LUNAR: A Native Table Engine for Embedded Devices

Xiaopeng Fan
East China Normal University
Shanghai, China
xpfan@stu.ecnu.edu.cn

Song Yan
East China Normal University
Shanghai, China
syan@stu.ecnu.edu.cn

Yuchen Huang
East China Normal University
Shanghai, China
ychuang@stu.ecnu.edu.cn

Chuliang Weng
East China Normal University
Shanghai, China
clweng@dase.ecnu.edu.cn

Abstract

Embedded systems have evolved tremendously in recent years. We perform a study on SQLite and find that the multiple layers of abstraction drastically reduce bandwidth utilization. To minimize the bandwidth loss in the I/O path, we propose Lunar, a novel native table storage engine. Lunar performs a cross-layer design across the database and file system to avoid the pitfalls of multi-layer abstraction while providing SQL-compatible APIs. It employs a type-aware storage layout that considers the access patterns of different data types. Then, Lunar designs a variable-size allocator to reduce fragmentation and optimize RAM and I/O bandwidth usage. Further, considering the limited resources on embedded devices, Lunar employs a modular architecture that enables selecting modules on demand. It also offers optional consistency modes to make a trade-off between resource consumption and consistency. Experiments show that Lunar achieves higher bandwidth utilization, outperforming state-of-the-art approaches while consuming fewer resources.

CCS Concepts: • **Computer systems organization → Embedded software**; • **Information systems → DBMS engine architectures**.

Keywords: embedded devices, storage management, modularity, consistency

ACM Reference Format:
Xiaopeng Fan, Song Yan, Yuchen Huang, and Chuliang Weng. 2023. LUNAR: A Native Table Engine for Embedded Devices. In *Proceedings of the 24th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES '23), June 18, 2023, Orlando, FL, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3589610.3596276

LCTES '23, June 18, 2023, Orlando, FL, USA

ACM ISBN 979-8-4007-0174-0/23/06…$15.00
https://doi.org/10.1145/3589610.3596276

1 Introduction

I/O performance has been recognized as one of the key factors affecting the system performance of embedded devices [19, 21, 25]. Traditional embedded devices' I/O system consists of a set of host software and storage devices. Recently, research efforts have been made toward various levels of the I/O stack to improve the efficiency of flash-based storage [15, 19, 21, 24, 29, 38, 41].

Embedded applications use databases, such as SQLite [44], to manage structured data. SQLite is ubiquitous, present in billions of devices, including smartphones, computers, televisions, and automobiles [13]. We start this paper with a study on SQLite bandwidth under three popular file systems (EXT4 [34], FATFS [7], and F2FS [27]). Figure 1 shows the bandwidth obtained by the block layer and the SQLite layer during the insertion operation. Surprisingly, we find that SQLite suffers a significant bandwidth loss compared to the block layer, with losses of 83.8%, 83.5%, and 81.7% under EXT4, FATFS, and F2FS, respectively.

Drilling down, we collect the bandwidth of the file system layer on the I/O path. As shown in Figure 3, we observe that compared with the block layer, the file system layer loses 10.9-48.6% of bandwidth, and the database layer further loses 61.7-78.8% of bandwidth relative to the file system layer. Indeed, a file is an abstraction of a collection of blocks, while the table is an abstraction on top of the file. Each layer of abstraction requires the maintenance of extra metadata, resulting in exacerbated I/O amplification that not only results in longer latency but also reduces the flash lifetime. The more abstractions present, the greater the loss of bandwidth.

To gain a clear understanding of the interaction between the database and file systems, we employed a finer-grained approach to track the specific I/O generated at the block layer during SQLite insert operations. As illustrated in Figure 4, we observe that a single SQLite insert operation results in 9 block I/Os to the storage device. The fundamental cause of this inefficiency is that the two abstractions of tables and

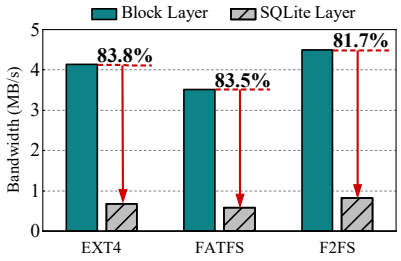

Figure 1. Bandwidth Tracing on the I/O Path. We use Mobibench [16] for insertion operations and collect the bandwidth of the SQLite layer and block layer, representing the bandwidth received from SQLite and the bandwidth issued to the block layer, respectively.

files interact in an uncoordinated manner, resulting in excessive I/O activity. This issue is known as *journaling of journal* (JOJ) [17, 25, 40]. While some studies provide solutions such as utilizing `fdatasync()` [17] or fine-grained logging [37], they mostly focus on a single layer of the I/O stack (e.g., database layer or file system layer). Despite these efforts, the integration of the database and file systems is suboptimal [17], leaving substantial room for improvement.

The results of our study illustrate how multiple layers of abstraction drastically reduce the database's bandwidth utilization. From the perspective of cross-layer (i.e., database layer and file system layer) optimization, there is one question: *Can we remove the file abstraction and directly construct the table on the block device?*

This approach is reminiscent of record-oriented file systems prevalent in the mainframe era [31]. Unlike mainframe, embedded devices face severe resource constraints. Additionally, we provide table-level storage services, which results in significant differences in the design of data structures and storage layouts, as compared to file systems. Constructing table structures directly on embedded storage devices presents a novel solution to address the issue of low bandwidth utilization, but it also poses some unique challenges.

First, there are typically three types of data in a database: table metadata, table data, and logs, each with distinct access characteristics. For instance, metadata has a smaller access granularity, and its access is mostly random. So, how to design a storage layout that considers access characteristics is crucial to meet the storage needs of different data types.

Second, existing allocators often use fixed-size blocks as allocation units, with common examples including the 4 KB block size for EXT4 and F2FS, and the 16 KB block size for FATFS on a 32 GB storage device. Recent studies show that SQLite file fragmentation is severe and can seriously affect performance [14, 16, 18, 20, 21]. The storage allocation contributes to fragmentation [14]. In addition, an entire block must be loaded to RAM even if only a single tuple is accessed, which leads to both I/O bandwidth and RAM wastage. Therefore, it is essential to develop an efficient allocator that can

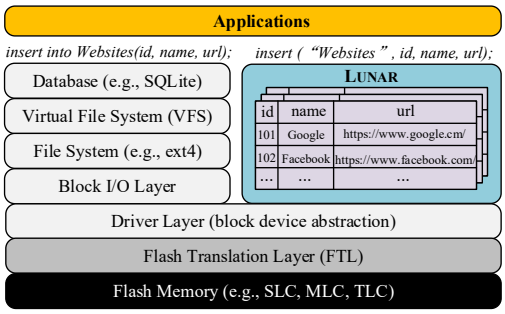

Figure 2. Traditional and LUNAR's I/O System.

minimize fragmentation and reduce the wastage of RAM and I/O bandwidth due to large fixed blocks.

Given that resources in embedded devices are severely constrained, another challenge is to have a modular architecture. The modularity allows the system to be tiny and low-power, which is critical for embedded devices. Besides, the modular design makes the system simple to upgrade existing modules and add new modules. Some studies have considered modularity, such as the embedded operating system's modular components [26] and modular network architecture for embedded devices [12]. However, existing storage systems for embedded devices, including file systems [7, 27, 34] and databases [39, 44], tightly integrate functional modules to provide services. As far as we know, there are currently no modular storage systems for embedded devices. Finally, not all embedded applications require the same consistency guarantees [22]. For resource-constrained embedded devices, storage systems need to make a trade-off between resource consumption and consistency.

In this paper, we propose LUNAR, a native table storage engine for embedded devices that abstracts the data service of the database's table. Figure 2 shows the traditional and LUNAR's I/O system. The key is that it removes the file abstraction and performs cross-layer design across the database and file system. To the best of our knowledge, it is the first native table storage engine designed for embedded devices. The contributions are summarized as follows:

- We perform a study on SQLite and find that the multiple layers of abstraction drastically reduce bandwidth utilization. To minimize the bandwidth loss in the I/O path, we propose LUNAR, a native table storage engine, which removes the file abstraction, integrates the file system and database layer, and provides SQL-compatible APIs.
- With file abstractions removed, LUNAR can manage storage space more effectively, including the design of storage layout and allocator. LUNAR employs a type-aware storage layout that considers the access patterns of different data types to improve performance. Then, LUNAR uses a variable-size block allocator designed to minimize fragmentation and reduce the wastage of RAM and I/O bandwidth.
- Resources on embedded devices are limited. LUNAR employs a modular architecture to support selecting modules

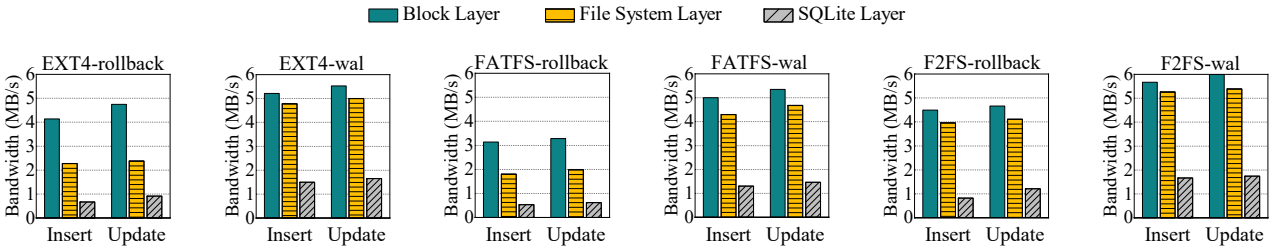

Figure 3. Bandwidth Tracing on the I/O Path.

on demand. Meanwhile, modularity makes it easy to upgrade existing modules and integrate new modules. In addition, LUNAR offers four optional consistency modes to make a trade-off between resource consumption and consistency.

• Experiments show that LUNAR achieves bandwidth utilization of up to 80.6%. LUNAR outperforms competitive alternatives with a throughput increase of 1.3-4.1× and low latency ranging from 27.9-57.2%. Compared to SQLite, LUNAR reduces ROM and RAM size by 81.1% and 93.9%, respectively.

2 Background and Motivation

2.1 Embedded Device I/O System Overview

The left-hand side of Figure 2 shows the embedded device I/O system. SQLite is the most extensively used database system, with potentially over one trillion active SQLite currently in use [13, 44]. It is estimated to be among the most widely deployed software libraries of any kind [13]. SQLite stores database files in the local file system.

The Virtual File System (VFS) enables applications to access different file systems in a unified way. File systems abstract storage devices into files. Numerous file systems have been developed for embedded systems [19, 27, 34].

File access requests are translated to block I/O at the block layer, and the driver layer provides an abstraction for block devices. The Flash Translation Layer (FTL) emulates block-level storage on top of flash memory, which has become the primary storage solution for embedded devices [19].

2.2 Understanding the Performance

We perform an extensive performance study to understand how databases utilize the underlying storage. We use Mobibench [16] with SQLite as its test database to perform synchronous 4 KB insert operations. SQLite supports two journal modes: rollback mode and write-ahead log (wal) mode [13]. We employ the default rollback mode [44]. SQLite runs on three popular file systems (EXT4 [34], FATFS [7] and F2FS [27]). We execute 100K SQL requests and collect the SQLite-layer bandwidth and block-layer bandwidth, which means the bandwidth received from SQLite and the bandwidth issued to the block layer, respectively. §4.1 further describes other details of the testbed.

Figure 1 shows that SQLite suffers from significant bandwidth loss compared to the block layer for all three file systems. Specifically, the losses are 83.8%, 83.5%, and 81.7% under the EXT4, FATFS, and F2FS, respectively.

To further determine the performance penalty on the I/O stack, we use the `strace` to collect the bandwidth of the file system under the Mobibench workload. Moreover, we add experiments on SQLite's wal mode and update operations.

Figure 3 presents the results. While performing insert operations with SQLite's rollback mode on EXT4, the file system layer loses 44.9% of the bandwidth compared to the block layer, and the database layer loses 83.8%. Indeed, a file is an abstraction of a collection of blocks, and a table is an abstraction on top of a file. Each layer of abstraction requires extra metadata maintenance, resulting in exacerbated I/O amplification. In addition, the uncoordinated interaction between the database and file systems due to multiple layers of abstraction results in much I/O activity (§2.3). In simpler terms, adding an extra layer of abstraction to the I/O system will incur a reduction of available bandwidth.

For SQLite's wal mode on EXT4, I/O operations are typically sequential, making it a good fit for flash storage. The file system layer experiences lower bandwidth losses compared to rollback mode. However, the database layer still incurs a bandwidth loss of 70.1-71.2% compared to the block layer.

FATFS and EXT4 show the same trend. For insertion operations in rollback mode, F2FS has an improvement of 1.73× and 2.20× over EXT4 and FATFS, respectively. Moreover, whether in rollback or wal mode, the bandwidth loss of the F2FS layer is only 11.8% and 8.3%, respectively. This is because F2FS benefits flash storage by converting random updates into sequential ones. Nevertheless, multiple layers of abstraction still cause a bandwidth loss of 70.5%-81.7% for the SQLite layer compared to the block layer.

The results from this experiment suggest that we need to be completely overhauled and vertically integrated so as to properly incorporate their respective characteristics.

2.3 Interaction Between the Database and File Systems

To better understand the uncoordinated interaction between SQLite and EXT4, we use `blktrace` [4] and `MOST` [16] to

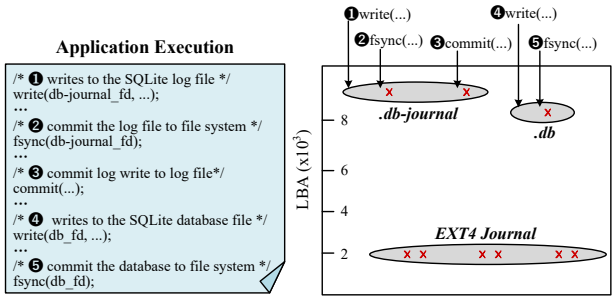

Figure 4. Block I/O Accesses in the Insert Operation.

perform an in-depth analysis of the block-level I/O activity caused by SQLite and EXT4. Specifically, we insert one record (4 KB) into the SQLite. EXT4 is mounted with metadata only journaling (`ordered` mode) [34]. SQLite employs the default rollback-recovery logging (`TRUNCATE` mode) that stores rollback information in `.db-journal` files [13].

The left of Figure 4 illustrates a simplified segment of the call trace during an insert, with five primary phases involved. The right of Figure 4 shows the I/O generated by each phase at the SQLite layer and EXT4 layer. It illustrates the logical block address (LBA) write accesses over the operation. Specifically, the accesses are clustered in three regions: the SQLite journal (i.e., `.db-journal`), the SQLite table (i.e., `.db`), and the EXT4 journal, respectively. For the `.db-journal` region, the first I/O is for creating and updating a `.db-journal` file, and the second I/O is a commit log write to this journal file to ensure atomicity. The `.db` region represents the insertion into the table. For each I/O in the above two regions, SQLite calls `fsync()` to make the results persistent. Each `fsync()` follows two writes to EXT4 `Journal`, including a writing journal descriptor and metadata.

An insert operation triggers 9 writes to the device. The inefficient interaction between the database and file systems causes I/O amplification which not only degrades performance but also reduces the lifetime of flash-based storage.

3 Design of Lunar

3.1 Overview

Lunar is a native table storage engine for embedded devices. The architecture is shown in Figure 5, and Lunar has the following design features.

• **Lightweight** (§3.2). Lunar performs cross-layer design by integrating the database and file system layers. It removes file abstraction and directly stores tables on the device, while providing SQL-compatible APIs such as `create table`, `insert`, `select`, and `update`. Lunar greatly improves bandwidth utilization and provides a shorter I/O path.

• **Storage Efficient** (§3.2). Lunar employs a novel type-aware storage layout that considers the access patterns of different data types to optimize performance. Moreover, it uses a variable-size block allocator to reduce fragmentation and minimize RAM and I/O bandwidth wastage.

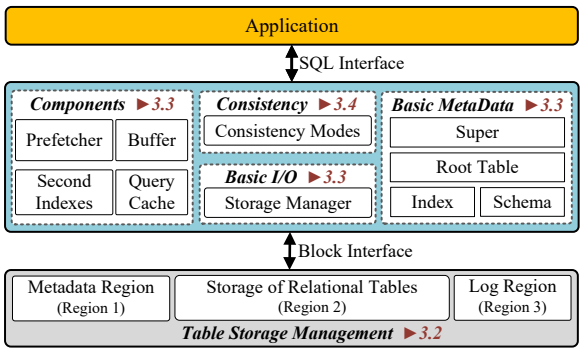

Figure 5. The Architecture of Lunar.

• **Modular** (§3.3). The modular architecture of Lunar allows for building a table storage engine by selecting specific modules to adapt to resource-constrained embedded devices. The modularity also makes it more convenient to upgrade existing modules and integrate new modules.

• **Flexible** (§3.4). Lunar provides four flexible consistency modes to meet the consistency requirements of various embedded applications. It makes a trade-off between resource consumption and consistency.

3.2 Table Storage Management

Our study in §2 shows that traditional file system-based databases have low bandwidth utilization due to multiple layers of abstraction. To address this issue, Lunar removes the file abstraction and stores tables directly on the device. While this approach does introduce some maintenance complexity, we believe it is worth the cost, as it significantly improves bandwidth utilization. Further, Lunar gains direct control over the storage management, allowing the design of an efficient data layout and allocator that corresponds to database access patterns.

3.2.1 Type-Aware Storage Layout. Figure 6 shows the high-level layout of Lunar data structures within the embedded storage device. While Lunar may share some similarities with file systems, it is crucial to recognize the fundamental distinction that it removes file abstraction, and instead, stores tables directly on the device. Lunar divides the storage space into five zones. The SUPER stores global system information, such as the total space size, number of tables, beginning address, and size of each zone; the ROOT_ZONE includes the root table and its metadata; the META_ZONE serves as a metadata collection for common tables; the DATA_ZONE stores the data of common tables, and the LOG is used to guarantee data durability and transactional writes. A detailed explanation of the ROOT_ZONE, META_ZONE, and DATA_ZONE is provided in §3.2.3.

As shown in Figure 7, the five zones are divided into three regions according to access characteristics. Region 1 comprises a set of data blocks with the smallest storage unit, e.g., 512 bytes, and three zones (SUPER, ROOT_ZONE, and

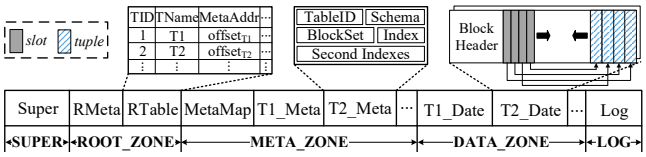

Figure 6. ZONEs Data Structure.

META_ZONE) belong to this region. These three zones store metadata information that is typically small in size, and is often accessed randomly, causing significant fragmentation. Lunar tries to control the fragmentation by assigning dedicated locations for metadata structures. The metadata structures are updated in place within these locations. Besides, the DATA_ZONE belongs to region 2. Unlike traditional allocators with fixed-size blocks, region 2 uses a variable-size block allocator. In §3.2.2, we describe how the allocator is designed. Region 3 consists of large blocks, e.g., 64 KB, that store the LOG. Given that logging involves large and sequential I/O operations, it is advantageous to have a separate large and contiguous region to accelerate the logging process and avoid interference with foreground tasks.

Different data types exhibit distinct access characteristics, including access granularity and sequential or random access. We categorize them into different regions according to data types. This not only improves processing efficiency within each individual region but also reduces interference between regions caused by differences in access characteristics, thus enhancing overall system performance.

3.2.2 Variable-Size Block Allocator. Existing allocators commonly utilize fixed-size blocks, such as 4 KB or 16 KB, as their allocation unit [7, 27, 34]. While these allocators are straightforward to implement, they present several issues.

Recent studies indicate that many of the fragmented files are SQLite files (with extensions `.db`, `.db-journal`, and `.db-wal`) [14, 16, 18, 20]. Accessing fragmented data results in a random I/O pattern, reducing the I/O locality and amplifying the overhead of managing the mapping cache in devices [18, 20]. Due to the multi-layer abstraction, databases cannot directly manage storage, and therefore, typically rely on the file system for persistence. The process of storage allocation is a major contributor to fragmentation [14, 14, 17, 19].

Slab allocators [5, 33], which are specifically designed for RAM management, are known for their effectiveness in preventing memory fragmentation caused by varying size allocations [6]. By drawing inspiration from the slab allocator, it is possible to implement storage management that minimizes fragmentation. However, flash-based storage devices are naturally different from RAM in that RAM is byte-addressable and does not provide persistence. These differences introduce significant challenges in terms of I/O granularity and persistence requirements.

I/O granularity and allocator design. RAM-based slab allocators can employ small allocation units (e.g., a few bytes

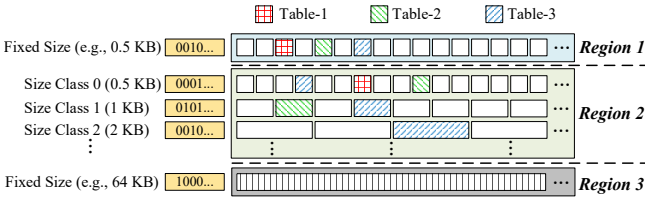

Figure 7. Variable-size Block Allocator. The figure shows the distribution of three tables across different regions.

or tens of bytes), owing to its support for byte-addressability. However, the usage of small allocation units causes significant I/O amplification for block devices. To minimize the additional I/O, allocation unit sizes in Lunar are aligned with 512-byte sectors, which represent the smallest atomic operation unit provided by the device. In addition, Lunar eliminates file abstractions, which enables the achievement of fine-grained storage management. Data transfer between the storage device and RAM occurs at the sector granularity, which reduces the wastage of RAM and I/O bandwidth.

As shown in Figure 7, the blocks in region 2 are conceptually organized in *size classes*. Each class contains two types of blocks: bitmap blocks for managing allocation status, and data blocks for storing the data. The bitmap blocks for all classes have a fixed size of 512 bytes, i.e., atomic operation size. Then, each class includes data blocks with a specified size. Size class 0 contains the smallest blocks, with a block size of 512 bytes. Subsequent size classes contain blocks of exponentially growing size, i.e. blocks in size class i+1 are twice as large as those in size class i. The growth factor is two. Note that the minimum and maximum size classes, as well as the growth factor, can be configured based on the workload, and it is necessary to ensure that all blocks in each class are sector-aligned. In particular, setting the growth factor to 1 results in a traditional fixed-size allocator.

Allocation requests can be categorized into two types. First, for allocation requests that are at most the maximum block size of a class (e.g., 64 KB), the best-fit algorithm is utilized to locate a fitting block. Second, for allocation requests that exceed 64 KB, Lunar divides the request into blocks of 64 KB and one additional block of smaller size. All blocks are allocated using the first method.

Persistence Requirements. Unlike slab allocators based on volatile RAM, allocators designed for storage devices have a persistent requirement to ensure consistency. The allocator's consistency refers to the consistency between the allocation status blocks, i.e., the bitmap blocks, and the data blocks. There are two types of inconsistency that may occur. First, if a block is marked as allocated on the bitmap but its data is not actually persisted, it results in storage space loss. Second, if the data has been persisted but the bitmap has not, it may cause data being overwritten. This section discusses the allocator's consistency, while the consistency between table metadata and table data is described in §3.4.

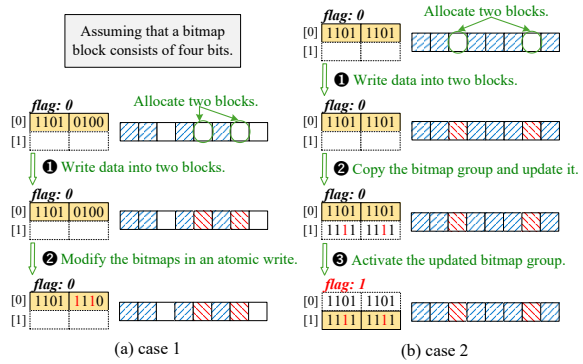

Figure 8. Allocator Persistence Requirements.

During the allocation process, there are three possible situations in which inconsistencies may arise.

Case 1: Figure 8(a) shows the allocation of one or more data blocks within a class, with the corresponding bitmaps of these data blocks belonging to a bitmap block. We first write the data to the data blocks, followed by updating the corresponding bitmap block in place, which leverages the atomicity of one sector provided by the devices. Atomic updates require only one I/O operation, making it the solution with the lowest consistency overhead.

Case 2: Figure 8(b) depicts the process of allocating one or more data blocks within a class, which necessitates updating bitmaps spanning multiple bitmap blocks. We use copy-on-write (COW) to ensure the consistency of multiple bitmap block updates. To achieve this, we reserve space for COW in the bitmap blocks by dividing them into two groups. Each class has a flag that indicates the currently active bitmap group. Specifically, we first write the data to the data block, then copy the current bitmap group, update the copy, and finally activate the updated group with the flag. Due to the small amount of bitmap block updates, using COW can avoid the overhead of journal recycling and release.

Case 3: For allocation requests that span across multiple classes, each class employs the aforementioned strategies to ensure consistency. Note that inconsistency may arise in case of a system crash during allocation between classes. Hence, we adopt a transaction mechanism to ensure consistency for allocation requests spanning multiple classes.

Limitation. Slab-like allocators suffer from an issue of storage space wastage. For example, an allocation request of 55 KB will occupy a 64 KB block. Space utilization and fragmentation are two competing factors, and we believe that with the continuous growth of storage device capacity, sacrificing some storage to reduce fragmentation, improve performance, and prolong device lifespan is worthwhile. Additionally, many existing technologies have proposed solutions to improve space utilization [8, 14, 30], but this study does not focus on this issue.

3.2.3 ZONEs Data Structure.

As shown in Figure 6, LUNAR divides the storage space into five zones.

Table 1. Implemented Modules.

Module		ROM (KB)	RAM (KB)
Basic	RAM Management	12.358	0.114
	Device Management	20.150	0.203
System Optimization	Buffer	1.732	0.023
	Prefetcher	2.834	0.025
Table Optimization	Query Cache	4.539	0.056
	Secondary Index(B+ Tree)	28.054	0.024

ROOT_ZONE. The ROOT_ZONE includes the root table and its metadata. The root table is a specialized table that stores critical information about common tables, such as their names and metadata locations. Functioning much like a directory tree in a file system, the root table contains structured data that includes comprehensive table-level information. In contrast to a directory tree, the root table is optimized for fast retrieval, utilizing a flat structure that simplifies management, much like a common table.

DATA_ZONE. As previously stated, region 2 is exclusively used for DATE_ZONE, which stores table data. The variable-size allocator assigns blocks to each table. Each block organizes tuples according to a slot-based layout. The block header contains essential information, such as the number of used slots, and the pointer to the next table block. The slot array maps each slot to its corresponding tuple position. This slot-based approach provides two benefits: first, the slot ordering ensures that the tuples are sorted in the correct order, and second, relocating tuples within the block does not require any modification in the index since the slot offset is used to retrieve the tuple.

META_ZONE. META_ZONE comprises two parts: an allocator and an array of table metadata. *MetaMap* manages the allocation status of metadata entries. Metadata entries are aligned on a 512 bytes boundary, which represents the atomic allocation unit. Specifically, each metadata entry has five properties: `BlockSet`, which stores the index information of all blocks owned by the table; `Schema`, the user-defined table schema; and `Index`, the primary index. The type of primary index is optional, including hash-based (default) and tree-based. The layout of index entries is deterministic, represented by a quad of `<Key, ClassID, BlockID, Offset>`. The `ClassID` identifies the class within the variable-size allocator, the `BlockID` identifies the block within that class, and the `Offset` is the slot offset within the slot array.

3.3 Modular Design

Embedded devices have limited resources, and the availability of resources varies across different devices, such as RAM ranging from several MBs to GBs [28]. On the other hand, many embedded systems are deployed in application-specific scenarios [12]. There is a requirement for optimization or re-implementation of some functions of the system. Unfortunately, current storage systems lack a modular architecture and have several limitations. Some functions in the storage system may be unnecessary for certain applications, making

a bulky storage system unsuitable for tiny-scale and low-power embedded devices. Besides, upgrading or adding new modules becomes difficult. It requires significant effort to restructure or even reimplement them from scratch.

Lunar adopts a modular design to address these issues. The basic module is tiny (ROM 32.51 KB, RAM 0.32 KB) to adapt to resource-constrained embedded devices. Additionally, Lunar's modularity facilitates module upgrading and integration. Table 1 presents a list of implemented modules. These modules are classified into three categories: basic, system optimization, and table optimization modules. The basic module is required, while the following two are optional.

The basic module is the core of the table storage engine, providing fundamental storage capabilities required by any scalable table engine. This module consists of two parts: RAM management and device management (introduced in 3.2). As shown in Figure 5, RAM management includes basic metadata and basic I/O. Lunar maintains system-level (`Super`, `Root Table`) and table-level (`index`, `Schema`) metadata in RAM to expedite processing. The storage manager in the basic I/O part handles both reads and writes to storage devices.

System optimization modules provide services that apply to all tables in the system. Considering the performance gap between flash-based storage and RAM, we adopt a data buffer with Least Recently Used (LRU) replacement strategy to promote I/O efficiency. Prefetching is an active read-ahead technique that involves loading data into RAM before it is actually requested, thus mitigating performance issues caused by I/O delays. To avoid prefetching worthless data from random access, we utilize an adaptive strategy with variable prefetch sizes mentioned in previous work [21].

Each table can customize its table optimization modules. currently, Lunar provides two table optimization modules known as query cache and secondary index. It allows building secondary indexes (hash-based or tree-based) on any properties as needed. In contrast to typical tightly-coupled database solutions, Lunar offers these modules as optional, allowing for adaptability to resource-constrained embedded devices and facilitating upgrades.

3.4 Flexible Consistency Mechanism

Lunar employs journaling to ensure crash consistency for table data and metadata, rather than using log-structuring, to avoid data relocation that disrupts carefully planned layouts. Resources are often constrained in embedded devices, and different embedded applications have various consistency needs. Lunar makes a trade-off between resource consumption and consistency. In general, weaker consistency results in lower resource consumption and better performance. Lunar provides four consistency modes that range from weak to strong: `disorder`, `metadata`, `data`, and `full`.

For `disorder` mode, Lunar only maintains the metadata log. This mode has the weakest consistency but the lowest resource consumption, making it attractive for applications running on extremely resource-constrained devices with low consistency requirements. No data log is recorded, and data and metadata are not required to be written in a certain order. In this mode, inconsistency may occur if the system crashes while persisting data after the metadata log has persisted, resulting in the retrieval of garbage data after recovery.

Table 2. The Consistency Modes of Lunar.

Mode	Sync Metadata	Metadata	Order	Sync Data	Data
disorder	×	✓	×	×	×
metadata	×	✓	✓	×	×
data	×	✓	✓	×	✓
full	✓	✓	✓	✓	✓

Lunar ensures the `metadata` mode (default) on top of `disorder` mode guarantee. `metadata` mode enforces an ordering constraint that requires the transaction-related data write must be completed before the journal writes the metadata. Therefore, if the associated data size is large, the transaction commit latency will be lengthy. This mode guarantees the persistence order of both data and metadata to prevent the retrieval of garbage data. Since the data log is not recorded, system consistency will not be affected if the system crashes while writing data, but data operations may be partially completed.

The `data` mode is on top of `metadata` mode guarantee. `data` mode not only records the metadata log but also the data log, thus there will be no partial data operations. The `full` mode provides the strongest consistency guarantee. On top of the `data` mode guarantees, `full` mode also ensures that all operations are synchronous. The demand for storage and RAM is substantial due to the maintenance of the data log, making the `data` and `full` modes unsuitable for some extremely resource-constrained embedded devices.

4 Evaluation

4.1 Experimental Setup

We conduct experiments on a Kendryte K210 core board [42]. The platform is equipped with a 2-core 64-bit RISC-V processor, 6 MB of RAM, and a 32 GB Secure Digital (SD) card. All systems run on the RT-Thread (V4.1.1) [43], which is the state-of-the-art embedded operating system.

Benchmark. We run Mobibench [16], YCSB [11], and SQLite-bench [36]. Mobibench utilizes SQLite as its test database. SQLite is pre-populated with 10K rows of records (4 KB per row). We perform synchronous insert/update operations. For YCSB, we employ the settings as the previous works [3, 41]. Each tuple contains an 8-byte primary key and five 100-byte columns. We vary two parameters: 1) the read/write ratio: write-only (100% write), read-heavy (90% read / 10% update), balanced (50% read / 50% update), and update-heavy (10% read / 90% update), and 2) the skewness of the Zipfian distribution: None (skewness = 0), Low (skewness = 0.5), and High (skewness = 1). For SQLite-bench, we

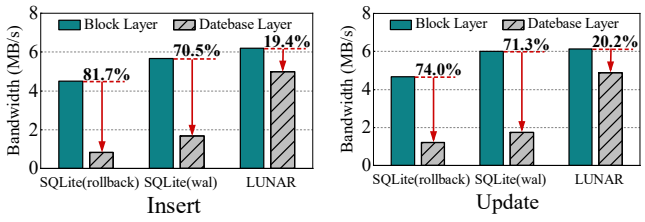

Figure 9. Mobibench Performance on Bandwidth.

compare the performance of systems for both read and write workloads, using both random and sequential access modes.

Compared systems. Considering that many embedded database systems are not open-source [17, 24, 41], we compare LUNAR with the widely used SQLite [44] and the latest database, FlashDB [2]. SQLite is a lightweight and high-speed database engine that has become the most widely deployed database engine in existence [13]. FlashDB focuses on providing data storage solutions for embedded products [2]. By default, LUNAR is equipped with the basic module and configured in `metadata` journaling mode.

4.2 Basic Performance

We used Mobibench to analyze how databases utilize underlying storage performance. We perform 100K SQL requests and measure two types of bandwidth: database-layer (i.e., SQLite and LUNAR) bandwidth, which refers to the amount of data received from the database, and block-layer bandwidth, which refers to the amount of data issued to the block layer. SQLite offers two journal modes for managing transactional data: rollback mode and write-ahead log mode[44]. In our evaluation, we considered these two journal modes. Based on the results of our previous experiments (§2.2), where F2FS performs the best performance, we decided to use F2FS for storing the SQLite tables.

Figure 9 shows the results. While performing insert operations using SQLite's rollback mode, there is an 81.7% bandwidth loss compared to the block layer. On the other hand, when SQLite operates in wal mode, there is a performance improvement compared to the rollback mode. This is due to the sequential nature of I/O operations, making it ideal for utilizing flash storage. However, there is still a 70.5% bandwidth loss relative to the block layer. The significant bandwidth loss of the SQLite is mainly caused by the uncoordinated interaction between F2FS and SQLite, resulting in substantial I/O activity due to multiple layers of abstraction. In contrast, LUNAR only incurs 19.4% performance loss compared to the block layer. Note that the abstraction of tables built on devices requires metadata maintenance, which inevitably results in some overhead. The key factor behind LUNAR's high bandwidth utilization is that it is a native table storage engine. Unlike file-based storage systems that involve multiple levels of abstraction, LUNAR eliminates file abstraction, thereby preventing I/O amplification. Moreover, through the elimination of file abstractions, LUNAR enables fine-grained data transfer between the storage device and RAM at the sector granularity, which further reduces bandwidth loss on the I/O path.

The experimental results for update and insert operations are consistent. In both rollback and wal modes, SQLite incurs significant performance losses of 74.0% and 71.3%, respectively, attributable to the multiple layers of abstraction involved. In contrast, LUNAR's cross-layer optimization enables it to achieve up to 79.8% bandwidth utilization.

4.3 Macro Benchmarks

4.3.1 YCSB. We run the YCSB benchmark with varying read/write ratios and skewness. LUNAR only utilizes the basic module and the system-level buffer module. The buffer size is set to 1 MB for all systems. Each system executes 20 million individual operations. As shown in Figure 10, under write-only workloads of various skewness, LUNAR's average latency is 49.7% and 57.2% of SQLite and FlashDB, respectively. This improvement is largely due to the cross-layer design which reduces software redundancy. By eliminating file abstraction and establishing a shorter I/O path, LUNAR accelerates I/O processing. Instead, SQLite and FlashDB tables are built on file abstraction, which causes many small, random, and synchronous I/Os due to the misaligned interaction of two-level storage abstraction. Furthermore, SQLite and FlashDB need to maintain two types of metadata, namely table metadata and file metadata.

For read-heavy workloads with varying degrees of skewness, LUNAR exhibits an average latency that is 27.9%-43.1% of that of SQLite. LUNAR's great efficiency comes from the fact that it removes the file abstraction, thereby enabling each request to simply traverse a mapping from the logical table offset to the physical location of the device. On the contrary, each SQLite request undergoes two mappings: one from the logical table offset to the logical file offset, and then another from the logical file offset to the physical location on the device. Furthermore, LUNAR eliminates file abstractions, enabling data transfer between the storage device and RAM at the sector granularity. This fine-grained transfer improves the I/O utilization.

FlashDB has an average latency that is 7.2-9.4× higher than that of LUNAR. Due to space limitations, only one symbol is shown. This is because FlashDB adopts an append-only out-of-place update strategy that is beneficial for writing but causes significant read amplification. Additionally, FlashDB lacks an efficient index to facilitate fast retrieval.

The performance of systems under update-heavy workloads follows a similar trend to write-only workloads. In this context, LUNAR achieves a greater performance improvement compared to write-only workloads. Because the update procedure involves read-modify-write operations, generates more I/O, and exacerbates the issue of software redundancy resulting from multiple layers of abstraction.

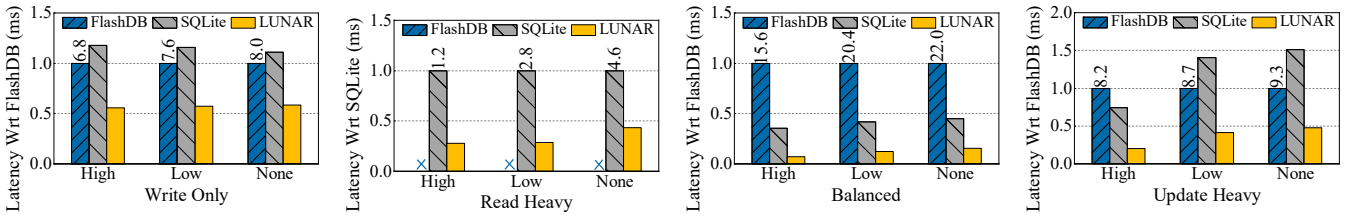

Figure 10. YCSB Performance on Latency.

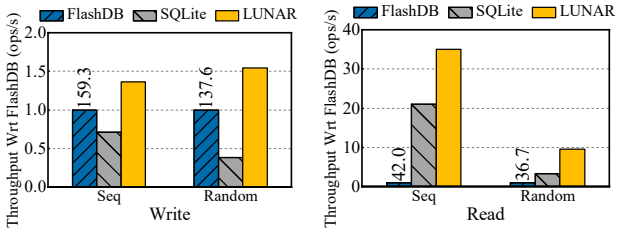

Figure 11. SQLite-bench Performance on Throughput.

4.3.2 SQLite-Bench.

We employ SQLite-bench to examine the performance of read and write workloads across both random and sequential modes. Figure 11 shows the results. Under both sequential and random writes, Lunar exhibits higher throughput, outperforming SQLite and FlashDB by 1.3-4.1×. Lunar's efficiency stems from two primary factors. First, unlike SQLite and FlashDB, which require a two-layer I/O stack consisting of a database and file system, Lunar performs cross-layer design to mitigate small random I/O resulting from misaligned interaction. Second, Lunar only maintains table metadata, whereas SQLite and FlashDB maintain two metadata for tables and files. FlashDB surpasses SQLite by 1.4-2.5× by employing out-of-place updates, while SQLite's in-place updates lead to more random I/O.

For read workloads, Lunar outperforms SQLite by 1.7-2.9×. This is attributed to Lunar's cross-layer design. Each request in SQLite must traverse two mappings (table-file-device), Lunar only requires one mapping (table-device). Lunar's fine-grained transfer between the storage device and RAM enhances the utilization of I/O. Note that due to FlashDB's out-of-place update design and lack of an efficient index, it exhibits poor read performance, which is consistent with the previous evaluation results shown in Figure 10.

4.4 Micro Benchmarks

Resource Consumption. Table 3 shows the ROM and RAM usage of several systems. Lunar (basic) only enables the basic module. In comparison to file systems that offer file-based storage services, Lunar provides rich structured storage capabilities for structured data. When compared to FlashDB, Lunar (basic) has richer table-level storage capabilities, with reductions of 8.2% and 62.5% in its ROM and RAM consumption, respectively. Compared to SQLite, ROM and RAM are reduced by 91.2% and 95.6%, respectively. For Lunar (all),

Table 3. Comparison of Resource Consumption.

System	ROM(KB)	RAM(KB)
Lunar (basic)	32.508	0.317
Lunar (all)	69.667	0.445
VFS+FatFs	29.745	0.846
FlashDB+VFS+FatFs	35.417	0.847
SQLite+VFS+FatFs	369.227	7.293

which enables the basic module as well as all system-level and table-level optimization modules, reduces the ROM and RAM consumption of SQLite by 81.1% and 93.9%, respectively. In summary, Lunar integrates the database and file system layers, resulting in lower resource consumption. Additionally, thanks to its modular architecture, Lunar's basic module can provide fundamental table-based storage services and adapt to extremely resource-constrained embedded devices. Furthermore, the modular design facilitates the upgrading of existing modules and the addition of new ones to meet the storage needs of diverse applications.

Different Consistency Modes. We utilized the YCSB to investigate the impact of different consistency modes on performance. The results are shown in Figure 12. The `none` implies that logging is not enabled in Lunar. Compared to `none`, under a write-only workload, `disorder`, `metadata`, `data`, and `full` lead to increased latency of 30.1%, 59.5%, 70.2% and 76.7%, respectively. For update-heavy workloads, the average latency for the four consistency modes increased by 36.8%, 58.9%, 73.4%, and 78.9%, respectively. Besides, the data storage consumption for `data` and `full` can be up to 5.7× that of `disorder` and `metadata`. Overall, for the four consistency modes, `disorder` provides the best performance with the least resource consumption, as it does not require synchronous persistence or data logging while providing the weakest consistency guarantees. Contrary to `disorder`, `full` provides the strongest guarantees but consumes the most storage resources and performs the worst.

Fragmentation. To verify the variable-size allocator's fragmentation advantage, we compared a fixed-size (i.e., 4 KB) allocator to our variable-size allocator. We age Lunar using the modified Geriatrix [23] tool by 5 GB of write activity within a 1 GB partition, which involves creating and deleting tables ranging from 8 KB to 512 KB in size. We evaluate fragmentation using the `layout score` metric [1]. A score of 1.0 indicates sequential allocation, while 0.0 indicates no sequential blocks. Results show that the fixed allocator and

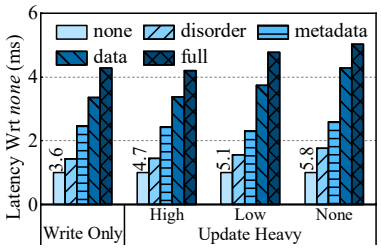

Figure 12. Performance with Different Consistency Modes.

variable-size allocator have `layout score` of 0.763 and 0.922, respectively, indicating that the variable-size allocator designed for storage devices exhibits lower fragmentation.

Different Data Volumes. We conducted an evaluation of performance for different data volumes using the YCSB with varying degrees of skewness under a balanced workload. Figure 13 shows the result. Under the uniform and skew workloads, as the dataset size increases from 256 MB to 1 GB, Lunar's performance drops by approximately 33.8% and 25.2%, respectively, due to buffer effects. Under uniform workloads, Lunar surpasses SQLite by 2.5-4.3× and FlashDB by 5.2-16.3×, with the improvement attributed to the cross-layer design that reduces software redundancy.

5 Related Work

Optimization on the I/O stack. Research efforts have been made toward various levels of the I/O stack to improve the efficiency of flash-based storage [15, 19, 21, 24, 29, 38, 41]. In the database layer, SQLiteKV [41] adopts the LSM-tree-based data structure but retains the SQLite operation interfaces. Jeong et al. [17] integrate external journaling to eliminate file system metadata journaling. In the file system layer, F2FS [27] is a flash-friendly file system that builds on append-only logging. LOFFS [45] is a low-overhead file System for large flash memory on embedded devices. In the block layer, iTRIM [29] considers the TRIM size and the logical addresses' pattern to reduce I/O contention. Han et al. [15] introduce a novel command queue-aware host I/O stack. In the FTL layer, OFTL [32] enables lazy persistence of index metadata and eliminates journals while keeping consistency.

Unlike existing works that mainly focus on a single layer of the I/O stack, Lunar is designed from a cross-layer perspective, integrating the database and file system layers. Latte [9] is a table storage engine designed for NVMe SSDs that are deployed on commercial servers, specifically for server-oriented solutions. Such solutions typically demand a substantial amount of resources to maintain a multitude of data structures, which makes them challenging to apply to strictly resource-constrained embedded devices.

Fragmentation. Fragmentation treatment can be divided into two categories, both active fragmentation avoidance and passive defragmentation [10, 14, 20, 35]. For fragmentation avoidance, FFS [35] introduces the idea of cylinder groups, which later evolved into block groups or allocation groups.

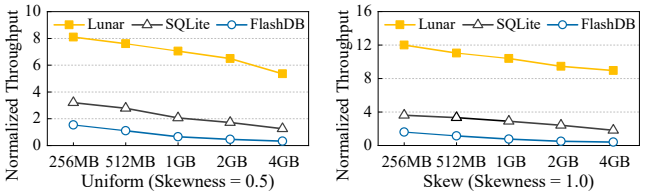

Figure 13. Different Sizes of the Dataset.

For defragmentation, `janusd` [14] achieves the effect of full file defragmentation without reducing the flash lifetime.

Distinct from traditional methods that optimize space utilization at the file system layer, Lunar improves space utilization by combining database access patterns. Moreover, Lunar uses fragmentation avoidance by employing a variable-size block allocator. Lunar can further reduce the impact of fragmentation by combining defragmentation techniques.

Modularity. The system's modular design facilitates loose coupling, enabling it to adapt to devices with varying resources and increasing its flexibility. TinyNet [12] is a lightweight, modular, and unified network architecture for representative low-power radio technologies including 802.15.4, BLE, and LoRa. The modular architecture of TinyNet simplifies the creation of new protocols by selecting specific modules in TinyNet. Unikraft [26] is a micro-library operating system whose components are modular and can be easily replaced or removed during compile time. To the best of our knowledge, Lunar is the first modular storage system that offers optional modules, delivering heightened flexibility in storage solutions.

6 Conclusion

This paper proposes Lunar, a novel native table storage engine for embedded devices. It performs a cross-layer design across the database and file system to avoid the pitfalls of layered abstraction and provides SQL-compatible APIs. The storage layout of Lunar is type-aware, considering the access patterns of different data types. To minimize fragmentation and reduce the waste of RAM and I/O bandwidth, Lunar employs a variable-size allocator. Further, given that resources on embedded devices are limited, Lunar adopts a modular architecture that enables module selection on demand. It also offers four optional consistency modes to balance resource consumption with consistency requirements. Experimental results demonstrate that Lunar achieves higher bandwidth utilization, better performance, and less resource consumption compared to state-of-the-art approaches.

Acknowledgments

This work was supported by the National Natural Science Foundation of China (Grant No. 62141214 and 62272171). Chuliang Weng (clweng@dase.ecnu.edu.cn) is the corresponding author.

References

[1] Nitin Agrawal, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2009. Generating Realistic Impressions for File-System Benchmarking. In *7th USENIX Conference on File and Storage Technologies (FAST)*. 125–138.

[2] Armink. 2020. *FlashDB*. https://github.com/armink/FlashDB

[3] Joy Arulraj, Matthew Perron, and Andrew Pavlo. 2016. Write-Behind Logging. *The VLDB Journal (VLDBJ)* 10, 4 (2016), 337–348. https://doi.org/10.14778/3025111.3025116

[4] Jens Axboe. 2007. *Block I/O Layer Tracing*. http://linux.die.net/man/8/blktrace

[5] Jeff Bonwick. 1994. The Slab Allocator: An Object-Caching Kernel Memory Allocator. In *USENIX Summer 1994 Technical Conference*. 87–98.

[6] Jeff Bonwick, Matt Ahrens, Val Henson, Mark Maybee, and Mark Shellenbaum. 2003. The Zettabyte File System. In *2nd USENIX Conference on File and Storage Technologies (FAST)*. 215–228.

[7] ChaN. 2009. *FatFS*. http://elm-chan.org/fsw/ff/00index_e.html

[8] Tseng-Yi Chen, Yuan-Hao Chang, Shuo-Han Chen, Nien-I Hsu, Hsin-Wen Wei, and Wei-Kuan Shih. 2017. On Space Utilization Enhancement of File Systems for Embedded Storage Systems. *ACM Transactions on Embedded Computing Systems (TECS)* 16, 3 (2017), 83:1–83:28. https://doi.org/10.1145/2820488

[9] Jiajia Chu, Yunshan Tu, Yao Zhang, and Chuliang Weng. 2020. LATTE: A Native Table Engine on NVMe Storage. In *36th IEEE International Conference on Data Engineering (ICDE)*. 1225–1236. https://doi.org/10.1109/ICDE48307.2020.00110

[10] Alexander Conway, Ainesh Bakshi, Yizheng Jiao, William Jannen, Yang Zhan, Jun Yuan, Michael A. Bender, Rob Johnson, Bradley C. Kuszmaul, Donald E. Porter, and Martin Farach-Colton. 2017. File Systems Fated for Senescence? Nonsense, Says Science!. In *15th USENIX Conference on File and Storage Technologies (FAST)*. 45–58.

[11] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking Cloud Serving Systems with YCSB. In *1st ACM Symposium on Cloud Computing (SoCC)*. 143–154. https://doi.org/10.1145/1807128.1807152

[12] Wei Dong, Jiamei Lv, Gonglong Chen, Yihui Wang, Huikang Li, Yi Gao, and Dinesh Bharadia. 2022. TinyNet: a Lightweight, Modular, and Unified Network Architecture for the Internet of Things. In *20th International Conference on Mobile Systems, Applications, and Services (MobiSys)*. 248–260. https://doi.org/10.1145/3498361.3538919

[13] Kevin P. Gaffney, Martin Prammer, Laurence C. Brasfield, D. Richard Hipp, Dan R. Kennedy, and Jignesh M. Patel. 2022. SQLite: Past, Present, and Future. *The VLDB Journal (VLDBJ)* 15, 12 (2022), 3535–3547.

[14] Sangwook Shane Hahn, Sungjin Lee, Cheng Ji, Li-Pin Chang, Inhyuk Yee, Liang Shi, Chun Jason Xue, and Jihong Kim. 2017. Improving File System Performance of Mobile Storage Systems Using a Decoupled Defragmenter. In *2017 USENIX Annual Technical Conference (ATC)*. 759–771.

[15] Kyuhwa Han and Dongkun Shin. 2020. Command Queue-Aware Host I/O Stack for Mobile Flash Storage. *Journal of Systems Architecture: Embedded Software Design (JSA)* 109, 101758 (2020). https://doi.org/10.1016/j.sysarc.2020.101758

[16] Sooman Jeong, Kisung Lee, Jungwoo Hwang, Seongjin Lee, and Youjip Won. 2013. Framework for Analyzing Android I/O Stack Behavior: From Generating the Workload to Analyzing the Trace. *Future Internet* 5, 4 (2013), 591–610. https://doi.org/10.3390/fi5040591

[17] Sooman Jeong, Kisung Lee, Seongjin Lee, Seoungbum Son, and Youjip Won. 2013. I/O Stack Optimization for Smartphones. In *2013 USENIX Annual Technical Conference (ATC)*. 309–320.

[18] Cheng Ji, Li-Pin Chang, Sangwook Shane Hahn, Sungjin Lee, Riwei Pan, Liang Shi, Jihong Kim, and Chun Jason Xue. 2019. File Fragmentation in Mobile Devices: Measurement, Evaluation, and Treatment. *IEEE Transactions on Mobile Computing (TMC)* 18, 9 (2019), 2062–2076. https://doi.org/10.1109/TMC.2018.2869737

[19] Cheng Ji, Li-Pin Chang, Riwei Pan, Chao Wu, Congming Gao, Liang Shi, Tei-Wei Kuo, and Chun Jason Xue. 2021. Pattern-Guided File Compression with User-Experience Enhancement for Log-Structured File System on Mobile Devices. In *19th USENIX Conference on File and Storage Technologies (FAST)*. 127–140.

[20] Cheng Ji, Li-Pin Chang, Liang Shi, Chao Wu, Qiao Li, and Chun Jason Xue. 2016. An Empirical Study of File-System Fragmentation in Mobile Storage Systems. In *8th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage)*.

[21] Cheng Ji, Riwei Pan, Li-Pin Chang, Liang Shi, Zongwei Zhu, Yu Liang, Tei-Wei Kuo, and Chun Jason Xue. 2020. Inspection and Characterization of App File Usage in Mobile Devices. *ACM Transactions on Storage (TOS)* 16, 4 (2020), 25:1–25:25. https://doi.org/10.1145/3404119

[22] Rohan Kadekodi, Se Kwon Lee, Sanidhya Kashyap, Taesoo Kim, Aasheesh Kolli, and Vijay Chidambaram. 2019. SplitFS: Reducing Software Overhead in File Systems for Persistent Memory. In *27th ACM Symposium on Operating Systems Principles (SOSP)*. 494–508. https://doi.org/10.1145/3341301.3359631

[23] Saurabh Kadekodi, Vaishnavh Nagarajan, and Gregory R. Ganger. 2018. Geriatrix: Aging What You See and What You Don't See. A File System Aging Approach for Modern Storage Systems. In *2018 USENIX Annual Technical Conference (ATC)*. 691–704.

[24] Woon-Hak Kang, Sang-Won Lee, Bongki Moon, Gi-Hwan Oh, and Changwoo Min. 2013. X-FTL: Transactional FTL for SQLite Databases. In *2013 ACM International Conference on Management of Data (SIGMOD)*. 97–108. https://doi.org/10.1145/2463676.2465326

[25] Hyojun Kim, Nitin Agrawal, and Cristian Ungureanu. 2012. Revisiting Storage for Smartphones. *ACM Transactions on Storage (TOS)* 8, 4 (2012), 14:1–14:25. https://doi.org/10.1145/2385603.2385607

[26] Simon Kuenzer, Vlad-Andrei Bădoiu, Hugo Lefeuvre, Sharan Santhanam, Alexander Jung, Gaulthier Gain, Cyril Soldani, Costin Lupu, Stefan Teodorescu, Costi Răducanu, Cristian Banu, Laurent Mathy, Răzvan Deaconescu, Costin Raiciu, and Felipe Huici. 2021. Unikraft: Fast, Specialized Unikernels the Easy Way. In *16th European Conference on Computer Systems (EuroSys)*. 376–394. https://doi.org/10.1145/3447786.3456248

[27] Changman Lee, Dongho Sim, Joo Young Hwang, and Sangyeun Cho. 2015. F2FS: A New File System for Flash Storage. In *13th USENIX Conference on File and Storage Technologies (FAST)*. 273–286.

[28] Borui Li, Hongchang Fan, Yi Gao, and Wei Dong. 2022. Bringing Webassembly to Resource-constrained IoT Devices for Seamless Device-cloud Integration. In *20th Annual International Conference on Mobile Systems, Applications and Services (MobiSys)*. 261–272. https://doi.org/10.1145/3498361.3538922

[29] Yu Liang, Cheng Ji, Chenchen Fu, Rachata Ausavarungnirun, Qiao Li, Riwei Pan, Siyu Chen, Liang Shi, Tei-Wei Kuo, and Chun Jason Xue. 2021. iTRIM: I/O-Aware TRIM for Improving User Experience on Mobile Devices. *IEEE Transactions on Computer-Aided Design of Integrated Circuits And System (TCAD)* 40, 9 (2021), 1782–1795. https://doi.org/10.1109/TCAD.2020.3027656

[30] Duo Liu, Yi Wang, Zhiwei Qin, Zili Shao, and Yong Guan. 2012. A Space Reuse Strategy for Flash Translation Layers in SLC NAND Flash Memory Storage Systems. *IEEE Transactions on Very Large Scale Integration Systems (TVLSI)* 20, 6 (2012), 1094–1107. https://doi.org/10.1109/TVLSI.2011.2142015

[31] Shaun Liu and Jack Heller. 1974. A Record Oriented, Grammar Driven Data Translation Model. In *1974 ACM Workshop on Data Description, Access and Control (SIGMOD)*. 171–189. https://doi.org/10.1145/800296.811511

[32] Youyou Lu, Jiwu Shu, and Weimin Zheng. 2013. Extending the Lifetime of Flash-Based Storage Through Reducing Write Amplification from File Systems. In *11th USENIX conference on File and Storage Technologies (FAST)*. 257–270.

[33] Matt Mackall. 2005. slob: introduce the SLOB allocator. https://lwn.net/Articles/157944/

[34] Avantika Mathur, Mingming Cao, Suparna Bhattacharya, Andreas Dilger, Alex Tomas, and Laurent Vivier. 2007. The New Ext4 Filesystem: Current Status and Future Plans. *Linux Symposium* 2 (2007), 21–33.

[35] Marshall K. McKusick, William N. Joy, Samuel J. Leffler, and Robert S. Fabry. 1984. A Fast File System for UNIX. *ACM Transactions on Computer Systems (TOCS)* 2, 3 (1984), 181–197. https://doi.org/10.1145/989.990

[36] Akira Moroo. 2018. SQLite3 Benchmark. https://github.com/ukontainer/sqlite-bench

[37] Daejun Park and Dongkun Shin. 2017. iJournaling: Fine-Grained Journaling for Improving the Latency of Fsync System Call. In *2017 USENIX Annual Technical Conference (ATC)*. 787–798.

[38] Hongwei Qin, Dan Feng, Wei Tong, Yutong Zhao, Sheng Qiu, Fei Liu, and Shu Li. 2021. Better Atomic Writes by Exposing the Flash out-of-band Area to File Systems. In *22nd ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)*. 12–23. https://doi.org/10.1145/3461648.3463843

[39] Mark Raasveldt and Hannes Mühleisen. 2019. DuckDB: an Embeddable Analytical Database. In *2019 ACM Conference on Management of Data (SIGMOD)*. 1981–1984. https://doi.org/10.1145/3299869.3320212

[40] Kai Shen, Stan Park, and Meng Zhu. 2014. Journaling of Journal Is (almost) Free. In *12th USENIX conference on File and Storage Technologies (FAST)*. 287–293.

[41] Zhaoyan Shen, Yuanjing Shi, Zili Shao, and Yong Guan. 2019. An Efficient LSM-tree-based SQLite-like Database Engine for Mobile Devices. *IEEE Transactions on Computer-Aided Design of Integrated Circuits And System (TCAD)* 38, 9 (2019), 1635–1647. https://doi.org/10.1109/TCAD.2018.2855179

[42] Sipeed. 2021. *Maix Bit.* https://wiki.sipeed.com/hardware/en/maix/index.html

[43] RT-Thread Development Team. 2019. RT-Thread. https://www.rt-thread.io/document/site/

[44] SQLite Development Team. 2017. *SQLite.* https://www.sqlite.org/index.html

[45] Runyu Zhang, Duo Liu, Xianzhang Chen, Xiongxiong She, Chaoshu Yang, Yujuan Tan, Zhaoyan Shen, and Zili Shao. 2020. LOFFS: A Low-Overhead File System for Large Flash Memory on Embedded Devices. In *57th ACM/IEEE Design Automation Conference (DAC)*. 1–6. https://doi.org/10.1109/DAC18072.2020.9218635

Received 2023-03-16; accepted 2023-04-21