Tiered Hashing: Revamping Hash Indexing under a Unified Memory-Storage Hierarchy

Jian Zhou
jianzhou@hust.edu.cn
Wuhan National Laboratory for Optoelectronics, HUST

Jianfeng Wu
magnetsaki20@hust.edu.cn
Wuhan National Laboratory for Optoelectronics, HUST

Weizhou Huang
huangweizhou@hust.edu.cn
Wuhan National Laboratory for Optoelectronics, HUST

You Zhou
zhouyou2@hust.edu.cn
School of Computer Science and Technology, HUST

Fei Wu*
wufei@hust.edu.cn
Wuhan National Laboratory for Optoelectronics, HUST

Liu Shi†
sl267311@alibaba-inc.com
Alibaba Group

Xiaoyi Zhang
shijing.zxy@alibaba-inc.com
Alibaba Group

Kun Wang
andrew.wk@alibaba-inc.com
Alibaba Group

Feng Zhu
f.zhu@alibaba-inc.com
Alibaba Group

Shu Li
s.li@alibaba-inc.com
Alibaba Group

ABSTRACT

NAND flash-based Solid State Drives (SSDs) provide a promising opportunity to enable the unified memory-storage hierarchy (UMH). The UMH renders a single memory address space for heterogeneous memories. Thus, the CPUs can directly access structured data in SSDs and eliminate bulk data copy/swap between the memory and storage devices. However, applying traditional indexing structures directly on SSDs may lead to poor performance. Particularly, the popular hash indexing generates highly randomized write traffic, incurring significant garbage collection overhead in SSDs. To address this problem, we propose a novel SSD-friendly hash indexing scheme called Tiered Hashing. It employs a multi-layer structure and opportunistic data movement (ODM) to construct skewed writes. Hence, the SSD can transform the writes into multi-streamed writes, where hot and cold data are separated to reduce GC overhead. Experimental results show Tiered Hashing reduces the average write latency and GC overhead by up to 94.98% and 90.71% compared to state-of-the-art hash indexings, without sacrificing read performance.

*Corresponding author.
†Work was done while at HUST.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
PACT '22, October 10–12, 2022, Chicago, IL, USA
© 2022 Association for Computing Machinery.
ACM ISBN 978-1-4503-9868-8/22/10…$15.00
https://doi.org/10.1145/3559009.3569652

CCS CONCEPTS

• **Computer systems organization → Processors and memory architectures**; • **Hardware → Memory and dense storage**; • **Software and its engineering → Layered systems**.

KEYWORDS

Hash Indexing, Unified Memory-storage Hierarchy, Multi-stream SSD

ACM Reference Format:
Jian Zhou, Jianfeng Wu, Weizhou Huang, You Zhou, Fei Wu, Liu Shi, Xiaoyi Zhang, Kun Wang, Feng Zhu, and Shu Li. 2022. Tiered Hashing: Revamping Hash Indexing under a Unified Memory-Storage Hierarchy. In *International Conference on Parallel Architectures and Compilation Techniques (PACT '22), October 10–12, 2022, Chicago, IL, USA*. ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3559009.3569652

1 INTRODUCTION

Many modern applications expect in-memory databases to simultaneously deliver DRAM-like low access latency and large storage capacity. On the one hand, hash tables are becoming the most privileged indexing structure because they perform point queries, including lookups and insertions, at constant time complexity (O(1)) regardless of the inserted data amount compared to a tree-like indexing structure. For example, mainstream in-memory databases, such as Redis [4] and Memcached [26] employ hash indexing for fast data access. On the other hand, due to the growing conflict between the large working data sets and the high cost to scale main memory, embracing a unified memory hierarchy (UMH) to extend DRAM with more cost-effective high-end SSDs [5, 9, 19, 24, 39] is becoming a promising and practical method. UMH provides in-memory applications with extra benefits, such as extended memory capacity and persistent data storage, with only a few or no code changes.

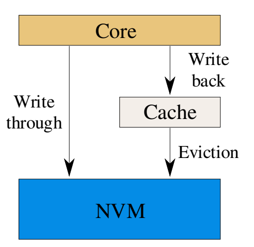
However, simply deploying hash indexing under UMH ignores the distinct features of underlying SSDs, thus incurring significant performance degradation. The main reason is that the access pattern of hash indexing is highly randomized, which is unfavorable with Nand flash SSDs. Because NAND flash is only page-addressable and because of its distinct erase-before-write feature, random write traffic from hash indexing can overburden the garbage collection (GC) in SSDs[1]. In addition, given the above limitation of Nand flash, UMH relies on either host or device memory to enable byte-addressability for SSDs. However, data written cannot be efficiently cached and should persist to NAND flash immediately. This further exacerbates GC overhead and reduces the lifetime of SSDs.

One of the most promising methods to improve write efficiency and reduce the GC overhead of SSDs is the multi-stream technique. It stores pages with different updating frequencies into separate logging areas. Hence, data in the same logging would invalidate simultaneously, then GC for frequently updated pages will not touch infrequently updated pages and vice versa. Nonetheless, hash indexing does not show such skewed writes since a given key is randomly mapped to the hash table by a set of different hash functions. Keys with varying update frequencies are highly likely to collocate to the same bucket. Hence, none of the existing hash indexings can leverage multi-stream technology to improve write efficiency.

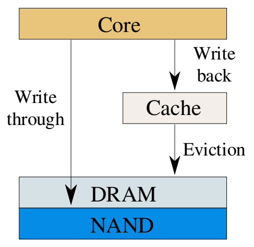
This paper aims to develop an efficient hash indexing that renders skewed write traffic that is more GC-friendly without sacrificing the point query performance. However, there are several critical challenges. First, skewed write does not come as a free lunch. A naive solution is to borrow the hierarchical data movement used by many tree-based indexings to generate naturally skewed writes. However, we find that the computation and storage overhead of corresponding data movement overshadows the benefits it brings. Second, flushing and persisting data under an SSD-based UMH introduce significantly more latency than other persistent memories; however, existing hash indexings issue excessive memory writes that hamper performance [26]. On the one side, many hash indexing applies key-value relocation to solve hash collisions, and each relocation would accomplish multiple flush operations. On the other side, persistent memory-optimized hash indexings employ write-ahead logs (WALs) and flush crucial write to prevent system crashes and ensure data durability, thus significantly reducing performance.

We propose Tiered Hashing to overcome the high GC overhead raised by traditional hash indexing against SSD-extended main memory under UMH. First, we employ a hierarchical structure and provide the range of each layer to SSDs. By doing so, different stream IDs can be tagged to each layer dedicatedly. Second, we invent an *opportunistic data movement (ODM)* strategy based on the hierarchical layout to build skewed writes by moving data from upper to lower layer beforehand, enabling the hash indexing to leverage the multi-stream feature of modern SSDs. Third, we adopt *maximum one flush policy* and *in-cacheline crash consistency* to merge multiple flushes into one, which solves the dilemma that the ODM introduced data movement could overshadow the benefits it brings. Finally, we apply *shadow reading* to optimize point queries as moving data to the lower layer increases the average length of the read path. We build a mathematical model in Section 3.7 to prove that Tiered Hashing can generate highly skewed writes. Our experimental results show that compared to state-of-the-art hash indexings, Tiered Hashing reduces the average insert/update latencies by up to 89.12%/94.98% and reduces the GC overhead by up to 87.21%/90.71% during insertions/updations.

[1]GC relocates valid flash pages before erasing a whole flash block. The overhead of GC primarily depends on the number of valid flash pages it needs to migrate. Random workloads may spread more valid pages across blocks, thus incurring higher overhead.



(a) Write schemes for NVM.



(b) Write schemes for UMH.

Figure 1: Write-through and Write-back under UMH.

The rest of this paper is organized as follows. Section 2 describes the background and motivation. Section 3 presents the design. Section 4 evaluates the performance and Section 5 concludes.

2 BACKGROUND AND MOTIVATION

This section discusses the background and motivates our work.

2.1 Unified Memory-Storage Hierarchy

The traditional memory hierarchy treats SSDs as secondary storage. Application requests must go through multiple hardware/software layers to copy the SSD data into the main memory, resulting in tremendous software overhead. As the performance of high-end SSDs increases, the corresponding software latency constitutes a significant portion of the entire delay. Thus, researchers employ the Unified Memory Hierarchy (UMH) to solve this problem, rendering a memory-like access mechanism for SSDs.

There are several different technologies to enable UMH. First, the **software-based** UMH mainly adopts the memory-mapped I/O interface (e.g., mmap), which in turn leverages the virtual memory in modern operating systems (OSes) [17, 30]. The mmap interface enables any block devices to be byte-addressable by mapping an application's virtual memory directly to a device file without any requirement of hardware support. The access to the memory-mapped virtual memory regions results in page faults. The OSes then transparently load the corresponding page from SSDs to DRAM. Since no hardware change is required, this method has been increasingly deployed in recent years. For example, many widely deployed in-memory databases, such as MongoDB [3], LMDB[1] use it to extend the DRAM capacity. However, page fault handling still incurs considerable performance overhead. Second, **hardware-based** UMH equips byte-addressable SSDs [5, 9, 11, 19, 24, 39], CXL-enabled SSD [16, 20] or adds a coprocessor [2, 24] to access SSDs directly from the processors, thus further reducing the software overhead. The former requires an advanced interconnect such as CXL [16, 20], PCIe [9, 19, 24], NVDIMM [11], or the combination

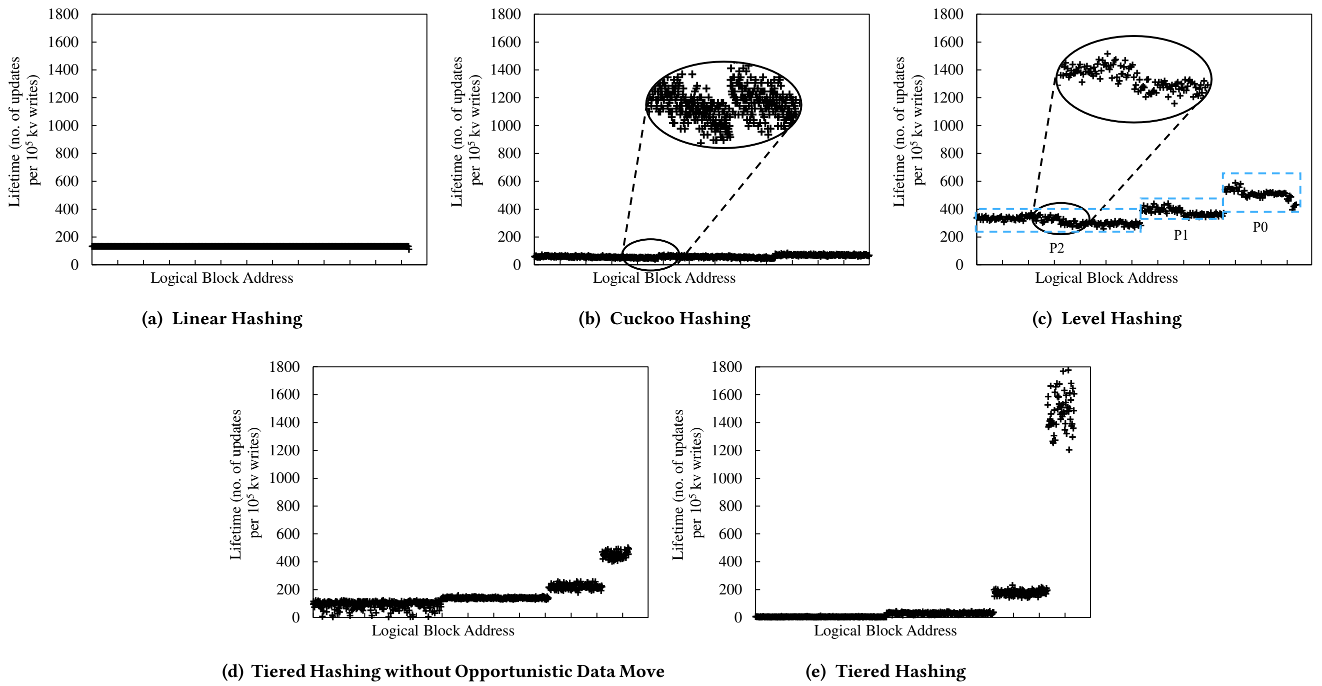


Figure 2: I/O characteristic of Different Hashing Indexings. The major difference is the distribution of updating frequency over LBAs, indicated by the y-axis.

of the two [39]. The latter changes the CPU [24] or GPU [2] to support an SSD protocol, such as NVMe. Hardware-assisted page swap can then cache popular data in DRAM to improve performance further [5, 22, 24, 39]. Both software and hardware-based UMH significantly extend memory capacity and enable data persistency like non-volatile memory (NVM) with more cost-effective SSDs.

Note that our proposed Tiered Hashing is compatible with both UMH solutions, and we assume hardware-based UMH in this paper. Unlike non-volatile memory (NVM), we further distinguish how to write data to SSDs and study their impact on performance and crash consistency. As shown in Figure 1, all UMH design requires either host or on-disk memory to hide the distinct feature of NAND flash. Different from NVM, we define two write schemes for SSDs. First, the write-through scheme allows the critical writes, e.g., checkpointing and snapshot writes, to bypass the DRAM cache. A memory fence instruction can then follow to prevent write reordering. Second, the write-back scheme allows other regular writes to leverage the CPU and DRAM cache to respond faster. The memory controller will write the NAND only when it evicts the dirty cache. Although the write-through and write-back schemes vary, there is no difference between the two from the NVM's perspective. However, while deployed under UMH, SSDs need to distinguish the critical and regular writes to improve performance, and these two writes schemes can be supported under either UMH. We will elaborate on how to utilize this feature in Section 3.3.

2.2 Multi-Stream SSDs

Multi-stream SSDs allow application and system software to assign a stream ID to the data with a similar lifetime during writing [38]. They then place the data with the corresponding stream ID into one flash block to improve the GC efficiency because data that belong to the same block are likely to be invalidated simultaneously. Rather than assign a stream ID in each request, Zoned Namespace (ZNS) SSDs divide logical-block addresses (LBAs) into multiple zones and save data on different zones to different erasing blocks [10, 18]. Multi-stream SSDs and ZNS SSDs share the same basic concept, allowing applications to advise the expected data lifetime to the device driver. The Multi-stream SSDs do not require application changes because they still offer a block I/O interface. However, ZNS SSDs mandate applications to change their method of utilizing the LBAs. This paper borrows the design of two. We submit a range of LBA to the SSD to distinguish the data lifetime and assume the underlying SSD equipt a flash translation layer (FTL) to provide a standard block I/O interface.

Other than the interface to advise data lifetime, another important aspect is how to identify and categorize the data's updating frequency. To achieve this, researchers develop various technologies to identify data with different lifetime [37] or redesign the software to write separately [31]. However, multi-stream SSDs work more efficiently only when the application's writes are notably skewed. For example, in RocksDB [21] which employs a tree-based indexing, different layers in the log-structured merge tree (LSM) have distinct updating frequencies naturally. Researchers thus seek ways to manually change the applications to specify a stream ID in each writes. Also, automatic stream management, such as FStream [31], AutoStream [37], and PCStream [21], makes stream allocation decisions transparently for the applications.

Note that both manual and automatic stream management requires the data lifetime to be naturally skewed [21, 23]. However, this requirement is not valid for many in-memory applications.

First, in-memory applications usually perform more random requests, especially hash indexing. Second, the number of streams in commercial SSDs is typically limited to only 4 to 16 [21, 36–38], which restricts the flexibility of memory allocation for in-memory applications further.

2.3 Hash Indexing

In recent years, researchers have proposed several new hash schemes, such as CCEH [27], Level Hashing [41] and Path Hashing [40], for persistent memories to guarantee the crash consistency based on conventional hash indexings, such as Cuckoo Hashing [29]. However, they focus more on NVMs including ReRAM [6], PCM [33], SST-MRAM [7] and 3D XPoint [28], which endure intensive random and in-place accesses better than flash. It is still essential to study how to develop SSD-friendly hash indexing while underlying persistent memory is NAND flash. Several works [32, 35] employ Linear Hashing [25] to maintain many small logs and perform more sequential writes. However, they may sacrifice the read performance and also cannot benefit from multi-stream technologies. In the following, we analyze different hash schemes to motivate our work.

One critical issue of hash indexing is handling hash collisions when multiple keys map to the same bucket. To reduce collisions, both Cuckoo Hashing [29] and Level Hashing [41] allocate multiple slots in one bucket and employ two or more hash functions to map a given key to multiple buckets. If one of the hash functions cannot locate a bucket with an empty slot, they will try another one. If none of the functions can identify such an open slot, they will attempt to move an existing key-value pair in the corresponding buckets using alternative hash functions. The difference is that Level Hashing only allows a maximum of one movement per insertion, while Cuckoo Hashing repeats data movement until it succeeds or reaches a threshold.

If all the above strategies fail, they will resize by allocating a larger hash table and moving all data to the new one. For example, Cuckoo Hashing [29] reallocates a new table that is two times larger than the old one and copies all existing key-value pairs. To reduce the resizing latency, Level Hashing [41] maintains two layers of hash tables, where the top layer is two times larger than the bottom layer. Level Hashing resizes the bottom layer to a four times larger "new" top layer so that the total table size doubles but only needs to move 1/3 of the data. Although in two consecutive hash resizing, Level Hashing still has to move all the data as described in [27], they do reduce the long-tail latency by performing two relatively small resizes. Clevel Hashing [12] offloads the resizing to background threads. CCEH [27] revises Extendible Hashing [14] to support crash consistency. It only resizes the particular bucket that has collision. However, it also introduces many small memory reallocations, making it impractical to apply on multi-stream SSDs under UMH.

2.4 Motivation

Lack of Write Skewness: To study the lifetime distribution in hash indexings, we insert 680 million key-value pairs (20GB data) to illustrate the lack of skewness of existing hash indexing. By configuring the size of different hash tables, we make each of them

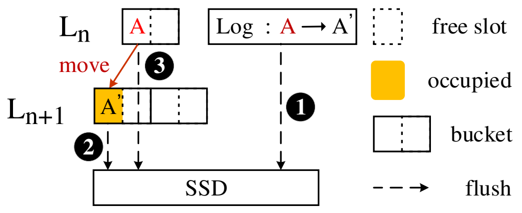


Figure 3: Tripple Flushes per KV Movement

resize twice in our comparison experiment. Figure 2a to 2b illustrate the access randomness of three hashing schemes by showing their number of writes (lifetime) on different LBAs.

Existing hash indexings show little skewness. The Level Hashing initializes 2^{25} buckets on the top level and 2^{24} buckets on the bottom level; each bucket contains four slots. Thus, it resizes two times in the experiment. We can see three parts in Figure 2c. The right part (P0) is its lifetime before resizing, the middle (P1) is the lifetime between two resizing, and the left (P1) is the lifetime after the second resizing. Though Level Hashing employs a hierarchical structure, the lifetime's hierarchy is not obvious in each part. The difference in lifetime distribution over LBAs is rather trivial; thus, it can hardly benefit from Multi-stream SSDs. Also, as shown in Figure 2a and 2b, the lifetime distribution of Cuckoo and Linear Hashing over LBAs is even less distinguishable than Level Hashing because they do not employ a hierarchical structure. The random writes to SSDs will cause significantly higher write amplification. Hence, it is crucial to develop a more SSD-friendly hash indexing.

Hash Movement incurs extra flushing overhead: Enabling key-value pairs' movement between multiple buckets is a primary way to help reduce the necessary rehashing. However, each move operation results in at least two consecutive flush operations in the critical execution path - first flush the destination bucket, then flush the source bucket with the new data. Moreover, a write-ahead log (WAL) is also necessary to guarantee data consistency and prevent data loss in a sudden crash because the movement involves multiple updates. Specifically, as shown in Figure 3, a reasonable strategy is to first write to the WAL to trace the moved key. It then writes to the destination slot and flushes it to NAND flash. Finally, it removes the data in the source slot and flushes it to inform SSD of a deletion. Such a movement strategy results in a considerable amount of writes and flushes to SSD, which may even overshadow the benefits brought by other hash optimizations.

3 TIERED HASHING DESIGN

This section describes how Tiered Hashing works and explains why it can be SSD-friendly.

3.1 Layout of Tiered Hashing

Equivalent to many other contenders, Tiered Hashing employs two hash functions to mitigate hash collisions.[2] Then, Tiered Hashing employs a hierarchical structure, which provides an opportunity to differentiate the lifetime of underlying NAND pages in each layer, such that we can achieve noticeable skewness to leverage

[2]As two hash functions result in a maximum load factor of around 95%, it does not make sense to use more than two hash functions [12, 29].

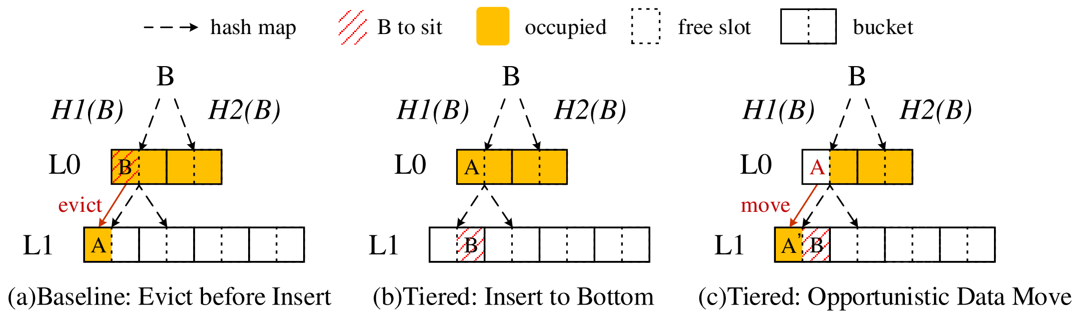

hash map
B to sit
occupied
free slot
bucket
B
H1(B)
H2(B)
L0
L1
evict
move
A
B
(a)Baseline: Evict before Insert
(b)Tiered: Insert to Bottom
(c)Tiered: Opportunistic Data Move

Figure 4: Comparison of different insertion schemes. All three figures dipict the case after insertion is finished. Assume a bucket contains two slots and A is initially sitting in the left-most slot of L0. Dashed arrows point to the middle of buckets.

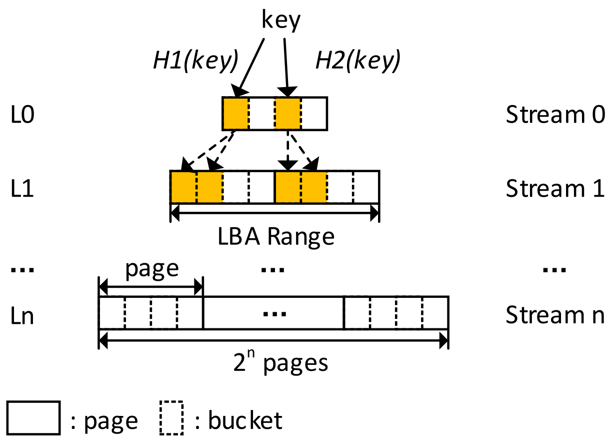


Figure 5: The Tiered Hashing Layout.

the multi-stream feature. As shown in Figure 5, Tiered Hashing consists of multiple layers of the hash table and maps each layer to a fixed region on the SSD. In Tiered Hashing, each layer is 2^n times larger than the layer above (where n is an integer). Each layer consists of multiple buckets located by the two hash functions, and each bucket contains four slots. Each slot stores a 16-byte key and a 15-byte value, which is large enough for most key-value pairs in Facebook's key-value store [8]. To support larger KV sizes, we would save the data in a separate log and then index a pointer to the log item, which is out of the scope of this paper.

Combining of Multi-stream SSDs With Tiered Hashing: Multi-stream SSDs allow applications to assign a stream ID with the write command. In our hardware-managed UMH architecture, the requests to SSDs are a set of memory writes transferred via PCIe. Tiered Hashing provides the range of each layer to the SSDs. The SSDs then assign a stream ID to each memory range. The SSD firmware determines the corresponding stream ID when a write request arrives based on the range table. Employing specified hardware can accelerate such searching processes. This design is feasible because the layers in Tiered Hashing are relatively stable and limited in number.

Benefits of Tiered Structure: The pyramid-like structure in Tiered Hashing lets buckets directed by the same hash value in different layers be affiliated with each other. New KVs are inserted into an upper bucket first; they can then move to the lower bucket together when the upper one is full. Such a design opens up free slots and leaves the LBA range of upper layers updated more frequently. Since each layer is associated with a fixed LBA range, the disk can then distinguish the update frequency based on LBAs and leverage the write skewness. As shown in figure 2d, Tiered Hashing exibits better skewness than existing ones.

3.2 Insertion Scheme of Tiered Hashing

One of the main differences between Tiered Hashing and existing hash indexings lies in the insertion policy. When a new KV arrives, the hash indexing will first try to find an empty slot in the bucket directed by the first hash function from the top layer. If failed, existing hash indexing will try to search the alternative bucket in the same layer directed by the second hash function. If both buckets are full, the hash indexing will randomly pick up a slot, move the KV inside it to its alternative bucket, and free the slot before the new KV is inserted. As shown in Figure 4(a), existing hash indexing picks A for eviction; it first tries to relocate A to the same level. Such an eviction could trigger an iteration over the whole layer until a free slot is found or no alternative bucket exists. Since the top layer has no alternative bucket for A, it will evict A from the source slot in the top layer to the destination slot in the next layer, introducing triple flushes. Then, B is inserted after A is migrated successfully to the second layer and the source slot occupied by A is freed, thus incurring long-tail latency. In contrast, when both buckets are full for B, Tiered Hashing will directly try to find a free slot in the second level by using one more bit of the hash value and leaving A in the original position. As shown in Figure 4(b), only one flush is needed to insert B in the second level. Note that Tiered Hashing will iterate the above lookup procedure until a free slot is found. Resizing happens when it cannot solve a collision at existing levels. Such a design also differs from LSM-tree since inserting new KVs will not evict elders in the upper layer and avoid table compaction, thus significantly reducing write traffic.

3.3 Opportunistic Data Movement

To further enhance the skewness of hash indexing, Tiered Hashing performs an opportunistic data movement (ODM) strategy during a regular insertion. As shown in Figure 4(c), Tiered Hashing searches an open slot for B from the top layer to the bottom layer until success. While inserting into a lower layer, the ODM strategy will

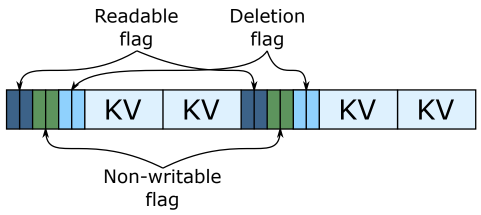


Figure 6: Flags for Tiered Hashing.

simultaneously move down the data (from buckets indexed by *hash(b)*) from the upper layers if possible, thus opening up free slots in upper layers. Such a strategy enables skewed writes between layers because moving the data from the upper layer will sink the data to the lower layer. It works because cold data does not update frequently, then they sink to lower layers accordingly. In contrast, hot data won't sink and remain in the upper layers. Thus, it helps generate skewed writes between different layers, which will benefit from Multi-stream SSDs.

Maximum One Flush Policy: Although our opportunistic data movement generates Multi-Stream SSD-friendly skewed write between layers, the crucial problem is that such a policy incurs numerous costly flushing operations. A maximum one flush policy is used to mitigate extra migration overhead introduced by the ODM strategy for each updating or insertion without sacrificing the crash consistency. We find extra data movement incurred by our proposed data movement can be carried out during the KV insertion operation. The rationale behind this is that the write traffic incurred by data movement can be cached and wait for the write-back, thus eliminating the costly flushing illustrated in Figure 3. In addition, since data movements and queries follow the same single direction, and no extra data is altered except for the inserted one, there is always an up-to-date copy for opportunistically moved data. Therefore, even a WAL log won't be needed. Moreover, as all the movement destinations in opportunistic data movement always colocate with the page that is serving the updating or insertion request, Tiered Hashing only needs to perform one flush operation towards the destination page after finishing all the writes.

3.4 Shadow Reading

Since searching starts from the top layer, migrating key-value pairs during insertion and updating reduces the upper layers' hit ratio. Therefore, more layers would be searched before a key is finally located, thus increasing the average search latency. Tiered Hashing develops shadow reading that allows the source slot in upper layers to serve both queries and insertions after an opportunistic data movement by employing two bitwise flags to enable the access control for each slot. A source slot can simultaneously serve query and insertion because Tiered Hashing will have duplicated keys in both source and destination slots after performing the ODM; thus, overwriting the source slot does not result in data loss. Specifically, we mask the source slot as writable and apply shadow reading rather than removing the data during opportunistic data movement. As a result, as data 'A' in red in Figure 4(c) shows, the pending reads can still access the source slot before another write overrides it. To achieve this, we leverage the extra free space in the metadata area to track the writable state for each slot.

Table 1: Truth Table for Shadow Reading

readable flag	non-writable flag	Description
0	0	Open slot (Initial)
0	1	N/A (Impossible)
1	0	Shadow slot
1	1	Valid slot

Figure 6 shows the metadata structure for slots in Tiered Hashing, while Table 1 lists the state of two flags regarding three different states of a slot. Tiered Hashing employs two flags for each slot to indicate its "readable" and "non-writable" states. Initially, they are both false, which means this slot is unreadable and writable.

To enable shadow reading without compromising the crash consistency, Tiered Hashing adopts a novel updating scheme while moving key-value pairs between layers. For the slots in the upper layers, we mark their "readable" flags as "1" and "non-writable" flags as "0". Thus, the slots in the upper layers are still readable, and we do not have to search the lower layers for reading. Meanwhile, as the "non-writable" flag is 0, the slot would be open for incoming writes. We will mark the "readable" flag as "0" before writing a new key-value pair.

Benefits brought by colocating flags and data in the same cacheline: To further reduce the overhead of flushing for data consistency and logging, we carefully design the placement of flags. If we can guarantee the order between the write of data and flags, we can eliminate logs and extra flush operations. Fortunately, while performing multiple writes to the same cacheline, the writing orders to the cacheline is equivalent to the order they reach the persistent memory [13]. Such an order can be guaranteed with release memory ordering supported in C++11, or the fence instruction on X64 architecture, both of which incur no runtime overhead. Therefore, Tiered Hashing substitutes WAL with the flagging mechanism and places each slot and its metadata in the same cacheline to eliminate extra flushes to guarantee consistency while updating the metadata and slot. This is feasible because, for 15-byte keys and 16-byte values, each cacheline can place two key-value pairs plus a 2-byte metadata area that saves 2-bit readable, 2-bit non-writable, and 2-bit deletion flags. As a comparison, existing hash indexings, such as Level Hashing, place the metadata in the header of each slot, failing to guarantee all flags and their corresponding data are located in the same cacheline.

3.5 Lazy Updating and Deletion

Since the ODM strategy opens up more empty slots in the above layers, Tiered Hashing adopts the lazy updating and deletion schemes to enhance skewed writes. The lazy updating and deletion scheme encodes the operation in an open slot in the above layers rather than directly updating the lower layer. Tiered Hashing places the new value of a key in the above layers. Hence during the read, it searches the above layer before entering the lower layer to ensure getting the updated value. Also, Tiered Hashing expresses deletion of a key using either a reserved value, such as zero, or a flag in the extra metadata area, such as the deletion flags in Figure 6. Later, during the ODM operation, the corresponding updating and deletion then sink to the lower layer.

3.6 In-Cacheline Crash consistency

Based on the above design and analysis, we use multiple strategies to guarantee crash consistency under different scenarios.

First, when inserting data into an open slot, once we write the key-value pair, we can alter the readable flag of the slot and persist both data and flags using one flush to make its data ready for reading. This design does not require an extra write-ahead log (WAL) to guarantee consistency because the write to the flag is atomic.

Second, when writing data to a shadow slot, we set the readable flag to 0 before writing the new data and reset both the readable and non-writable flags to 1 afterward. Since all writes are in the same cacheline, we only protect the order we update the flags and values and perform one flush until all writes are finished.

Third, when migrating the data, we simply use the above methods to write to the destination slot. After safely writing to the destination, we remove the data from the source by changing the corresponding flags. However, we avoid flushing the source slot after that. If the system crashes, we will scan the table to search for duplicated KV pairs. If two layers contain the same KV pair, the one in the upper layer can be safely removed. Since the path of data movement is short and the direction is fixed, the recovery only needs to scan limited buckets and can be made incrementally by checking all buckets on each layer directed by the same hash value during regular workloads.

Finally, when updating a key-value pair, we first seek upper layers to find an open or shadow slot to place the data. If success in finding an available slot, we use the first or second strategy to insert the data. Then, we wait until the ODM strategy move down the updated value. If failed to find an available slot in the upper layers, we then update the corresponding value "in-place". Because we support 16-byte values, which exceed the 8-byte maximum atomic updating size, this is the only case we employ a WAL to protect the "in-place" value override.

3.7 Modeling Skew Factor

Tiered Hashing generates skewed writes between layers by employing ODM. Here, we model the skew factor to demonstrate its effectiveness. Note that we use a pure random workload to test the skewness generated by our design alone. The skew factor in this paper indicates the ratio of updating frequency in an upper layer versus the lower layer. For simplicity, we adopt two layers in Tiered Hashing. We denote the distance between the two layers as r, which means the size of the lower layer is 2^r times as much as the upper layer. The insertion process contains the following three phases. In the first phase, new key-value pairs will initially hit the upper layer with open slots. When the upper layer is full in the second phase, the subsequent insertion will hit the lower layer and trigger opportunistic data movement. Suppose a perfect hash function randomly maps the keys to the table; the ODM process will move one over 2^r of key-value pairs from an upper layer's page down to the lower layer on average. In the third phase, the upper layer will open up free slots again, and the subsequent insertion will hit the upper layer until it is full for another time. Then, the subsequent insertion will trigger phase two once more. Assume that a page has N key-value pairs, and there are X writes to the

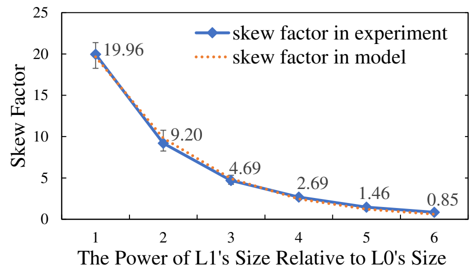


Figure 7: The skew factor, which equals to the proportion of writes to layer 0 and layer 1 as the two layers' distance increases from one to six.

lower layer. According to the description above, the skew factor F is:

$$F = \lim_{X \to \infty} \frac{N + \frac{N}{2^r} \cdot P \cdot X}{X} \approx \frac{N \cdot P}{2^r}$$

The factor P denotes the average percentage of key-value pairs that Tiered Hashing moves from an upper layer's bucket when it performs the ODM strategy. It differs according to the number of key-value pairs in source and destination buckets in two layers. As the number of key-value pairs inserted increases, the impact of the first phase decreases, and the impact of the second and the third phase becomes dominant.

Figure 7 shows the skewness of writes between two layers when the factor r increases from one to six. We set the top layer (L0) to have 1024 buckets and the bottom layer (L1) to have $1024 \cdot 2^r$ buckets, inserting key-value pairs into the hash table until resizing is needed. The number of key-value pairs inserted ranges from 11k to 185k. Experimental results show that the skew factor decreases from 19.96 to 0.85 as r increases from one to six. Note that when r is six, the skew factor is less than one, which means it is reversed, and the write frequency in L1 is higher than that of L0. Our mathematical model matches well with the experimental results when P is 30.7%.

Note that the above mathematical analysis assumes a simple random KV workload. When the KV insertions are skewed like many real-world workloads, such as the YCSB [15], our ODM can generate even higher skewed writes because the keys that are updated less frequently will gradually sink to the bottom layers. However, other hash indexings will mix keys with different updating frequencies.

3.8 Resizing Scheme

Tiered Hashing adopts a cost-efficient resizing scheme by adding a new bottom layer two times larger than the layer above. We lock the write permission of the top layer. While inserting into the bottom layer, we then apply the ODM scheme to move down the data from the upper layers. We can safely remove the top layer until we move all the data within the top layer down.

4 EVALUATION

In this section, we evaluate our design and analyze the results.

4.1 Evaluation Methodology

Our evaluation uses an in-house multi-stream SSD emulator similar to FlatFlash [5]. The emulator divides the host memory into three regions: the first region represents a regular host DRAM; the second region models the SSD-Cache; the third region simulates the NAND flash. To track the pages cached in the SSD-Cache and to inject memory access latencies of the SSD regions, the emulator uses *mprotect* to control the protection bit in the page table.

We experiment on a Linux Server (Linux 5.4.0-53-generic) with 40-core Intel Xeon E7-4820 2.0GHz CPUs (each core has 32KB L1 instruction cache, 32KB L1 data cache, 256KB L2 cache, and 15MB last level cache). The SSD configuration is described in Table 2, which is comparable to Tiny-Tail Flash [34] and FlatFlash [5]. We configure the SSD capacity to be 16GB and set the size of the SSD-Cache to be 0.1% of the SSD capacity.

To gain deep insights, we compare different mainstream indexings with various configurations as baselines. The Cuckoo [29] and Linear Hashing are configured to use one stream. To understand whether Level Hashing [41] can benefit from Multi-Stream SSD, we configure default Level Hashing to use one stream (denoted as Level). In Level-stream, we bind the top and bottom levels to different streams (denoted as LevelS). To prove that our solution has similar GC overhead with tree-like indexings, we also configure LevelDB — an LSM-based KV engine — as one of the baselines. The LevelDB is tailored for UMH to persist its data via memory interface. Finally, Tiered Hashing attaches each level to a separate stream, denoted as X-Tiers, where X is the number of levels. Note that Tiered Hashing employs the lazy updating and deletion scheme and multi-stream feature by default.

We first use synthetic random workloads in all experiments until Section 4.5 to demonstrate the effectiveness of Tiered Hashing and analyze why it is SSD-friendly. We then employ YCSB [15] with Zipfian key distribution to show how skewed key accesses in real-world workloads would perform under different indexings.

Table 2: SSD Parameters

# of Blocks	16384	Page Size	4KB
# of Pages per Block	256	SSD Cache Size	16MB
Page Read	60us	Page Write	800us
Block Erase	3ms	Cacheline Read	4.8us
Cacheline Write	0.6us	Over-provision	20%

4.2 Write Performance Analysis

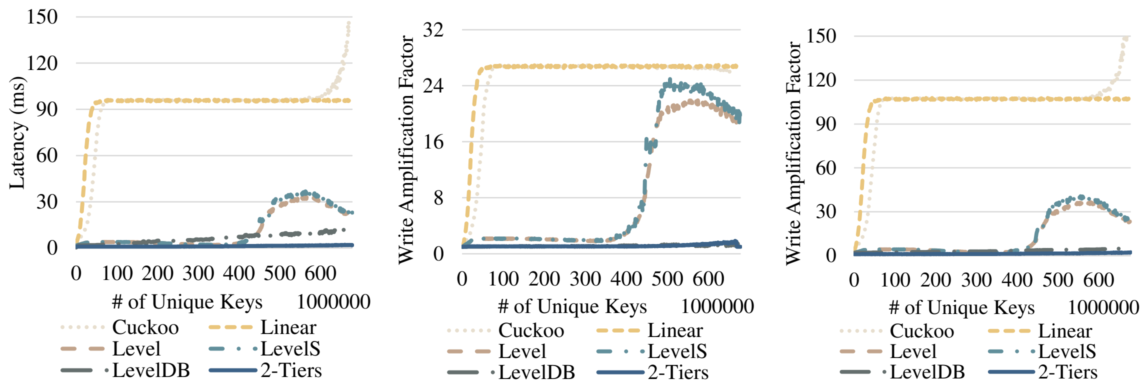
In this experiment, we stress the SSD to evaluate the write performance of different hash indexings. We collect the latency of hash tables and SSD GC efficiency under different schemes during insertions and updations. The amount of data inserted is 20GB, which is enough to trigger GC. Given that there is no extra design for resizing scheme of Tiered Hashing, we configure the capacity of each hash data structure the same to accommodate total data inserted just enough (95%), thus avoiding resizing and preventing its complex interaction with internal GC. Since the performance changes with the fullness of indexings (i.e., the number of inserted KVs), we collect the statistics per 100k insertions. The results are shown in Figure 8.

4.2.1 Overall Write Performance. Figure 8a shows the insert latencies of different indexings. We can find that the latencies of Cuckoo and Linear Hashing quickly reach a high level of 94ms. In the end, the latency of Cuckoo Hashing climbs to about 250ms because it works more aggressively to move key-value pairs with different hash functions as the number of hash collisions rises. Linear Hashing's latency reaches 93ms because of its strategy of linear probing. Level Hashing does not show a noticeable difference no matter it applies multi-stream technology or not. Level Hashing's insert latency is 3.1ms on average at the beginning, then raises to 22.9ms on average. Tiered Hashing's insert latencies remain at the lowest level (about 1.3ms) because its opportunistic data movement strategy generates skewed writes, which benefit from Multi-stream SSDs. We can see a slight improvement when comparing Tiered Hashing to LevelDB, with LevelDB showing 5.14 times higher latency than that of Tiered Hashing. Such a consequence is mainly due to the costly compaction of LevelDB. As the number of inserted keys increases, both the latency and fluctuation of latency for LevelDB expand as well, while the latency of Tiered Hashing stays low constantly.

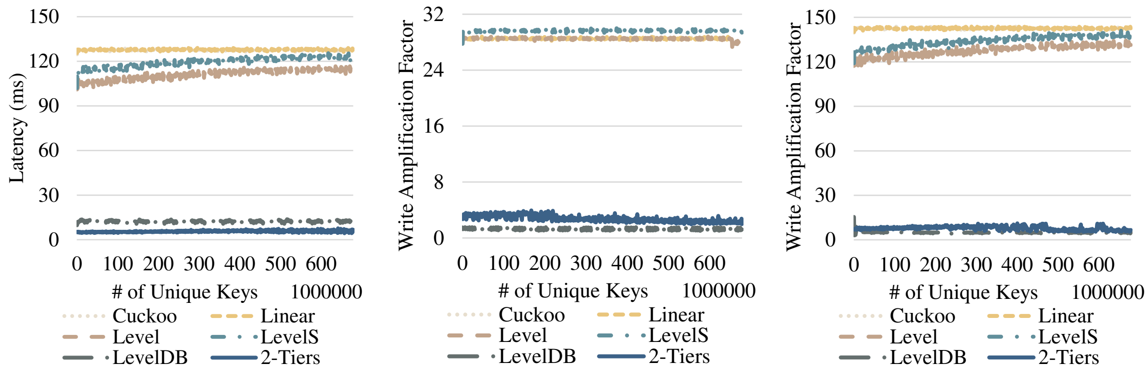
Figure 8d shows the update latencies of different hash indexings, which is relatively steady compared to the insert latencies. Both Cuckoo Hashing and Linear Hashing show a latency of 127ms on average. Level Hashing without multi-stream needs 114ms to serve update operations, and Level Hashing with multi-stream needs 119ms. Tiered Hashing's average update latency is the lowest (5.6ms) due to its opportunistic data movement and lazy updating strategies. We can also find the update latency of Tiered Hashing is slightly lower than that of LevelDB, with LevelDB showing 2.23 times higher latency than that of Tiered Hashing.

4.2.2 Internal SSD Write Amplification. To understand why Tiered Hashing outperforms others in insertion and updating workloads, we collect the internal write amplification factor (WAF) for different indexing designs. We denote the ratio of total flash writes times devided by the number of writes received by SSD as the internal WAF, which mainly comes from SSD GC overhead. In other words, the internal WAF reflects how SSD-friendly the write traffic generated by different indexings is.

Figure 8b shows the internal WAF for insertion workloads. Cuckoo Hashing and Linear Hashing's WAFs quickly rise to 26.7, proving that the SSD performance decreases significantly under random hash table writes. Since Level Hashing employs a two-layer structure, Level Hashing's WAF initially stays at 2.05 for a period; then, it rises to 18.6 because the SSD begins to perform active garbage collection heavily when it is about full. Surprisingly, with the aid of the multi-stream feature, LevelS shows slightly higher internal WAF than Level when more keys are inserted. Such a result shows that the combination of the multi-stream feature and indexings without enough skewness is negatively optimized. Tiered Hashing's WAF remains at about 1.14 throughout the experiment, which is similar to the 1.12 of LevelDB. Figure 8e shows the internal WAF during updating. Tiered Hashing's WAF is 2.63 on average, which is slightly larger than the 1.25 of LevelDB, while other hash indexings' internal WAFs are about 28.7 on average. The above statistics



(a) Latency for Insertion. (b) Internal WAF for Insertion. (c) External WAF for Insertion.



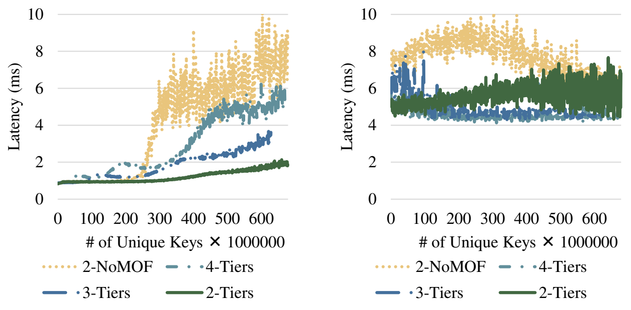
(d) Latency for Updating. (e) Internal WAF for Updating. (f) External WAF for Updating.

Figure 8: Write Performance Analysis

prove that Tiered Hashing achieves GC efficiency similar to tree structures. As shown in Figure 2e, Tiered Hashing with ODM does the best in reducing WAF and improving GC efficiency because it generates a highly skewed write distribution.

4.2.3 External SSD Write Amplification. To study the effect of combining multi-stream with our Tiered Hashing, we also define external WAF as the ratio of total write times to the flash to the number of user writes. Such an external WAF reflects the write amplification of indexing schemes and the SSD GC. The pattern of external WAF is similar to internal WAF except for the numerical gap. The Cuckoo and Linear hashing generate significant write amplification than the others. We find that Tiered Hashing's performance is slightly better than LevelDB's regarding external WAF, which may be due to the costly compaction of LevelDB. In short, both internal and external WAF prove that Tired Hashing has tree-like GC efficiency.

4.2.4 Sensitivity Analysis of Multiple Layers. To demonstrate the effect of multi-tiers of Tiered Hashing, we also evaluate Tiered Hashing with two, three, and four tiers of hash tables. Figure 9 show the results of different experiments. During insertions, 3-Tiers and 4-Tiers Tiered Hashing exhibit about 43.04% and 129.64% higher latencies than 2-Tiers. During updating, they exhibit about



(a) Latency for Insertion. (b) Latency for Updating.

Figure 9: Impact of Settings

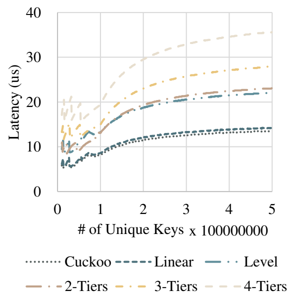
12.55% and 18.34% lower latencies. We show the impact on search performance in the following subsection.

4.3 Search Performance

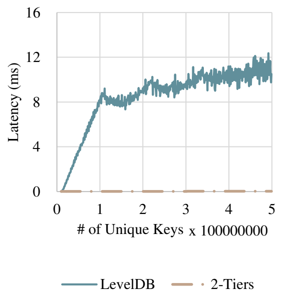
In this experiment, we evaluate the search performance of different indexings. We start from an empty table and insert a different

Table 3: Comparison of Average External Write Amplification

Avg EXWAF	Cuckoo	Linear	Level	LevelS	2-Tiers	2-NoMOF	LevelDB
Insert	105.05	104.14	13.21	14.14	1.68	4.58	3.31
Update	142.58	142.58	127.87	134.22	7.31	8.54	4.87



(a) Different Hashing.



(b) Comparing to LevelDB.

Figure 10: Search Performance Analysis.

Table 4: Effectiveness of Shadow Reading

	With SR(us)	Without SR(us)
avg	188.5	208.4
max	361.8	361.8

Table 5: Comparison of Read Amplification

	Cuckoo	Linear	Level	2-Tiers	LevelDB
RAF	1.86	1.86	2.05	2.28	141.4

number of unique keys. Afterward, we perform random search operations and collect the average search latency. As shown in Figure 10a, Cuckoo and Linear Hashing respond much faster than other schemes because they both use a one-level table structure. As both Tiered Hashing and Level Hashing employ hierarchical structures, we limit the number of layers in Tiered Hashing to two, three, and four. When the number of layers is two, Tiered Hashing's search latencies are identical to Level Hashing. Also, Tiered Hashing's search performance is almost proportional to the total number of layers. 3-Tiers performs 20.79% worse than 2-Tiers, and 4-Tiers performs 55.00% worse than 2-Tiers. As shown in Figure 10b, the search latency of LevelDB is much higher than all hash indexings' due to the deep path to find a key. Specifically, as the number of inserted keys increases, the search latency increases dramatically to more than 458 times of 2-Tiers.

4.4 Effectivenss of Stand-alone Design

4.4.1 Effectiveness of Maximum One Flush Policy. To demonstrate the effect of the Maximum One Flush policy (MOF) alone, we configure two-tiered Tiered Hashing without corresponding design, denoted as 2-NoMOF. 2-NoMOF employs ODM without MOF, thus incurring numerous extra flushes to SSD. Figure 9a compare the latencies of different configurations during insertion. Results show that with the increase of inserted keys, all four configurations exhibit higher latencies, while the 2-NoMOF performs worse than others (222.00% worse than 2-Tiers). As shown in Figure 9b, the 2-NoMOF performs the worst during updating procedure as well (37.87% worse than 2-Tiers). However, the performance of 2-NoMOF gradually reaches 2-Tiers when more keys are updated since in-place updating dominates, which prevents ODM from incurring many more data movements. In addition, we also compare the external WAF of Tiered Hashing with and without MOF. As shown in Table 3, the WAF of 2-NoMOF is 2.73 times of 2-Tiers' during insertion. To conclude, with MOF policy, write traffic to SSD is effectively reduced.

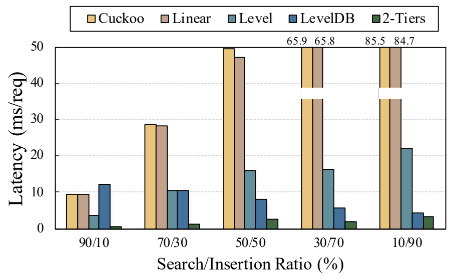
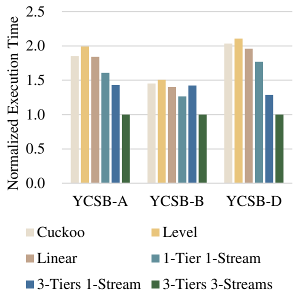


Figure 11: The Overall YCSB Performance Analysis.

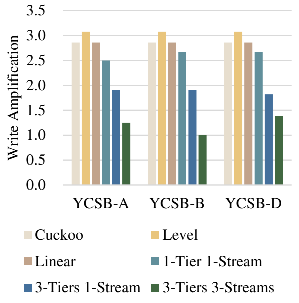
4.4.2 Effectiveness of Shadow Reading. To quantify the effectiveness of shadow reading (SR) alone, we configure Tiered Hashing with and without shadow reading. As shown in Table 4, shadow reading improves the average search latency by 10.6%. Since the depth of layer dominates the tail latency of searching, the maximum search latency shows no difference.

4.5 YCSB Performance Analysis

In this experiment, we use YCSB [15], a macrobenchmark for key-value stores, to analyze the performance of different hash indexings and LevelDB. We use the default configuration of Zipfian distribution in the workloads and vary the ratios of search/insertion from 90/10 to 10/90. The amount of data for all YCSB benchmarks is 20GB. The overall YCSB performance is shown in Figure 11. We observe that Cuckoo and Linear Hashing have 19.16 - 27.16 times higher latencies than Tiered Hashing in all workloads. Level Hashing has 7.02 - 7.10 times higher latency than Tiered Hashing. LevelDB has 0.37 - 24.03 times higher latency. This is because Cuckoo Hashing has many eviction operations; Linear Hashing has a long path to find a free slot to insert. Both of them and Level Hashing can't utilize the efficiency of multi-stream technology. Though the four hash indexings don't show much difference in search performances, their insert performances differ significantly.



(a) Latency for YCSB.



(b) Internal WAF for YCSB.

Figure 12: Multi Stream Analysis under YCSB.

In the case of LevelDB, its write performance can rival that of Tiered Hashing because LevelDB has in-memory memtables which will be persisted in coarse granularity (such as 2 MB). This characteristic of LSM-based LevelDB makes its write performance better than the hash-based structure. However, LevelDB's read performance can not compete with Tiered Hashing. As shown in Table 5, the read amplification (RAF, similar to external WAF) of LevelDB is significantly larger than all hashings, which is more than 62 times than the RAF of 2-tiers. To conclude, Tiered Hashing shows the best performance across all workloads.

4.5.1 Effectiveness of Collaboration with Multi-stream. This experiment studies the extent of improvement brought by combining Tiered Hashing with the multi-stream feature. Figure 12a and Figure 12b compare the latency and internal WAF of Tiered Hshing with/without the aid of the multi-stream feature under different YCSB benchmarks. Compared with 3-Tier 1-Stream, 3-Tier 3-Stream decreases the average WAF by 34.38%, 47.49% and 24.14% in YCSB-A, YCSB-B and YCSB-D. 3-Tier 3-Stream also reduces the average latencies by 30.10%, 29.73% and 22.28% in YCSB-A, YCSB-B and YCSB-D. By comparing 3-Tier 1-Stream with 3-Tier 3-Stream, we can see a noticeable drop in WAF and latency if muli-stream is employed. The collaboration with multi-stream does provide improvements. Since YCSB has hotspot in key distribution, the collaboration of Tierd Hashing and multi-stream shows obvious improvement.

5 CONCLUSION

This paper presents the Tiered Hashing, a pyramidal hierarchical structured hash indexing that employs a *opportunistic data movement* scheme to generate multi-stream SSD-friendly skewed write workload to improve the garbage collection efficiency of SSDs. We then develop *maximum one flush policy* and *shadow reading* to mitigate the overhead of data movement. Our experiments show that Tiered Hashing delivers comparable search performance against other hash indexings, such as Level Hashing, and write efficiency similar to LSM-tree indexings, such as LevelDB.

ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their valuable feedback. This work was supported in part by the National Natural Science Foundation of China under Grant No.62102155, No.61821003, No.61872413, No.U2001203, No.61902137, and in part by Alibaba Group through Alibaba Innovative Research Program No.2022010344.

REFERENCES

[1] [n.d.]. Lightning Memory-mapped Database. https://symas.com/lmdb/.
[2] [n.d.]. Memcached. https://developer.nvidia.com/blog/gpudirect-storage/.
[3] [n.d.]. MongoDB: Memory Mapped File Usage. https://docs.mongodb.com/manual/faq/storage/.
[4] [n.d.]. Redis. https://redis.io/.
[5] Ahmed Abulila, Vikram Sharma Mailthody, Zaid Qureshi, Jian Huang, Nam Sung Kim, Jinjun Xiong, and Wen-Mei Hwu. 2019. FlatFlash: Exploiting the Byte-Accessibility of SSDs within A Unified Memory-Storage Hierarchy. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 971–985.
[6] Hiroyuki Akinaga and Hisashi Shima. 2010. Resistive random access memory (ReRAM) based on metal oxides. *Proc. IEEE* 98, 12 (2010), 2237–2251.
[7] Dmytro Apalkov, Alexey Khvalkovskiy, Steven Watts, Vladimir Nikitin, Xueti Tang, Daniel Lottis, Kiseok Moon, Xiao Luo, Eugene Chen, Adrian Ong, et al. 2013. Spin-transfer torque magnetic random access memory (STT-MRAM). *ACM Journal on Emerging Technologies in Computing Systems (JETC)* 9, 2 (2013), 1–35.
[8] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. 2012. Workload analysis of a large-scale key-value store. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE joint international conference on Measurement and Modeling of Computer Systems*. 53–64.
[9] Duck-Ho Bae, Insoon Jo, Youra Adel Choi, Joo-Young Hwang, Sangyeun Cho, Dong-Gi Lee, and Jaeheon Jeong. 2018. 2B-SSD: the case for dual, byte-and block-addressable solid-state drives. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 425–438.
[10] Matias Bjørling, Abutalib Aghayev, Hans Holmberg, Aravind Ramesh, Damien Le Moal, Gregory R Ganger, and George Amvrosiadis. 2021. {ZNS}: Avoiding the Block Interface Tax for Flash-based {SSDs}. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. 689–703.
[11] Renhai Chen, Zili Shao, and Tao Li. 2016. Bridging the I/O performance gap for big data workloads: A new NVDIMM-based approach. In *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Press, 9.
[12] Zhangyu Chen, Yu Huang, Bo Ding, and Pengfei Zuo. 2020. Lock-free Concurrent Level Hashing for Persistent Memory. In *2020 {USENIX} Annual Technical Conference ({USENIX} {ATC} 20)*. 799–812.
[13] Nachshon Cohen, David T. Aksun, Hillel Avni, and James R. Larus. 2019. Fine-Grain Checkpointing with In-Cache-Line Logging. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems* (Providence, RI, USA) *(ASPLOS '19)*. Association for Computing Machinery, New York, NY, USA, 441–454. https://doi.org/10.1145/3297858.3304046
[14] Ronald Fagin, Jurg Nievergelt, Nicholas Pippenger, and H Raymond Strong. 1979. Extendible hashing—a fast access method for dynamic files. *ACM Transactions on Database Systems (TODS)* 4, 3 (1979), 315–344.
[15] Steffen Friedrich and Norbert Ritter. 2018. *YCSB*. Springer International Publishing, Cham, 1–4. https://doi.org/10.1007/978-3-319-63962-8_131-1
[16] Donghyun Gouk, Sangwon Lee, Miryeong Kwon, and Myoungsoo Jung. 2022. Direct Access,{High-Performance} Memory Disaggregation with {DirectCXL}. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. 287–294.
[17] Siddharth Gupta, EcoCloud, Lei Yan, Mark Sutherland, Abhishek Bhattacharjee, and Peter Yan-Tek Hsu. 2020. AstriFlash: An Online Flash-Based Memory Hierarchy.
[18] Kyuhwa Han, Hyunho Gwak, Dongkun Shin, and Jooyoung Hwang. 2021. ZNS+: Advanced zoned namespace interface for supporting in-storage zone compaction. In *15th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 21)*. 147–162.
[19] Jian Huang, Anirudh Badam, Moinuddin K Qureshi, and Karsten Schwan. 2015. Unified address translation for memory-mapped SSDs with FlashMap. In *ACM SIGARCH Computer Architecture News*, Vol. 43. ACM, 580–591.
[20] Myoungsoo Jung. 2022. Hello bytes, bye blocks: PCIe storage meets compute express link for memory expansion (CXL-SSD). In *Proceedings of the 14th ACM Workshop on Hot Topics in Storage and File Systems*. 45–51.
[21] Taejin Kim, Duwon Hong, Sangwook Shane Hahn, Myoungjun Chun, Sungjin Lee, Jooyoung Hwang, Jongyoul Lee, and Jihong Kim. 2019. Fully automatic stream management for multi-streamed SSDs using program contexts. In *17th {USENIX} Conference on File and Storage Technologies ({FAST} 19)*. 295–308.
[22] Apostolos Kokolis, Dimitrios Skarlatos, and Josep Torrellas. 2019. PageSeer: Using page walks to trigger page swaps in hybrid memory systems. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 596–608.
[23] Changman Lee, Dongho Sim, Jooyoung Hwang, and Sangyeun Cho. 2015. F2FS: A new file system for flash storage. In *13th {USENIX} Conference on File and*

Storage Technologies ({FAST} 15). 273–286.

[24] Gyusun Lee, Wenjing Jin, Wonsuk Song, Jeonghun Gong, Jonghyun Bae, Tae Jun Ham, Jae W. Lee, and Jinkyu Jeong. 2020. A Case for Hardware-Based Demand Paging. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. 1103–1116. https://doi.org/10.1109/ISCA45697.2020.00093

[25] Witold Litwin. 1980. Linear Hashing: a new tool for file and table addressing.. In *VLDB*, Vol. 80. 1–3.

[26] Virendra J Marathe, Margo Seltzer, Steve Byan, and Tim Harris. 2017. Persistent memcached: Bringing legacy code to byte-addressable persistent memory. In *9th {USENIX} Workshop on Hot Topics in Storage and File Systems (HotStorage 17)*.

[27] Moohyeon Nam, Hokeun Cha, Young-ri Choi, Sam H Noh, and Beomseok Nam. 2019. Write-optimized dynamic hashing for persistent memory. In *17th {USENIX} Conference on File and Storage Technologies ({FAST} 19)*. 31–44.

[28] Intel Newsroom. 2015. Introducing Intel Optane technology–bringing 3D XPoint memory to storage and memory products.

[29] Rasmus Pagh and Flemming Friche Rodler. 2004. Cuckoo hashing. *Journal of Algorithms* 51, 2 (2004), 122–144.

[30] Anastasios Papagiannis, Giorgos Xanthakis, Giorgos Saloustros, Manolis Marazakis, and Angelos Bilas. 2020. Optimizing Memory-mapped I/O for Fast Storage Devices. In *2020 {USENIX} Annual Technical Conference ({USENIX}{ATC} 20)*. 813–827.

[31] Eunhee Rho, Kanchan Joshi, Seung-Uk Shin, Nitesh Jagadeesh Shetty, Jooyoung Hwang, Sangyeun Cho, Daniel DG Lee, and Jaeheon Jeong. 2018. FStream: managing flash streams in the file system. In *16th {USENIX} Conference on File and Storage Technologies ({FAST} 18)*. 257–264.

[32] Hongchan Roh and Sanghyun Park. 2008. An efficient hash index structure for solid state disks. In *2008 International Conference on Information and Knowledge Engineering, IKE 2008*. 256–261.

[33] H-S Philip Wong, Simone Raoux, SangBum Kim, Jiale Liang, John P Reifenberg, Bipin Rajendran, Mehdi Asheghi, and Kenneth E Goodson. 2010. Phase change memory. *Proc. IEEE* 98, 12 (2010), 2201–2227.

[34] Shiqin Yan, Huaicheng Li, Mingzhe Hao, Michael Hao Tong, Swaminathan Sundararaman, Andrew A. Chien, and Haryadi S. Gunawi. 2017. Tiny-Tail Flash: Near-Perfect Elimination of Garbage Collection Tail Latencies in NAND SSDs. In *15th USENIX Conference on File and Storage Technologies (FAST 17)*. USENIX Association, Santa Clara, CA, 15–28. https://www.usenix.org/conference/fast17/technical-sessions/presentation/yan

[35] Chengcheng Yang, Peiquan Jin, Lihua Yue, and Dezhi Zhang. 2016. Self-Adaptive Linear Hashing for solid state drives. In *2016 IEEE 32nd International Conference on Data Engineering (ICDE)*. IEEE, 433–444.

[36] Fei Yang, Kun Dou, Siyu Chen, Mengwei Hou, Jeong-Uk Kang, and Sangyeun Cho. 2015. Optimizing nosql db on flash: A case study of rocksdb. In *2015 IEEE 12th Intl Conf on Ubiquitous Intelligence and Computing and 2015 IEEE 12th Intl Conf on Autonomic and Trusted Computing and 2015 IEEE 15th Intl Conf on Scalable Computing and Communications and Its Associated Workshops (UIC-ATC-ScalCom)*. IEEE, 1062–1069.

[37] Jingpei Yang, Rajinikanth Pandurangan, Changho Choi, and Vijay Balakrishnan. 2017. AutoStream: automatic stream management for multi-streamed SSDs. In *Proceedings of the 10th ACM International Systems and Storage Conference*. ACM, 3.

[38] Hwanjin Yong, Kisik Jeong, Joonwon Lee, and Jin-Soo Kim. 2018. vStream: virtual stream management for multi-streamed SSDs. In *10th {USENIX} Workshop on Hot Topics in Storage and File Systems (HotStorage 18)*.

[39] Jie Zhang, Miryeong Kwon, Donghyun Gouk, Sungjoon Koh, Nam Sung Kim, Mahmut Taylan Kandemir, and Myoungsoo Jung. 2021. *Revamping Storage Class Memory with Hardware Automated Memory-over-Storage Solution*. IEEE Press, 762–775. https://doi.org/10.1109/ISCA52012.2021.00065

[40] Pengfei Zuo and Yu Hua. 2017. A write-friendly hashing scheme for non-volatile memory systems. In *Proc. MSST*.

[41] Pengfei Zuo, Yu Hua, and Jie Wu. 2018. Write-optimized and high-performance hashing index scheme for persistent memory. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*. 461–476.