which transfers a data filtering or index comparison rule to the SSD. This command uses the *tblID* and *a index comparison rule number (icrID)* as parameters. Note that multiple queries can simultaneously access a single database table, thus each table may have multiple filtering rules. As such, FLIXR allocates multiple index comparison rules identified by the *icrID* and *(tblID)*. Typically, the index comparison rules are relatively simple operations, such as *less than*, *greater than*, and *bitmap match*.

Read Page With Filtering. Once the index comparison rules are registered in the SSD, the FLIXR database system issues *READ_IDX* command, which performs index comparison operations and page-level data access filtering. This NVMe command uses the *tblID* and *icrID* as parameters (plus the normal NVMe read parameters). Note that these two new parameters are used as the identifiers of the registered index comparison rules.

If a DBMS uses only per-page indices, each *READ_IDX* command calls the index comparison function to check whether the target page keys meet the registered filtering condition or not. The FLIXR firmware can access the per-page indices without additional index structure scanning since the indices are associated with the LBAs in the page mapping table. The FLIXR firmware simply performs the lightweight comparison operations for the target page. If the target page contains the items that meet the filtering condition, FLIXR allows the normal flash read process and returns the fetched page to the host system. Otherwise, the FLIXR firmware does not issue a page read request to the flash array to prevent the SSD from fetching the unnecessary page data.

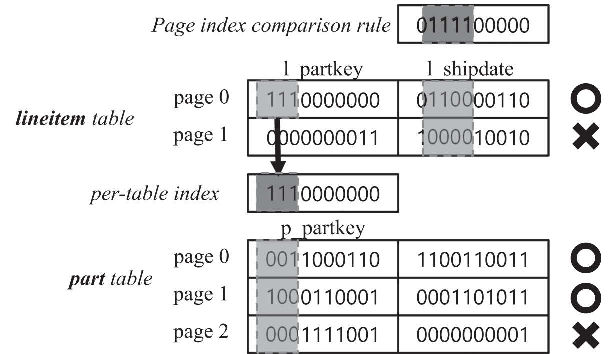


Fig. 7. An example of per-table indexing (In this example, FLIXR performs TPC-H Query 14 (See Fig. 3).)

If a DBMS uses both host-side and per-page indices, it first traverses the host-side index. Once it reaches a leaf node, the DBMS issues *READ_IDX* command with the SSD page number in the current node. After then, the SSD processes the indexing and page fetching in the same manner as the in-SSD per-page index-only case.

In summary, FLIXR supports the index comparison rules (④), which specify the filtering rules based on the created FLIXR's page-level indices (see Fig. 5). When the host database systems access the database table from the SSD, FLIXR can apply the light-weight index comparison operations (⑤) by comparing the per-page index values with the previously registered index comparison rules. If the page data does not include the items that meet the filtering conditions, FLIXR cancels the flash memory read to eliminate the unnecessary page accesses.

4.5 Support for Join Processing

Join is a key database operation to identify common records across multiple tables. For the join processing, the key values of one table are compared against the key values in another table. In the example TPC-H Query 14 (see Fig. 3), two database tables (*lineitem* and *part*) have a common key (*partkey*) in both tables. This query requests the common *partkey* records that are present in both tables and performs certain computations on these common records.

Fig. 7 shows an example of how FLIXR applies a novel join processing. In this example, FLIXR performs TPC-H Query 14, described earlier, with a join on the *lineitem* and *part* tables. We assume that the DBMS uses per-page indices only. Also, we assume that the bitmap-based per-page indices are created for the *partkey* in each table, and for *l_shipdate* for the *lineitem* table. The query first performs the filtering operation using *l_shipdate* on the *lineitem* table. The filtering condition selects only a subset of pages from the *lineitem* table that matches the *l_shipdate* filter; in the example, page 0 is fetched, but page 1 is filtered out. For every page selected by the filter (page 0 in our example), FLIXR concurrently scans that page to find all the unique partkey values present on that page. It then creates a per-table bitmap index, as shown in the figure. This bitmap index is a single global index that shows what are all the partkey values that are present in all the fetched pages. In this example, the Fig. shows that the first three partkeys values are present (as

indicated by three bits of 1 in the per-table index). This per-table index is then used to scan the *part* table and identify any page that has at least one partkey that matches the partkey values in the per-table index. In this example, page 0 has at least one matching partkey (the third bit of the *p_partkey* bitmap is 1), page 1 has at least one matching partkey (the first bit of the *p_partkey* is 1). But page 2 has partkey values that match none of the partkeys present in the lineitem table. Hence, page 2 will not be fetched in any join processing.

As described above, there are two major steps in join processing. The first step is to filter one of the tables and create the per-table index. The second step uses this newly created intermediate per-table index to find common keys in the second table. This multi-step join process can be initiated using the following three APIs. Note that these three APIs may be embedded in a single join API called by the query optimizer. For clarity, we explain all three embedded APIs separately.

Per-Table Index Creation. Like the per-page index creation process, the database administrator can offload per-table index creation functions to the SSD using the new NVMe command, called *SEND_TIDX_ICR*. The offloaded index creation function is registered in the *table index rule array (TIRA)* indexed by *tblID* and per-table index creation id (*tiaID*).

Step1 of Join. When the query optimizer calls join it generates *READ_IDX_JOIN1* and *READ_IDX_JOIN2* calls. The *READ_IDX_JOIN1* starts the per-table index creation operation. This operation applies the filtering condition on a table, and for every fetched page, it automatically scans for the unique join keys that are present on the page and create a per-table index based on the registered per-index creation function. Note that this per-table index could be reused in future joins on the same table with similar filtering conditions. Hence, FLIXR saves the intermediate per-table index in storage and may then avoid re-creating this index in future join calls on the table.

Step2 of Join. In the second step, *READ_IDX_JOIN2* API call is made. This API takes as parameters the two table ids (tblId1 and tblId2) that are used in the join operation, and the id of the intermediate table index created, and the common key value that is used for the join process. While the SSD reads the second table used in the join process, FLIXR firmware uses the per-table index to filter any page that has no matching keys using the *READ_IDX_JOIN2* call.

4.6 Design Considerations

Consistency. In current SSDs, FTL entries are only cached in DRAM, but FTL itself is stored in flash. FLIXR index is also stored in flash even if part of the index data is cached in DRAM. Hence, there is no index data loss in case of power failure. The only issue to consider is if an index is currently being updated within the DRAM FTL, but it has not been copied into the flash. To create a consistent view of the index, the FTL index must be atomically updated. For this purpose, when a page write request is received, which is the only time an index may be updated, first, we clear a single redo-log bit in that FTL entry to zero before starting the index creation process. Then, the index creation process starts. Once that process is completed, the redo-log bit is set one, indicating the completion of index creation. On a reboot from a power failure, any FTL entry whose redo-log bit is still set to 0 is considered to have an invalid index, and a new index creation process is initiated just on that page. We rely on SSD's default crash consistency to allow any FTL entry with the built-in index to be preserved across power failures.

Security. FLIXR does not introduce new security vulnerabilities since accesses to the flash pages first go through the database administration security protocol, and then each I/O access is further validated by OS file system checks before the access reaches the FTL. FLIXR thus piggybacks on existing security checks to determine access rights. Since SSD is a shared I/O device, it may be possible to construct side-channel attacks by accessing FTL to extract information such as whether a particular page has a pre-built index, and with the additional effort, it may be possible to narrow the set of key values stored, which is outside the scope of this paper.

Exploiting FLIXR for Various use Cases. FLIXR can be adapted to various database organization scenarios with simple extensions to the current implementation. For example, suppose the database tables are stored with a sorted column layout. In that case, FLIXR can exploit this knowledge by essentially stopping its index scan search if the search hits a page whose index exceeds the search criteria. This functionality can be implemented by enabling the FLIXR's API to communicate this unique data organization knowledge to the FLIXR firmware.

While the current description of FLIXR uses row-storage databases, FLIXR can be extended to improve the efficiency of columnar storage. For instance, the FTL structure can be extended to have pointers to all the pages that hold the remaining column data associated with the current page. That way, single-column page access will also be able to identify other columns associated with the same rows. FLIXR can work even if a row spans across two pages. With the SEND_TBL_COL command, FLIXR can support various row layouts, including multi-page-sized rows. If a single row exceeds an SSD page, an extension of the FLIXR firmware is required. For example, a single bit would be added to each FTL entry. Each bit indicates that the index is valid for that page, and for every other page, FLIXR can set the bit to true.

Exploiting Internal Parallelisms. FLIXR can simply exploit the benefit of various types of parallelism inside SSDs. Prior studies revealed that it is critical to exploit channel-level, package-level, die-level, and plane-level parallelisms to improve the I/O performance of flash memory [4], [34]. Several works focused on such issues and proposed the various types of the flash I/O request scheduling schemes that make SSDs benefit from such internal parallelisms [35], [36], [37]. As FLIXR's overhead is negligible and agnostic to I/O scheduling, it is not harmful to achieve high degrees of internal parallelism of SSDs. For example, FLIXR can be simply combined with the proposed I/O scheduling schemes.

4.7 Applying and Using FLIXR

FLIXR requires a few modifications to DBMS: (1) The query optimizers in a DBMS application should take advantage of the available FLIXR indices to extract maximum benefits. The query optimizers send the necessary metadata to the NVMe host driver to create WRITE_IDX, READ_IDX, READ_IDX_JOIN1, and READ_IDX_JOIN1 commands listed in Table 1.

TABLE 1
New NVMe Commands for FLIXR

Command	Type	Description	Parameter
Sending indexing rules			
SEND_TBL_COL	Adm	Send table column item structure	tblID
SEND_IDX_ENC	Adm	Send per-page index generation rule	tblID
SEND_IDX_ICR	Adm	Send per-page index comparison rule	tblID, icrID
SEND_TIDX_ENC	Adm	Send per-table index generation rule (for join processing)	tblID, tiaD
In-SSD indexing & filtering with I/O			
WRITE_IDX	I/O	Write pages with index generation	tblID
READ_IDX	I/O	Read pages with index comparison	tblID, icrID
READ_IDX_JOIN1	I/O	Read pages with index comparison and generate per-table indices	tblID, icrID, tiaID
READ_IDX_JOIN2	I/O	Read pages with index comparison using per-table indices	tblID1, tblID2, icrID, tiaID
Host-side index creation			
UPDATE_HOST_INDEX	I/O	Read updated index of currently written page	tblID, icrID

(2) The DBMS applications (or at least the DBMS runtime) must use FLIXR API whenever the DBMS administrator creates the indices. Similarly, a database table creation operation should also pass the schema information to FLIXR using SEND_TBL_COL, SEND_IDX_ENC, SEND_IDX_ICR, and SEND_TIDX_ENC command in Table 1. (3) The NVMe host driver and SSD firmware modification is needed for newly added commands and metadata generated by the DBMS applications. Note that we do not change the NVMe protocol itself since all the APIs are simply enhanced versions of existing NVMe commands which have several bytes of reserved space for transferring all the API parameters. We implemented FLIXR functionality with approximately two person-years of effort, which we believe is relatively inexpensive.

Once the FLIXR API and firmware are applied to a DBMS and SSDs, administrators can easily use FLIXR. In any basic configuration (e.g., *conf* file in mysql) in any database, if administrators add an attribute (*index:*) and a line (*mode: with an argument*) accordingly, they can select indexing mode. If the argument is set to *flixr_ssd_only*, the database creates only the in-SSD per-page index. If the argument is set to *flixr_ssd_host*, the FLIXR API automatically creates both the per-page index and the tiny B+-tree that we explained in Section 4.2. Then the API makes the database access the indices of the requested pages accordingly. If the argument is not set, only the traditional host-side index of the corresponding DBMS is created.

4.8 Cost Overhead

FLIXR creates the indices per page and associates these per-page indices to each logical block. Thus FLIXR requires additional storage space for the per-page indices. With the FLIXR API, the administrators can set the index of any size. In the case of the database tables that we used in this paper, the size of the index structures ranges from 8 to 24 bytes per page. FLIXR's per-page index structure can include multiple per-page indices as long as they fit in the pre-assigned size. Hence more indices can be packed per-page if more space is assigned to FLIXR indices. The storage overhead of FLIXR's per-page indices is approximately 0.05–0.15% of the entire storage space with the default configuration.

Moreover, the size of the FLIXR index structure is only up to 10% compared to the traditional index structure (B+-tree nodes). The traditional index structures take up space, and more importantly, they take several layers of indirection before reaching the target data. By creating indices on a per-page basis and storing them within an FTL entry, we dramatically reduce the cost of accessing the target data. The small-sized per-page index enables the SSD controllers with low computation power to traverse it quickly, thus reducing the query processing time.

FLIXR can also create per-table indices associated with database tables and index creation rules for join processing. While various sizes of the per-table index are available, we used the 8-byte per-table index. The size of the per-table index space allocated in SSD DRAM is negligible since most databases have a limited number of tables.

Although FLIXR's index structure size is very small compared to the large SSD storage space, the index structures may sacrifice some DRAM space that can be utilized for caching the page mapping table. However, as we show in our results, the positive effect of in-SSD indexed table access outweighs the DRAM use. Furthermore, DRAM capacity has been growing steadily in SSDs, and we believe FLIXR storage is a clever usage of DRAM rather than simply caching more page mapping information or flash pages.

We can also minimize the DRAM space use with demand-based fetching. Demand-based mechanisms utilize the small DRAM space as a cache for the *recently-used* mapping entries, and the *cold* mapping information is stored in the large space of the over-provisioned flash memory [38]. The performance degradation is negligible compared to the pure in-DRAM page-level mapping [39]. Along with the feature, as mentioned in Section 4.3, it is possible for FLIXR's index to be decoupled from the FTL entries, and FLIXR can use a dedicated in-SSD DRAM space to cache a subset of indices. As a result, our FLIXR implementation does not sacrifice the pre-assigned page buffer space in the SSD development platform.

5 Evaluation

We implemented FLIXR on the Cosmos+ OpenSSD board – the open-source SSD research and development platform [40]. The OpenSSD platform equips a Xilinx Zynq-7000 programmable SoC with a dual-core ARM Cortex-A9 application processor. Thus users can program the hardware logic

TABLE 2
Evaluation Platform Configuration

OpenSSD platform	
Processor	Dual-core ARM Cortex-A9 @ 1 GHz
FPGA	Xilinx Zynq-7000 (350K logic cells)
DRAM	On-board 1 GB DDR3-1066
NAND flash	8 channels, 8 ways/channel, 2 TB MLC
Flash page size	16 KB
Interconnection	AXI-lite (command) and AXI (data) bus
Protocol	NVM express 1.1
Host interface	PCIe Gen.2 8× (Max. 4GB/s)
Page cache replacement	Least Recently Used (LRU)
Host system	
CPU	Quad-core Intel i7-4790 @ 4 GHz
DRAM	16 GB DDR3-1600
OS & file system	Linux kernel 3.19.0, direct I/O

as well as the SSD controller firmware [41]. The controller firmware consists of the NVMe command decoder, page buffer management, and flash translation layer functions running on the embedded processor. OpenSSD supports up to 1.38 GB/s sequential read data bandwidth. Although this performance metric is lower than the commercial high-end NVMe SSDs, the read bandwidth supported by the OpenSSD is similar to the mid-range NVMe SSDs and much higher than SATA SSDs [42]. Hence, the OpenSSD evaluation platform reflects real commodity SSD systems. In addition, it is also possible to emulate various performance ranges by adjusting the configurations of the SSD controller. OpenSSD assigns 16 MB DRAM space as page buffers, and this space is relatively small compared to the total SSD DRAM size. FLIXR's index structures do not sacrifice this page buffer space. Thus FLIXR does not change the performance of normal page transfers. The host system equips an Intel i7 CPU and 16 GB DRAM. The host system runs Linux operating system, including NVMe driver [6]. Note that we modified the NVMe driver by adding the new commands for FLIXR operation as shown in Table 1. Table 2 lists the detailed configuration of the SSD platform and the host system.

For evaluation, we use microbenchmarks and real database workloads. We created four microbenchmarks that performs basic database operations: *Scan with filtering (Scan)*, *join processing (Join)*, *Insert*, and *Delete*. The *Scan* kernel simply filters rows of a table with a query condition. The *Join* kernel performs a simple inner join with two tables. The *Delete* kernel creates random queries with random conditions to delete the rows that meet the conditions. The *Insert* creates random queries to add randomly generated rows in a single database table. We used *lineitem* (for all the microbenchmarks) and *part* (for *Join*) tables that are in TPC-H benchmarks.

For real database workloads, we use online transaction processing (OLTP) and online analytical processing (OLAP) workloads. In order to study FLIXR's index maintenance performance, we tested the *NewOrder* and *Payment* transactions in the TPC-C benchmark [43], an OLTP benchmark. The selected transactions include frequent database writes and updates. Thus, efficient index updates are necessary for these workloads. We configured the database tables with the setting of *warehouse* as 100. For OLAP workloads that are data-intensive but not update-intensive, we selected the TPC-H benchmark suite [30]. We used the database tables with a scale factor of 10. We study four queries (queries 1, 4, 6, and 14), including key operations in the TPC-H benchmark. Queries 1 and 6 perform the aggregation function with the filtering operations. Queries 4 and 14 include the join processing that includes the inter-table relation conditions. Due to the OpenSSD software stack limitations, we emulated our database setup to match the TPC-C and TPC-H schema implemented in MySQL [44], [45] in terms of database management functions, such as logging, table management.

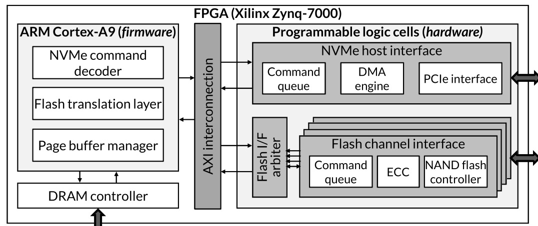
We first implement the no-indexing that does not access database tables with any indices. With this implementation, once a query or transaction arrives, the host system sequentially searches all the database records. To compare the performance of FLIXR with the conventional host-oriented index structures, we employed a B+-Tree index structure. The host processor performs all index-related operations for conventional host-side indexing, such as creation, updates, and traversing of indices and index accesses inside SSD for the large-sized database tables. The generated index structures are stored in the SSD, and some of the layers in the tree are cached on host DRAM on demand. We call this mechanism as *Host-Side Index* in the evaluation section. We also implemented in-SSD B+-tree indexing to compare its performance to FLIXR. The in-SSD B+-tree indexing creates the same index structure as the host-side index. However, unlike the host-side index, the SSD controllers traverse the indices once a request arrives at DBMS. While traversing the B+-tree structure, this scheme automatically accesses flash pages where the indices are stored. Also, the SSD controller keeps hot B+-tree nodes into a predetermined space inside the DRAM buffer in the SSD. We implemented such a scheme by modifying the SSD firmware.

We implemented FLIXR with two types of indexing mechanisms – in-SSD per-page indexing only (called *FLIXR-S*), and the combination of a (tiny) host-side B+-tree and in-SSD per-page indexing schemes (called *FLIXR-HS*), which was described in Section 4.1. In the case of the *FLIXR-HS*, all nodes of the tiny B+-tree are kept in the host DRAM as the tree size is small enough, as explained in Section 4.8.

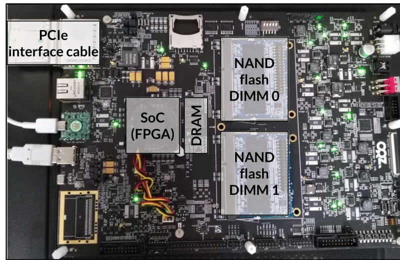
6 Experimental Results

6.1 Basic Operation Performance

We measured *Queries per Minute* as the performance metric and normalized the experimental results to the performance of the host-side index. Fig. 9 shows the performance of the microbenchmarks on five different system configurations: a database system without indexing, host-side indexing, in-SSD B+-tree indexing, FLIXR-S and FLIXR-HS. In *Scan*, *Join*, and *Delete*, FLIXR-HS achieves the best performance among all the architectures. FLIXR-HS exhibits 75% (Scan), 40% (Join), and 30% (Delete) better throughput than the host-side index, respectively. Note that the host-side index shows up to 3x better throughput than the no-indexing scheme in these microbenchmarks. Both FLIXR-HS and the host-side index could achieve the performance improvement by exploiting B



(a) SSD controller architecture on FPGA



(b) SSD development board

Fig. 8. Evaluation platform.

+-tree traversing to access the rows in the database table. In addition, FLIXR-HS could reduce the overhead in the host-side index and efficiently access the per-page indices in SSD, thus achieving significant performance improvement over the host-side index only.

FLIXR-S could mitigate a heavy overhead for fetching indices from SSD to the host DRAM even with iterating all the per-page indices. However, it should iterate all the FTL table entries to search for the rows that satisfy query conditions. As a result, in *Scan* and *Delete*, FLIXR-S shows 8% lower performance than the host-side index as the overhead of the iteration is slightly heavier than the B+-tree traverse overhead.

In the case of *Insert*, a database without indexing shows the best performance as it has no overhead of index update. However, we don't consider this configuration as a critical one because the database without indexing is not a realistic case. We observe that using only in-SSD indices (FLIXR-S) achieves the best performance among the architectures containing indexing schemes. FLIXR-S and FLIXR-HS achieve 58% and 39% better performance than the host-side index, respectively. FLIXR-HS requires additional host DRAM accesses and computations to update its own B+-tree structure, so it achieves less speedup than FLIXR-S. The host-side index incurs heavier I/O and computation overheads than FLIXR, thus exhibiting a worse speedup than FLIXR.

In the case of the in-SSD B+-tree indexing, we observe that it significantly increases the SSD controller's computation overheads. *Delete* and *Insert* requires more computations than *Scan* and *Join* due to the updates of the B+-tree indices while the computation power of the SSD controller is worse than the host CPUs. As a result, the in-SSD B+-tree indexing suffers from a slowdown compared to the host-side index and FLIXR. Unlike *Delete* and *Insert*, the in-SSD B+-tree indexing achieves similar performance to the FLIXR-S in*Scan* and *Join* as those operations do not require heavy computations for the B+-tree updates. Although the performance of B +-tree traversal with the SSD controller is worse than the host CPUs, the in-SSD B+-tree indexing mitigates the I/O overheads so that it could achieve similar performance to the FLIXR-S.

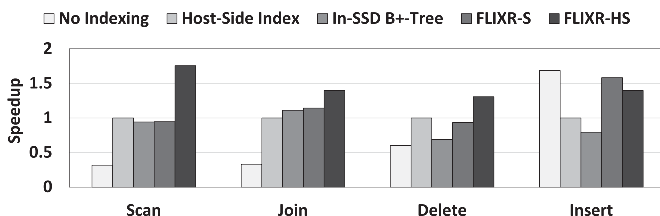


Fig. 9. Basic operation performance (Queries per Minute (QpM), normalized to the host-side index).

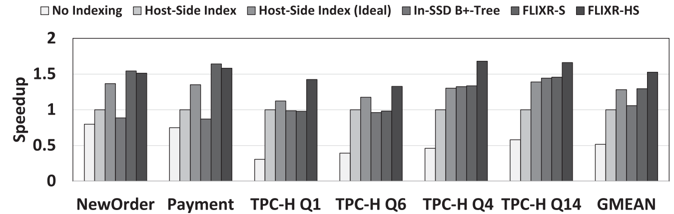


Fig. 10. Database performance (tpmC in NewOrder and Payment. QpM in TPC-H queries. All results are normalized to the host-side index.)

As shown in the results in this section, FLIXR-S and FLIXR-HS show different performance results. The database administrators can select either of the solutions depending on the target database characteristics.

6.2 Database Workload Performance

Fig. 10 compares the performance of TPC-C and TPC-H benchmarks. We measured *Transactions per Minute (tpmC)* with TPC-C benchmarks and *Queries per Minute (QpM)* with TPC-H benchmarks. TPC-C benchmarks have frequent page updates. On the other hand, TPC-H does not have data updates, and hence index maintenance and updates are not as frequent.

In addition to the four configurations that we mentioned in the previous section, we also studied the performance of an ideal host-side index configuration. Since some real workloads tend to show skewed access patterns, some of the B+-tree nodes may be frequently re-accessed in the host memory. The entire B+-tree is loaded into host memory in the ideal configuration, and no B+-tree nodes are replaced. In essence, we assume an infinite-sized host-side cache for caching B+-tree nodes.

Overall, FLIXR-S and FLIXR-HS achieve 29.6% and 52.6% performance improvement over the host-side index, respectively. In TPC-C benchmarks, FLIXR-S and FLIXR-HS outperform the host-side indexing by 59% and 55%, respectively. As we showed in Section 6.1, FLIXR-S exhibits the best performance in the insert operations, thus achieving significant speedup over the host-side index. FLIXR-HS also reduces computation overhead for B+-tree updates compared to the host-side index as the B+-tree in FLIXR has fewer leaf nodes than the traditional B+-tree that has per-row leaf nodes.

Unlike the simple scan in Section 6.1, in TPC-H queries 1 and 6, which are the cases of point queries, FLIXR-S suffers from the performance degradation as shown in Fig. 10. On the other hand, the experimental results show that FLIXR-HS resolves such a problem and the overhead for traversing all the records on a page is not critical to overall performance. In TPC-H queries 4 and 14, which require *join processing*, both

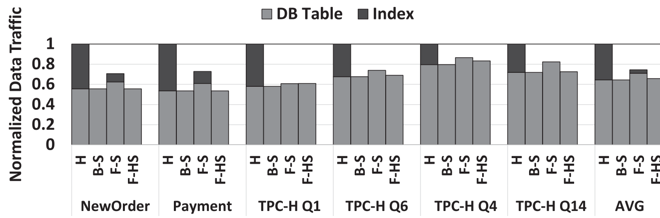
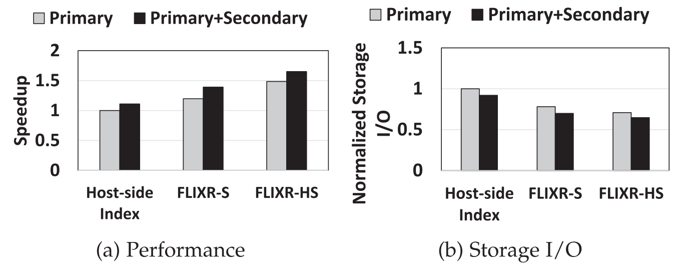


Fig. 11. Data traffic from storage (H: Host-side index, B-S: In-SSD B +-tree, F-S: FLIXR-S, F-HS: FLIXR-HS).



(a) Performance (b) Storage I/O

Fig. 12. Performance and data traffic with multiple keys (Results with TPC-H Q6 and Q14, all results are normalized to the host-side index using primary key only.)

FLIXR-HS and FLIXR-S benefit from the per-table join index. FLIXR supports bitmap indexing for join processing. Hence, they improve the query processing performance more effectively than the host-side index. Among them, FLIXR-HS could outperform FLIXR-S as it filters a large portion of pages in the first table faster than FLIXR-S. FLIXR-HS and FLIXR-S improve the performance queries 4 and 14 by 67% and 39% over the host-side index.

As shown in Fig. 10, the ideal host-side index shows 28.1% better performance than the conventional host-side index scheme due to the reduced index access time. However, the computations of the page addresses and memory references for access validation and OS checks still become the performance bottleneck. FLIXR significantly reduces such overhead, thus achieving the best performance. In the case of the in-SSD B+-tree indexing, it suffers from a slowdown in the TPC-C benchmarks. It is because the benchmarks frequently perform *Delete* and *Insert* in indices as well as the database tables. As mentioned in the previous section, due to the limited computation power of the SSD controller, the overheads caused by the indices incur a slowdown with the in-SSD B+-tree indexing. In the cases of TPC-H benchmarks, the in-SSD B+-tree indexing shows similar performance to FLIXR-S. On average, the in-SSD B+-tree indexing shows a 5.8% speedup over the host-side index, which is 46.6% worse than FLIXR-HS.

6.3 Storage I/O

To study FLIXR's bandwidth reduction impact, we measure the amount of page data fetched from the SSD to host as shown in Fig. 11. The fetched data size is normalized to the host-side index. We split the total data transfer into two components for host-side index bars. In the case of the host-side index, 36% of total storage traffic is caused by index accesses. When looking only at the data pages, FLIXR-HS reduces the traffic to the host by 31% compared to the host-side index, and only 3% of total traffic was for the tiny B +-tree update interfaced with the in-SSD per-page indices.

Both FLIXR-S and FLIXR-HS are coarse-grained index mechanisms, and hence they may occasionally send a data page to the host even when a particular key value is not present on the page. Such a situation occurs when the key is within the key ranges stored in the index for that page, but that particular key may not be present on that page. Because of this coarse grain index, the database table accesses with FLIXR are slightly higher than a precise indexing scheme used with the host-side index. However, the host-side index incurs additional data transfer between storage and host-side memory to access the index structures, even when the top few levels of the index trees are cached on the host. The in-SSD B+-tree indexing exhibits the same I/O overhead as the overhead caused by the database table access with the host-side index as it does not send any pages for index access to the host. However, the SSD controller suffers from a heavy computation overhead, as mentioned in Section 6.1. Thus, the in-SSD B +-tree indexing spends a long time to transfer the same amount of SSD pages as the host-side index.

Managing indices of large databases tends to create vast index access overheads, particularly when database entries are updated [2]. With FLIXR, the index update time, in fact, grows sub-linearly as the index building time can be masked within the page write access latency. As such, FLIXR outperforms the host-side index even with a much larger database.

6.4 Effect of Secondary Key Indexing

Among the evaluated database workloads, TPC-H queries 6 and 14 include the filtering conditions for multiple keys. or instance, query 6 reads three columns from *lineitem* table as mentioned in Section 3.1. We can use the items in the *l_shipdate* column, which has the widest value range, as the primary key. The items in other two columns (*l_discount* and *l_quantity*) can be used for creating the secondary key indices. For those queries, we apply the index filtering using both the primary and the secondary keys for FLIXR and the host-side indexing. Note that FLIXR can generate indices for multiple keys if the created indices are packed within the reserved per-page index fields.

In this section, we evaluate the performance impact of the multi-key indexing supported by FLIXR. We measure the performance and storage I/O traffic changes when the indices are built for both the primary and the secondary keys. The conventional database systems access indices created for these columns if the corresponding indices are pre-made.

Fig. 12 shows the experimental results. The performance improvement with FLIXR-HS using the secondary key is 65% compared to the host-side indexing mechanism. FLIXR-S also outperforms the host-side indexing mechanism by 39%. FLIXR-HS with multiple keys improves performance by 17% compared to when only the primary key is used for indexing.

Data traffic from storage is obviously reduced when the multi-key indices are applied (as shown in the right-hand Fig. 12). For the host-side indexing, the storage I/O traffic for database table accesses is decreased by 10.3% when compared to just a primary key only system, however, the entire traffic (both data and index traffic) is reduced by only 8%. It is because with the conventional host-side indexing the host database system now needs to access more index structures – the primary and second indices. On the other hand, FLIXR's in-storage indexing mechanism allows that more efficient page-filtering using multi-key indices.

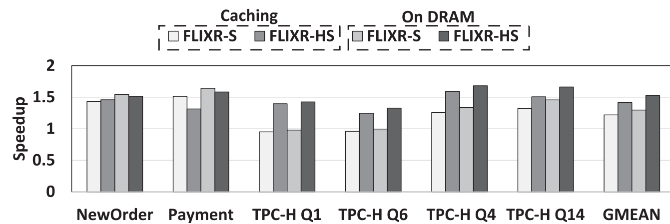


Fig. 13. Performance of FLIXR with a caching mechanism.

6.5 Separating Indices From FTL

As mentioned in Section 4.3, FLIXR provides the functionality of separating the index structure from FTL and caching it into DRAM to provide more flexibility and mitigate the performance penalty when accessing large-sized and in-flash index structures. In this experiment, we study the performance of FLIXR index separation and caching. We configured FLIXR to store the index pages in flash and then uses only 16MB SSD DRAM to cache indices and measure their performance.

Fig. 13 shows the performance of FLIXR with separate index data combined with caching. The *On DRAM* configuration, the rightmost two bars in each group, shows the performance when all the per-page indices are managed in the embedded DRAM along with the FTL table. This configuration will guarantee the fastest index mapping from LBAs. The *caching* configuration results, the leftmost two bars in each group, show the performance with a very limited cache space used for index storage. Our evaluation shows that the performance of FLIXR-S and FLIXR-HS with caching achieve 94.1% and 93.6% of the *On Dram* performance. This evaluation reveals that FLIXR can minimize the use of the embedded DRAM with the caching mechanism while achieving performance benefits similar to the FTL-embedded index.

7 Conclusion

Large-scale data analytics require extensive indexing of data to reduce unnecessary data movement between the storage and host. In this paper, we represent an efficient in-SSD indexing mechanism – FLIXR, which exploits SSD's unique FTL page-mapping architecture to organize page-level indices. FLIXR can execute user-defined index generation functions whenever page data is written or updated in flash memory. The page-level indices from FLIXR are embedded in FTL, which are utilized for performing filtering or join processing using the SSD controller. FLIXR's page-level indexing mechanism enables efficient data filtration even with the wimpy embedded processor on the storage platform. Our evaluation with the Open-source SSD development platform reveals that the overall query processing performance is improved by 52.6% compared to the conventional host-side indexing mechanism.

Acknowledgments

The views, opinions, and/or findings expressed are those of the author(s) and should not be interpreted as representing the official views or policies of the Department of Defense, the U.S. Government, or the Korea Government. Gunjae Koo and Yunho Oh have contributed equally to this work as the first authors.

References

[1] O. Kocberber, B. Grot, J. Picorel, B. Falsafi, K. Lim, and P. Ranganathan, "Meet the walkers: Accelerating index traversals for in-memory databases," in *Proc. 46th Annu. IEEE/ACM Int. Symp. Microarchit.*, 2013, pp. 468–479

[2] J. Yu and M. Sarwat, "Two Birds, One stone: A fast, yet lightweight, indexing scheme for modern database systems," *Proc. Very Large Data Bases Endowment*, vol. 10, no. 4, pp. 385–396, Nov. 2016.

[3] D.-S. Hwang, W.-H. Kang, G. Oh, and S.-W. Lee, "Flash-aware index scan in postgreSQL," in *Proc. 31st IEEE Int. Conf. Data Eng. Workshops*, 2015, pp. 161–166.

[4] H. Roh, S. Park, S. Kim, M. Shin, and S.-W. Lee, "B+-tree index optimization by exploiting internal parallelism of flash-based solid state drives," *Proc. Very Large Data Bases Endowment*, vol. 5, no. 4, pp. 286–297, Dec. 2011. [Online]. Available: http://dx.doi.org/10.14778/2095686.2095688

[5] G. Lee, S. Shin, W. Song, T. J. Ham, J. W. Lee, and J. Jeong, "Asynchronous I/O stack: A low-latency kernel I/O stack for ultra-low latency SSDs," in *Proc. USENIX Conf. USENIX Annu. Tech. Conf.*, 2019, pp. 603–616.

[6] NVM Express, "NVM Express Revision 1.1," 2014. [Online]. Available: http://www.nvmexpress.org/wp-content/uploads/NVM-Express-1_1.pdf

[7] C.-Y. Liu, Y. Lee, M. Jung, M. T. Kandemir, and W. Choi, "Prolonging 3D NAND SSD lifetime via read latency relaxation," *Proc. 26th ACM Int. Conf. Architectural Support Program. Lang. Oper. Syst.*, 2021, pp. 730–742.

[8] G. Koo *et al.*, "Summarizer: Trading communication with computing near storage," in *Proc. 50th Annu. IEEE/ACM Int. Symp. Microarchit.*, 2019, pp. 219–231.

[9] D. Tiwari *et al.*, "Active flash: Towards energy-efficient, in-situ data analytics on extreme-scale machines," in *Proc. 11th USENIX Conf. File Storage Technol.*, 2013, pp. 119–132.

[10] S.-W. Lee, D.-J. Park, T.-S. Chung, D.-H. Lee, S. Park, and H.-J. Song, "A log buffer-based flash translation layer using fully-associative sector translation," *ACM Trans. Embedded Comput. Syst.*, vol. 6, no. 3, pp. 18–es, Jul. 2007.

[11] P. Raju, R. Kadekodi, V. Chidambaram, and I. Abraham, "PebblesDB: Building key-value stores using fragmented log-structured merge trees," in *Proc. 26th Symp. Oper. Syst. Princ.*, 2017, pp. 497–514.

[12] N. Dayan and S. Idreos, "Dostoevsky: Better space-time trade-offs for LSM-tree based key-value stores via adaptive removal of superfluous merging," in *Proc. Int. Conf. Manage. Data*, 2018, pp. 505–520.

[13] D. Kim, C. Park, S.-W. Lee, and B. Nam, "BoLT: Barrier-optimized lsm-tree," in *Proc. 21st Int. Middleware Conf.*, 2020, pp. 119–133.

[14] M. Sahli, E. Mansour, and P. Kalnis, "StarDB: A large-scale DBMS for strings," *Proc. Very Large Data Bases Endow.*, vol. 8, no. 12, pp. 1844–1847, Aug. 2015.

[15] A. Acharya, M. Uysal, and J. Saltz, "Active disks: Programming model, algorithms and evaluation," in *Proc. 8th Int. Conf. Architectural Support Prog. Lang. Operating Syst.*, 1998, pp. 81–91.

[16] E. Riedel, G. A. Gibson, and C. Faloutsos, "Active storage for large-scale data mining and multimedia," in *Proc. 24rd Int. Conf. Very Large Data Bases*, 1998, pp. 62–73.

[17] J. Wang, D. Park, Y.-S. Kee, Y. Papakonstantinou, and S. Swanson, "SSD in-storage computing for list intersection," in *Proc. 12th Int. Workshop Data Manage. New Hardware*, 2016, pp. 1–7.

[18] J. Do, Y.-S. Kee, J. M. Patel, C. Park, K. Park, and D. J. DeWitt, "Query processing on smart SSDs: Opportunities and challenges," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2013, pp. 1221–1230.

[19] S. Seshadri *et al.*, "Willow: A user-programmable SSD," in *Proc. 11th USENIX Conf. Operating Syst. Des. Implementation*, 2014, pp. 67–80.

[20] R. Balasubramonian *et al.*, "Near-data processing: Insights from a MICRO-46 workshop," *IEEE Micro*, vol. 34, no. 4, pp. 36–42, Jul. 2014.

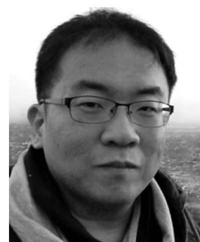
[21] S.-W. Jun *et al.*, "BlueDBM: An appliance for big data analytics," in *Proc. 42nd Annu. Int. Symp. Comput. Archit.*, 2015, pp. 1–13.

[22] S. Kim, H. Oh, C. Park, S. Cho, S.-W. Lee, and B. Moon, "In-storage processing of database scans and joins," *Informat. Sci.*, vol. 327, pp. 183–200, 2016. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0020025515006003

[23] B. Gu *et al.*, "Biscuit: A framework for near-data processing of big data workloads," in *Proc. 43rd Int. Symp. Comput. Archit.*, 2016, pp. 153–165.

[24] I. Jo *et al.*, "YourSQL: A high-performance database system leveraging in-storage computing," *Proc. Very Large Data Bases Endow.*, vol. 9, no. 12, pp. 924–935, Aug. 2016.

[25] E.-M. Lee, S.-W. Lee, and S. Park, "Optimizing index scans on flash memory SSDs," *SIGMOD Rec.*, vol. 40, no. 4, pp. 5–10, Jan. 2012, doi: 10.1145/2094114.2094116.
[26] S. T. On, H. Hu, Y. Li, and J. Xu, "Lazy-update B+-tree for flash devices," in *Proc. 10th Int. Conf. Mobile Data Manage., Syst. Serv. Middleware*, 2009, pp. 323–328.
[27] K. K. Matam, G. Koo, H. Zha, H.-W. Tseng, and M. Annavaram, "GraphSSD: Graph semantics aware SSD," in *Proc. Proc. 46th Int. Symp. Comput. Archit.*, 2019, pp. 116–128.
[28] R. Ramakrishnan and J. Gehrke, *Database Management Systems*, 3rd ed. New York, NY, USA: McGraw-Hill, 2003.
[29] Y. Chen, Y. Lu, K. Fang, Q. Wang, and J. Shu, "Utree: A persistent B+-tree with low tail latency," *Proc. Very Large Data Bases Endow.*, vol. 13, no. 12, p. 2634–2648, Jul. 2020.
[30] TPC, "TPC benchmark H (TPC-H)." [Online]. Available: http://www.tpc.org/tpch/
[31] P. Mishra and M. H. Eich, "Join processing in relational databases," *ACM Comput. Surv.*, vol. 24, no. 1, pp. 63–113, Mar. 1992.
[32] T. Johnson, "Performance measurements of compressed bitmap indices," in *Proc. 25th Int. Conf. Very Large Data Bases*, 1999, pp. 278–289.
[33] K. Wu, E. J. Otoo, and A. Shoshani, "A performance comparison of bitmap indexes," in *Proc. 10th Int. Conf. Informat. Knowl. Manage.*, 2001, pp. 559–561
[34] F. Chen, B. Hou, and R. Lee, "Internal parallelism of flash memory-based solid-state drives," *ACM Trans. Storage*, vol. 12, no. 3, May 2016, Art. no. 13, doi: 10.1145/2818376.
[35] M. Jung and M. T. Kandemir, "Sprinkler: Maximizing resource utilization in many-chip solid state disks," in *Proc. IEEE 20th Int. Symp. High Perform. Comput. Archit.*, 2014, pp. 524–535.
[36] B. Mao and S. Wu, "Exploiting request characteristics and internal parallelism to improve SSD performance," in *Proc. 33rd IEEE Int. Conf. Comput. Des.*, 2015, pp. 447–450.
[37] H. Wang, P. Huang, S. He, K. Zhou, C. Li, and X. He, "A novel I/O scheduler for SSD with improved performance and lifetime," in *Proc. IEEE 29th Symp. Mass Storage Syst. Technol.*, 2013, pp. 1–5.
[38] Y. Zhou, F. Wu, P. Huang, X. He, C. Xie, and J. Zhou, "An efficient page-level FTL to optimize," in *Proc. 10th Eur. Conf. Comput. Syst.*, 2015, pp. 1–16.
[39] D. Ma, J. Feng, and G. Li, "LazyFTL: A page-level flash translation layer optimized for NAND flash memory," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2011, pp. 1–12.
[40] J. Kwak, S. Lee, K. Park, J. Jeong, and Y. H. Song, "Cosmos+ OpenSSD: Rapid prototype for flash storage systems," *ACM Trans. Storage*, vol. 16, no. 3, Jul. 2020, Art. no. 15, doi: 10.1145/3385073.
[41] Xilinx, "Zynq-7000 all programmable SoC data sheet," 2018. [Online]. Available: https://www.xilinx.com/support/documentation/data_sheets/ds190-Zynq-7000-Overview.pdf
[42] FastestSSD, "SSD Ranking: The fastest solid state drives," 2018. [Online]. Available: http://www.fastestssd.com/featured/ssd-rankings-the-fastest-solid-state-drives
[43] TPC, "TPC benchmark C (TPC-C) standard specification revision 5.11." [Online]. Available: http://www.tpc.org/tpc_documents_current_versions/pdf/tpc-c_v5.11.0.pdf
[44] Oracle. MySQL. [Online]. Available: https://www.mysql.com/
[45] Vadim Tkachenko. tpcc-mysql benchmark tool: Less random with multi-schema support, 2016. [Online]. Available: https://www.percona.com/blog/2016/08/09/tpcc-mysql-benchmark-tool-less-random-with-multi-schema-support/



Gunjae Koo (Member, IEEE) received the BS and MS degrees in electrical and computer engineering from Seoul National University, in 2001 and 2003, respectively, and the PhD degree in electrical engineering from the University of Southern California, in 2018. He is currently an assistant professor with the Department of Computer Science and Engineering, Korea University. His research interests include computer system architecture and spans parallel processor architecture, storage and memory systems, accelerators, and secure processor architecture. Prior to joining Korea University, he was an assistant professor with Hongik University. His industry experience includes the position of a senior research engineer with LG Electronics and also a research intern with Intel.

Yunho Oh (Member, IEEE) received the BS, MS, and PhD degrees from the School of Electrical and Electronic Engineering, Yonsei University, Seoul, Korea, in 2009, 2011, and 2018, respectively. He had worked as a software engineer with Mobile Communications Business, Samsung Electronics. Currently, he is working as an assistant professor with the Department of Electrical and Computer Engineering, SungKyunKwan University (SKKU). Prior to joining SKKU, he worked as a postdoctoral researcher with Parallel Systems Architecture Lab (PARSA), EPFL, Switzerland. His research interests include hardware and software architectures for energy-efficient datacenters, processor architectures (CPUs, GPUs, and neural network accelerators), in-storage processing, memory system, and high-performance computing.

Won Woo Ro (Senior Member, IEEE) received the BS degree in electrical engineering from Yonsei University, Seoul, Korea, in 1996, and the MS and PhD degrees in electrical engineering from the University of Southern California, in 1999 and 2004, respectively. He worked as a research scientist with the Electrical Engineering and Computer Science Department, University of California, Irvine. He currently works as a professor with the School of Electrical and Electronic Engineering, Yonsei University. Prior to joining Yonsei University, he worked as an assistant professor with the Department of Electrical and Computer Engineering, California State University, Northridge. His industry experience includes a college internship with Apple Computer, Inc. and a contract software engineer with ARM, Inc. His current research interests include high-performance microprocessor design, GPU microarchitectures, neural network accelerators, and memory hierarchy design.

Hung-Wei Tseng (Member, IEEE) received a PhD in computer science and engineering from the University of California, San Diego. He is currently an assistant professor with University of California, Riverside. He is especially interested in topics related to heterogeneous computer architecture and near-data processing, including building intelligent storage devices and efficient use of hardware accelerators. Prior to joining the University of California, Riverside, he was an assistant professor with the Department of Computer Science and the Department of Electrical and Computer Engineering, NC State University.

Murali Annavaram (Fellow, IEEE) received the PhD degree in computer engineering from the University of Michigan, Ann Arbor, in 2001. He is currently a professor with the Ming-Hsieh Department of Electrical and Computer Engineering and with the Department of Computer Science (joint appointment), the University of Southern California. He is the founding director with the REAL@-USC-Meta center that is focused on research and education in AI and learning. His research group tackles a wide range of computer system design challenges, relating to energy efficiency, security and privacy. Prior to his appointment with USC he worked with Intel Microprocessor Research Labs from 2001 to 2007. In 2007 he was a visiting researcher with the Nokia Research Center working on mobile phone-based wireless traffic sensing using virtual trip lines, which later become Nokia Traffic Works. In 2020 he was a visiting faculty scientist with Facebook, where he designed the checkpoint systems for distributed training. He is a senior member of the ACM.

▷ **For more information on this or any other computing topic, please visit our Digital Library at** www.computer.org/csdl.