Persistent Spiral Storage

Wenyu Peng
Engineering Science Joint Doctoral Program
San Diego State University
San Diego, USA
wpeng@sdsu.edu

Tao Xie
Computer Science
San Diego State University
San Diego, USA
txie@sdsu.edu

Paul H. Siegel
Electrical and Computer Engineering
University of California, San Diego
La Jolla, USA
psiegel@ucsd.edu

2024 IEEE 42nd International Conference on Computer Design (ICCD) | 979-8-3503-8040-8/24/$31.00 ©2024 IEEE | DOI: 10.1109/ICCD63220.2024.00046

***Abstract*—The advent of byte-addressable persistent memory (PM) has led to a resurgence of interest in adapting existing dynamic hashing schemes to PM. Compared with its two well-known peers (extendible hashing and linear hashing), spiral storage has received little attention due to its limitations. After an in-depth analysis, however, we discover that it has a good potential for PM. To show its strength, we develop a persistent spiral storage called PASS (Persistence-Aware Spiral Storage), which is facilitated by a group of new/existing techniques. Further, we conduct a comprehensive evaluation of PASS on a server equipped with Intel Optane DC Persistent Memory Modules (DCPMM). Experimental results demonstrate that compared with two state-of-the-art schemes it exhibits better performance.**

***Index Terms*—dynamic hashing, persistent memory, spiral storage, key-value store, indexing data structure**

I. Introduction

In the past decade, various persistent memory (PM) technologies such as 3D-XPoint [1] have been proposed. They normally offer near-DRAM performance plus byte-addressability as well as disk-like persistence and capacity. Researchers hope that these PM devices can either eventually replace the long-lasting volatile DRAM so that the dream of a large-scale persistent main memory comes true, or at least they can be inserted into an appropriate place in the current memory hierarchy to further mitigate the performance gap between DRAM and storage. After Intel started shipping Optane DCPMM in 2018, part of this dream has been realized, as a server with a DRAM-PM hybrid memory system where PM is connected directly to a CPU now becomes a commercial product. To reap the full benefits of Optane DCPMM, a question naturally arises: How to adapt a hash table that was originally designed for disk/DRAM to PM?

Hashing schemes can be categorized into two camps: static hashing [2] and dynamic hashing [3]–[5]. Unlike static hashing, a dynamic scheme can adjust the size of its hash table on demand without the need to rehash the entire table. This expansion-on-demand feature makes it an attractive indexing data structure. For example, extendible hashing [3] and linear hashing [4], two popular dynamic hashing schemes, have been widely used in file systems (e.g., IBM GPFS [6]) and databases (e.g., SQL Server Hekaton [7]) on disk/DRAM-based systems. They became natural candidates for the recent development of PM-oriented dynamic hashing schemes such as CCEH [8] and Dash [9]. While CCEH [8] focused on extendible hashing only, Dash [9] targeted both extendible hashing and linear hashing. The two adapted dynamic hashing schemes perform well on PM [8], [9].

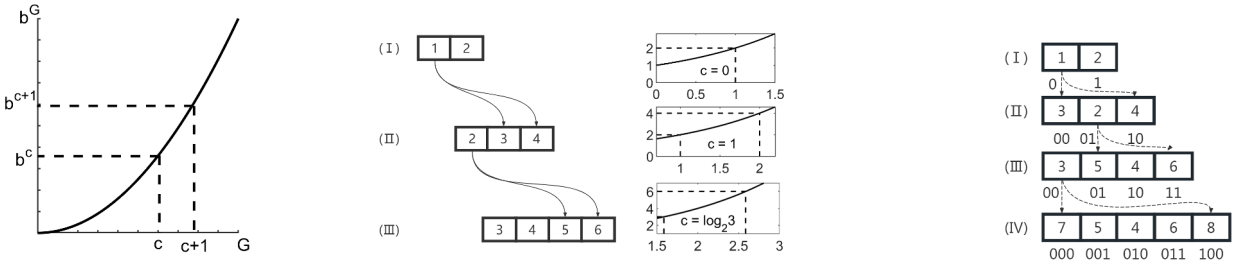
Unlike its two popular peers, spiral storage [5], [10] has received little attention as it has some inherent limitations. Two address mapping operations are needed to search a key k. The first one leads to the calculation of a power function (see Equation 1), whereas the second one requires the computing of a recursive function (see Algorithm 1). Both are time-consuming. Besides, spiral storage employs a directoryless mode. Thus, it demands periodically reallocating a contiguous memory region to perform an expansion, which is very expensive on PM. However, after an in-depth study, we found that spiral storage also possesses a desirable feature. It performs each hash table expansion through address remapping, and thus, leads to fewer rehashing operations compared with its two peers. As a result, multiple PM reads and writes can be avoided. We speculate that spiral storage has a potential for PM.

To confirm our speculation, we propose several techniques to upgrade the original spiral storage to a persistent spiral storage called PASS (Persistence-Aware Spiral Storage). First, we design a new structure for PASS on PM (see Section III-B). Second, we propose an approximation method to accelerate the calculation of the power function (see Section III-C) and a look-up table to replace the recursive logical-physical address mapping function (see Section III-D). The two strategies combined turn the two-step address mapping into one quick mapping table access, which solves an inherent limitation of spiral storage (see Section III-E). PASS also adopts a group of existing optimization techniques including stashing, fingerprinting [11], optimistic concurrency control [9], and bucket load balancing [9].

We conducted a comprehensive experimental study on a 24-core server equipped with Intel DCPMM. Our experimental results show that PASS outperforms the state-of-the-art PM-oriented hashing schemes. Overall, we make the following contributions. First, we investigate spiral storage on a modern computing platform and discover its potential for PM. To the best of our knowledge, this is the first research in adapting spiral storage to PM. Second, we propose several techniques so that PASS can deliver a similar or even better performance than the state-of-the-art approaches. Finally, we provide a comprehensive empirical evaluation of PASS.

2576-6996/24/$31.00 ©2024 IEEE
DOI 10.1109/ICCD63220.2024.00046

Authorized licensed use limited to: Univ of Calif San Diego. Downloaded on March 12,2025 at 18:22:45 UTC from IEEE Xplore. Restrictions apply.



(a) Graph of b^G versus G (b) An example of table expansion (c) Logical-physical address mapping

Fig. 1: Conventional spiral storage.

II. Spiral Storage

A. Conventional Spiral Storage

Conventional spiral storage is a directoryless mode dynamic hashing [5], [10]. To search a key k, it requires two address mapping operations: $k \rightarrow hash(k) \rightarrow$ logical address, and then, logical address $\rightarrow$ physical address. In its original design, buckets are deleted at the left side of the table and added at the right side. A simple optimization is to reuse the space deleted, which requires a logical to physical address mapping operation [5]. A hash function that returns a hash value between $[0, 1]$ is needed for the first address mapping operation. A logical address can be calculated by Equation 1

$$logical = \lfloor b^G \rfloor . \quad (1)$$

While b is called growth factor, G is computed by Equation 2

$$G = \lceil c - hash(k) \rceil + hash(k). \quad (2)$$

An increase of c causes an expansion [5]. The value of G ranges from c to c + 1. To expand the logical address space, a new value of c is chosen to eliminate the current first logical bucket. The new value c' is obtained by Equation 3

$$c' = log_b(first + 1), \quad (3)$$

where *first* is the logical address of the current leftmost bucket, which can be computed as *first* = $\lfloor b^c \rfloor$. The relationship among b, c, and G is shown in Figure 1a. As the interval $(c, c+1)$ is moved to the right along the x axis, the size of the interval (b^c, b^{c+1}) on the y axis will increase accordingly. Since the number of newly added buckets is more than the number of buckets deleted, an expansion is achieved. Records from the deleted buckets are then spread across the newly added buckets.

The example shown in Figure 1b explains how spiral storage expands its logical address space. Initially, the growth factor b is set to 2, and c is configured to 0. Thus, the range of logical address space is [1, 2] based on Equation 1 and Equation 2. Now, there are only two logical buckets as shown in the left side of Figure 1b(I). The mapping on the right side of Figure 1b(I) shows how all records within the interval (0, 1) on the x axis map into these two logical buckets. When the logical address space expands, a new c is calculated by Equation 3 where *first* is 1 and b is 2. Thus, the new c is $log_2(1 + 1)$ = 1. Now the range of logical address space becomes [2, 4]. While the logical bucket 1 is deleted, two new logical buckets are added as shown in Figure 1b(II). A further expansion is shown in Figure 1b(III). The logical-physical mapping algorithm is shown in Algorithm 1.

Figure 1c shows an example of expansion from size 2 to size 5 with the mapping of a logical bucket to a physical bucket without deleting any bucket. The numbers within the buckets are the logical addresses of the buckets, whereas the numbers below the buckets are the physical addresses of the buckets. In the first expansion, when logical bucket 1, which resides in physical bucket 0, is mapped into logical buckets 3 and 4, the physical bucket 0 is reused for the logical bucket 3 as shown in Figure 1c(II). Logical bucket 4 (physical bucket 2) is then added to the end of the table.

The procedure of finding the physical address of a logical bucket is as follows: If the logical bucket resides in the first instantiation of the physical bucket (i.e., the logical bucket stays in a newly created physical bucket), its physical address is the distance between the last logical bucket and the current first logical bucket (Line 10 of Algorithm 1). For example, the physical address for the logical bucket 8 is 4 (i.e., 8 - 4 = 4) as shown in Figure 1c(IV). To find the physical address of a reused bucket such as where logical bucket 7 resides, spiral storage has to backtrack the logical instantiations to find when the physical bucket was instantiated (Line 7 of Algorithm 1). Logical bucket 7 is in physical bucket 0 in Figure 1c(IV).

Algorithm 1 Logical-Physical Address Mapping

```
1:  function PhysicalAddress(logical_address):
2:      high = ⌊(1 + logical_address)/b⌋
3:      low = ⌊logical_address/b⌋
4:      if low < high then
5:          physical_address = PhysicalAddress(low)
6:      else
7:          physical_address = logical_address - low
8:      end if
9:      return physical_address
10: end function
```


Authorized licensed use limited to: Univ of Calif San Diego. Downloaded on March 12,2025 at 18:22:45 UTC from IEEE Xplore. Restrictions apply.

III. The Design of PASS

A. Design Overview

Motivated by the insights stated in the introduction, we develop some optimization techniques for spiral storage. First, an approximation method (see Section III-C) is proposed to avoid the calculations of a power function with a floating-point exponent (see Equation 1). Next, we found that the mapping between a logical address and its associated physical address is fixed, which is a property of spiral storage. Thus, a predetermined look-up table is proposed to replace the time-consuming recursive function shown in Algorithm 1. The comparison of address mapping is shown in Table I.

B. Structure of PASS

Similar to CCEH [8] and Dash [9], PASS employs a three-level structure (i.e., directory $\rightarrow$ segment $\rightarrow$ bucket). The directory contains some metadata items and a group of directory entries. Each entry points to an array of segments, which contains one or multiple segments. PASS splits at the segment level. Like Dash-LH [9], PASS expands a hash table by several fixed-size segments before triggering a double expansion. For example, each of the first 4 entries points to a one-segment array. Each of the next 4 entries points to a two-segment array, and so on. Each segment is composed of a fixed number of normal buckets and stash buckets (for overflow records). If the number of overflow records is more than the capacity of the stash buckets, a newly allocated stash bucket chain will be added to the segment. PASS borrows the layout of a regular bucket of Dash-EH [9]. The size of a bucket is set to 256 bytes to match the block size of Optane DCPMM [1].

TABLE I: Performance comparisons of address mapping

Address Mapping Operation	Server
Murmur hash function	163 MOPS
Spiral Storage (SS)	68 MOPS
SS with power table only	56,244 MOPS
SS with look-up table only	1,823 MOPS
PASS	90,332 MOPS

C. An Approximation Method

Since the logical address of a key k is the largest integer that is smaller than b^G, we do not have to calculate the exact value of b^G as long as we can quickly discover this integer. For a particular key k, we can obtain the value of its corresponding G based on Equation 2 after c is initialized. Based on Equation 1, we know that $logical \leq b^G < logical + 1$, which leads to $log_b(logical) \leq G < log_b(logical + 1)$ (called Inequality 1). The growth factor b is a fixed value once the design of a hash table is completed. In our implementation, b is set to 2. Therefore, we can calculate log_2 for a group of consecutive integers starting from 1 in advance.

However, scanning such a table to find out the integer $logical$ that satisfies Inequality 1 is a slow process. Thus, we augment G to a large integer represented by $\lfloor scalar \times G \rfloor$ where $scalar$ is a large integer constant. Based on Inequality 1, we have $\lfloor scalar \times log_b(logical) \rfloor \leq \lfloor scalar \times G \rfloor < \lfloor scalar \times log_b(logical + 1) \rfloor$. Thus, we can use $\lfloor scalar \times G \rfloor$ as the index to directly obtain the corresponding value of $logical$, which can be calculated beforehand, and then, stored in a table called *power table*. The algorithm of power table construction is shown in Algorithm 2 where c_max is the value of c when the size of the table reaches its maximum. PASS constructs the power table offline and stores it in PM. When the system crashes, the power table will still be valid, and thus does not need to be recalculated. Now, we use an example to illustrate how the method works. Suppose that $G = 6.21$, $scalar = 256$, $c\_max = 16$, we can quickly obtain the value of the logical address (i.e., 74) from Table II because $table[\lfloor scalar \times G \rfloor] = table[1589] = 74$ (see Table II). Equation 1 also gives us $\lfloor 2^{6.21} \rfloor = 74$.

TABLE II: A segment of a sample power table

1,583	1,584	...	1,588	1,589	...	1,593	1,594
72	73	...	73	74	...	74	75

Algorithm 2 Power-table construction

```
1:  function Power_table_contruction(scalar, c_max):
2:      /* prepare an array in PM */
3:      power = pm_alloc(scalar × c_max × 8 Bytes)
4:      logical = 1
5:      for i = 0; i < scalar × c_max; i++ do
6:          temp = log2(init) × scalar
7:          if i >= ⌊temp⌋ then
8:              logical++
9:          end if
10:         power[i] = logical − 1
11:     end for
12: end function
```

D. Look-up Table

Since the mapping between a logical address and its associated physical address is fixed, we propose a predetermined *look-up table* to replace the time-consuming recursive function shown in Algorithm 1. The index of the table is the logical address of a key and the value stored in the location indexed is the corresponding physical address. Table III is a sample look-up table. If the logical address is 8, then the physical address is 4 which can be easily found in the table. Like the power table, PASS builds the look-up table offline and it is stored in PM. PASS calculates the physical address of each logical address beforehand and the size of the look-up table can be determined in advance. In our experiments, a hash table that holds 200 million records requires a 16 MB look-up table. Experimental results exhibit that after the look-up table is used the address mapping performance of spiral storage can be improved by 26.94× on the server shown in the 4th row of Table I.

TABLE III: A segment of a sample look-up table

logical address	1	2	3	4	5	6	7	8
physical address	0	1	0	2	1	3	0	4

Authorized licensed use limited to: Univ of Calif San Diego. Downloaded on March 12,2025 at 18:22:45 UTC from IEEE Xplore. Restrictions apply.

E. Mapping table

For a key k, the power table stores the logical address corresponding to $hash(k)$, whereas the look-up table holds its associated physical address. Apparently, the two tables can be combined into one so that PASS can directly obtain the physical address of k by accessing the combined table once rather than accessing the two tables sequentially, which further shrinks its address mapping time. The combined table is called *mapping table*. Like its two peers, PASS now enjoys a simple one-step address mapping for a basic operation like a search. When a mapping table is employed, the address mapping performance of PASS achieves a remarkable speedup of 554.18× compared to the Murmur hash function (see the last row of Table I). Although PASS only needs the mapping table for a basic operation, a power table and a look-up table are still indispensable as PASS needs the logical address of a key to determine its rehashing process during an expansion. Since the size of the mapping table is the same as that of the power table (i.e., 83 MB), the total space overhead of the three tables is 182 MB. Meanwhile, to accommodate 200 million records, PASS needs to allocate a 16GB memory pool, which leads to a 1.13% PM utilization for the three tables combined. All three tables are static as none of them changes over expansions.

IV. Performance Evaluation

A. Experimental Setup

All experiments are conducted on a server with an Intel Xeon Gold 6252 CPU clocked at 2.1 GHz. It is equipped with 192 GB of DRAM (6×32 GB DIMMs) plus 768GB of Optane DCPMM (6×128 GB DIMMs on all six channels), which is configured in the AppDirect mode. It has 24 cores (48 hyperthreads) and a L3 cache of 35.75 MB. The server runs Arch Linux with kernel 5.10.11 and PMDK 1.7. All the code is compiled using GCC 12.1.0 with all optimization enabled.

For all hash tables, we first pre-load with 10 million records. Next, an insert-only benchmark with 190 million records is run. After insertion, we then execute 190 million records of positive search and negative search. Finally, we measured 190 million deletions on the table with 200 million records. The two Dash schemes employ MurMur hash, which is fast and can offer high-quality hashes. We use uniformly distributed keys for all the experiments. The fixed-length keys and their values are both set to 8 bytes. Pointers are used for the variable-length keys whose keys are set to 16 bytes and values are set to 8 bytes. Each result is the average of five independent runs. The code is available at https://github.com/CASL-Wpeng/PASS.

B. Single-thread Performance

Figure 2a shows that when fixed-length keys are used the positive search performance of PASS is 1.03×, 1.08× of that of Dash-EH, Dash-LH, respectively. This is mainly because assisted by the mapping table the speed of address mapping of PASS is higher than the hash function used by the existing schemes. In insertion, PASS achieves a speedup of 1.18×, 1.12× compared with Dash-EH, and Dash-LH, respectively.

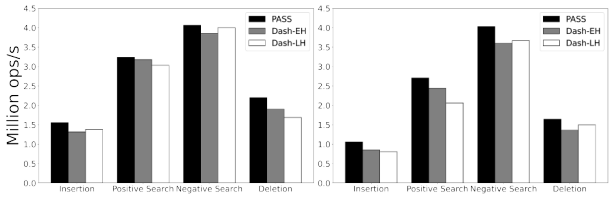


Fig. 2: Single-thread performance under (a) fixed-length keys (left) and (b) variable-length keys (right).

The main reason is that PASS rarely needs to perform rehashing after a segment is split, which saves many PM reads and writes. Figure 2b illustrates the results of variable-length key experiments. The positive search performance of PASS is 1.11×, 1.31× of that of Dash-EH, and Dash-LH, respectively. In insertion, PASS achieves a speedup of 1.24×, 1.31× compared with Dash-EH, and Dash-LH, respectively. In summary, PASS performs best in all basic operations under both fixed-length and variable-length keys.

V. Conclusion

Spiral storage was proposed several decades ago. Since then, it has been rarely applied to any real-world applications due to its limitations. In this research, we show that an optimized spiral storage could become a competent dynamic hashing scheme for PM just like its two peers. Although Intel discontinued the future development of Optane memory in 2022, researchers are still exploring new techniques for PM.

VI. Acknowledgement

We thank Kaisong Huang and Tianzheng Wang at Simon Fraser University for providing us with the server equipped with Intel DCPMM. This work was partially supported by the US National Science Foundation under grant CNS-1813485.

References

[1] J. Yang, J. Kim, M. Hoseinzadeh *et al.*, "An empirical guide to the behavior and use of scalable persistent memory," in *18th USENIX FAST*, 2020, pp. 169–182.

[2] P. Zuo, Y. Hua, and J. Wu, "Write-optimized and high-performance hashing index scheme for persistent memory," in *13th USENIX Symposium on Operating Systems Design and Implementation*, 2018, pp. 461–476.

[3] R. Fagin, J. Nievergelt, N. Pippenger *et al.*, "Extendible hashing—a fast access method for dynamic files," *ACM Transactions on Database Systems (TODS)*, vol. 4, no. 3, pp. 315–344, 1979.

[4] W. Litwin, "Linear hashing: a new tool for file and table addressing." in *VLDB*, vol. 80, 1980, pp. 1–3.

[5] R. J. Enbody and H.-C. Du, "Dynamic hashing schemes," *ACM Computing Surveys (CSUR)*, vol. 20, no. 2, pp. 850–113, 1988.

[6] F. Schmuck and R. Haskin, "GPFS: A shared-disk file system for large computing clusters," in *1st USENIX FAST*, 2002.

[7] J. Levandoski, D. Lomet, S. Sengupta *et al.*, "Indexing on modern hardware: Hekaton and beyond," in *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 2014, pp. 717–720.

[8] M. Nam, H. Cha, Y.-r. Choi *et al.*, "Write-optimized dynamic hashing for persistent memory," in *17th USENIX FAST*, 2019, pp. 31–44.

[9] B. Lu, X. Hao, T. Wang *et al.*, "Dash: Scalable hashing on persistent memory," *arXiv preprint arXiv:2003.07302*, 2020.

[10] J. K. Mullin, "Spiral storage: Efficient dynamic hashing with constant performance," *The Computer Journal*, vol. 28, no. 3, pp. 330–334, 1985.

[11] I. Oukid, J. Lasperas, A. Nica *et al.*, "FPTree: A hybrid SCM-DRAM persistent and concurrent b-tree for storage class memory," in *Proceedings of the SIGMOD*, 2016, pp. 371–386.

Authorized licensed use limited to: Univ of Calif San Diego. Downloaded on March 12,2025 at 18:22:45 UTC from IEEE Xplore. Restrictions apply.