VisionEmbedder: Bit-Level-Compact Key-Value Storage with Constant Lookup, Rapid Updates, and Rare Failure

Yuhan Wu*†, Feiyu Wang*, Yifan Zhu*, Zhuochen Fan*, Zhiting Xiong‡, Tong Yang*†, and Bin Cui*
*School of Computer Science, Peking University, Beijing, China, †Peng Cheng Laboratory, Shenzhen, China,
‡College of Computer, National University of Defense Technology, Changsha, China

***Abstract*—In key-value storage scenarios where storage space is at a premium, our focus is on a class of solutions that only store the value, which is highly space-efficient. While these solutions have proven their worth in distributed storage, networking, and bioinformatics, they still face two significant issues: one is that their space cost could be further reduced; the other is their are vulnerable to update failures, which can necessitate a complete table reconstruction.**

To address these issues, we introduce VisionEmbedder, a compact key-value embedder with constant-time lookup, fast dynamic updates, and a near-zero risk of reconstruction. VisionEmbedder cuts down the storage requirement from 2.2L bits to just 1.6L bits per key-value pair with an L-bit value, and it significantly reduces the chance of update failures by a factor of n, where n is the number of keys (for instance, 1 million or more). The compromise with VisionEmbedder comes with a minor reduction in query throughput on certain data sizes. The enhancements offered by VisionEmbedder have been theoretically validated and are effective across any dataset. Additionally, we have implemented VisionEmbedder on both FPGA and CPU platforms, with all codes made available as open-source.

I. Introduction

Key-Value (KV) storage, involving storing KV pairs and the fast lookup of values for user-given keys, plays a crucial role in computer science across various fields, including databases, operating systems, and networking [1]–[7]. In scenarios where high-speed storage space is limited and costly, such as CPU caches, on-chip registers in FPGAs, SRAM in ASIC chips, and local storages in distributed systems, space efficiency becomes the primary concern for KV storage.

Based on differences in space efficiency, we categorize existing works into two types: 1) Key-stored Solutions and 2) Value-Only (VO) tables. This paper focuses on the latter. Key-stored solutions, including cuckoo hashing, RocksDB, and LevelDB, store either the actual key or its hash value, the latter also known as a fingerprint. In contrast, VO tables, also known as multi-set classifiers and filters, like Bloomier [8], Othello [9], and Color [10], do not store the key or its fingerprint. These tables only store or encode the value, resulting in extremely high storage efficiency. For example, Bloomier can use as little as $1.23L$ bits for each KV pair when the value is L bits long. A VO table can significantly reduce space requirements, especially when the value's length is shorter than the key's, often by one or more orders of magnitude. It comes with a trade-off in handling alien keys[1]. This paper focuses on VO tables, and their application scenarios include:

- Lookup tables in the networking field, like the MAC address table with 48-bit key and 8-bit value in network switches [11]. Each entry in the MAC address table is a KV pair that records the network device's MAC address (48 bits) and its corresponding switch port (8 bits). This table is well-suited to be implemented with a VO table.
- Index for distributed storage (often with keys larger than 32 bits and values around 4 bits). In KV storage databases, when the data storage demand surpasses the capacity of a single node, the data is distributed across multiple backend nodes. Clients need to know the node's ID to access the data. Besides using directory servers or direct hash calculation for positioning, one method is to store a very small directory table on the client side. This table records the node ID for all data (or hot data), and is ideal for a VO table since it only needs to locate the storage node, requiring a short value length. The typical work is Smash [12].
- Others. In the biological field, SeqOthello [13] has used a VO structure for efficient mapping and querying processes. When the length of the value is one, VO structures can function as binary classifiers. In Log-Structured Merge-trees (LSMs), VO structures might be used to determine which SSTable contains the data.

The performance metrics of VO tables include three aspects: **1) Space Cost**. Since the VO table is primarily deployed in a hierarchical architecture consisting of limited-space fast storage and ample-space slow storage, such as the data/control plane in network devices, we focus only on the space occupied by fast storage when discussing space, aligning with existing works [9], [10]. **2) Lookup speed**. The table must provide lookup responses with minimal delay (should answer the value of the given key with low latency) and handle a large volume

Corresponding author: Tong Yang (yangtongemail@gmail.com).

[1]When queried for a key that has not been inserted or has been deleted, the system does not indicate the key's absence but returns a meaningless value instead.

TABLE I: Algorithm Comparison.

Algorithm	Space per L-bit value	Lookup Time	Update Performance	
			Failure Probablity	Amortized Time
Bloomier	$1.23L$ (bits)	$O(1)$	$O(\frac{1}{n})$	$O(n)$
Othello&Color	$2.33L, 2.2L$	$O(1)$	$O(1)$	$O(1)$
Ours	$1.6L$ (bits)	$O(1)$	$O(\frac{1}{n})$	$O(1)$

of requests efficiently (high throughput). **3) Update performance**. The table should allow for quick updates to data, including inserting, deleting, or altering KV pairs. Updates need to be executed rapidly to keep pace with changes in the data. If the table may require significant reconstruction during the update process, its probability must be sufficiently low.

However, existing VO tables have not well met the practical requirements of space efficiency, lookup speed, and update performance simultaneously. We categorize them into two types: static solutions that do not support incremental updates and dynamic solutions that do. The typical work of static solutions is the Bloomier filter [8], whereas dynamic solutions include Othello [9] and Color [10]. These solutions, like ours, operate within a fast-slow hierarchical architecture. We compare them in Table I based on the three performance aspects mentioned above. Bloomier offers the best space efficiency, but its update time is $O(n)$, meaning it takes time proportional to the number of elements when adding a new key. Othello and Color improve the update time from $O(n)$ to amortized $O(1)$, which means the average time per operation is constant, but this comes at the cost of almost doubling the space requirement. Moreover, they suffer from a significant drawback: a high probability of update failure, which is a constant rather than a negligible quantity, interrupting the update/lookup process and necessitating an $O(n)$ time to reconstruct the entire table.

The aim of this paper is to design a solution that excels in space efficiency, lookup speed, and update performance. In comparison to static solutions, we seek to enhance the update speed to a constant time without increasing the probability of needing to reconstruct. Against existing dynamic solutions, our goal is to achieve higher space efficiency and a state-of-the-art near-zero reconstruction risk.

Towards the design goal, this paper introduces a compact data structure, which we refer to as VisionEmbedder. **1) In terms of space cost,** VisionEmbedder utilizes only $1.6L$ bits for each value that is L bits in length. This is a reduction of $0.6L$ from the space taken by current dynamic update algorithms like Othello & Color, which use $2.2L$ bits, and is only $0.37L$ more than the most space-efficient static algorithm, the Bloomier filter. **2) Regarding lookup speed,** VisionEmbedder performs comparably to the fastest Othello & Color, with each having its strengths and weaknesses under various datasets, placing them on an equal footing. **3) Most notably, in the aspect of update failures,** VisionEmbedder reduces the theoretical update failure probability of Othello & Color by n times. Extensive empirical testing has demonstrated that VisionEmbedder decreases the average number of failures from approximately one per operation, as seen with Othello & Color, to a mere 0.001.

The working process of VisionEmbedder is essentially to solve a set of equations: For each key, select **three variables** from m variables by hashing[2], and establish an equation that the XOR (exclusive or) sum of the selected variables equals to the value in the KV pair. For n inserted KV pairs, there are n equations and there should be enough variables (*e.g.*, $m > 1.6n$) so that a feasible solution can be found. Actually, it is not challenging to solve such a set of equations statically.

The challenge is constantly incremental update: In amortized constant time, find a new solution when one equation (KV pair) changes or when a new equation (KV pair) is added. Specifically, when inserting a new equation, we need to modify one of its three variables to make it hold. All other equations that include the modified variable will be affected. To make these affected equations hold, for each one, we need to modify one of the other two unmodified variables. A modified variable may incur new affected equations. We repeat this process iteratively, and each iteration is one step. If the number of affected equations tend to decrease in iterations, then the update can be completed constantly.

To constantly find a new solution, our key technique, namely vision update, foresees a fast path with the least affected equations. A basic update strategy is to modify the variable included by the least number of equations. When the number of variables m is much larger than n, this method can find the new solution in a few steps. But the basic strategy is a local decision with only one-step vision. It cannot distinguish which is better when two variables are included by the same number of equations. What is worse, it is possible that less affected equations incurs more affected equations in the next step. Therefore, we use the Depth First Search (DFS) to look forward more steps and foresee the cost of modifying each variable, *i.e.*, the number of affected equations after more steps. The more steps (depth) we look forward, we will get a better choice while consume more time. We balance the time cost of looking forward and modifying the variables. With more inserted KV pairs, we dynamically increase the depth we look forward, to achieve the best overall efficiency.

We have implemented the VisionEmbedder on FPGA. The update scheme is calculated by the CPU, and the FPGA takes update message and performs high-speed lookup operation. The lookup scheme is FPGA friendly, which only needs three parallel hash calculations and memory reads, and then the lookup results can be obtained by XOR summing up the read results.

Key contribution:

- We devise VisionEmbedder, which is the first VO scheme with asmortized constant update time and $O(\frac{1}{n})$ failure

[2]We use three independent hash function to map one key to three variables, and select them.

probability. It is also space efficient, with a $1.6L$-bit space cost per KV pair with L-bit value.

- We prove our results by rigorous mathematical analysis and large-scale experiments. VisionEmbedder reduces update failure frequency from 1 to < 0.001, saves 50% redundant space, and achieves comparable update and lookup speed.
- We implement VisionEmbedder on an FPGA and achieve 279 million lookups per second for about 1 million key value pairs. All codes are available at Github [14].

II. Preliminaries and Motivation

Hierarchical Storage. VO tables are designed for a hierarchical storage, comprising two parts: (1) Slow space like DRAM, offering large space, and (2) Fast but limited space space such as SRAM. For instance, in network devices or FPGAs, the data plane is associated with fast space, and the control plane with slow space. In distributed storage, the local client corresponds to fast space, while the remote server represents slow space. Both parts have their own computing resources. Unless specified otherwise, when discussing space costs, we are specifically referring to the costs associated with the fast space.

VO Table Overview. VO tables consist of two parts: a compact Value Table in the fast space and a large Assistant Table in the slow space. Lookup operations only access the Value Table, while update operations access both the Value Table and the Assistant Table.

Value Table. We introduce a commonly used Value Table design, integral to various other data structures such as Bloomier filters [8], XOR filters [15], and Invertible Bloom Lookup Tables [16]. The Value table is a structure comprising three arrays, each containing w integers of L-bits in length. Within this structure, the t^{th} integer of the j^{th} array is denoted as $A_j[t]$. Each array is independently linked to a unique hash function, denoted as $h_j(\cdot)$, which maps an input key to one of the indices ranging from 1 to w within its corresponding array.

Static Construction. The table can be built for a static set of n key-value (KV) pairs. For each pair $\langle k_i, v_i \rangle$, it picks three integers via hash functions: $A_1[h_1(k_i)]$, $A_2[h_2(k_i)]$, and $A_3[h_3(k_i)]$, aiming to have their XOR sum equal the value v_i. This forms a series of equations for all pairs, simplified as $A_1[h_1(k)] \oplus A_2[h_2(k)] \oplus A_3[h_3(k)] = v$ for each k. To solve these equations efficiently, the Bloomier filter uses a fast, greedy algorithm, achieving linear time complexity ($O(n)$) with nearly 100% success if the table's capacity (m) is more than 1.23 times the number of KV pairs. This capacity helps minimize collisions and is effective for large datasets. Throughout this process, the complete KV pair and other information required by the algorithm are stored in the Assistant Table. However, Bloomier does not support dynamic updates effectively: Adding or modifying a KV pair necessitates the addition/alteration of an equation. For a long period, there has been no rapid method to adjust the table for such updates.

Dynamic Update. In recent advancements [9], [10] enabling dynamic updates, the structure of the value table has been modified from containing three arrays to just two. This modification, which simplifies the equations, enables the use of methods similar to cuckoo hashing [17] and the 'two choices' [18] principle, effectively facilitating rapid updates. During this process, the full KV pair and additional information are also stored in the Assistant Table. However, this approach comes with a significant trade-off in the form of a high failure probability. When two different key-value pairs hash to the same integer and their values differ, the equations become unsolvable, leading to a failure. In such cases, the data structure must switch hash functions and undergo a complete reconstruction. According to the birthday paradox [19], the probability of such occurrences is not infinitesimally small but rather a constant. This flaw is inherent in systems that rely on two-hash schemes.

Therefore, to circumvent this limitation, our choice is to employ a value table with three arrays (or potentially more) and to develop an algorithm that allows for dynamic updates. This approach aims to balance the need for efficient updates with the robustness of the data structure, minimizing the likelihood of failure while accommodating changes in the dataset.

Symbol	Description
n	Number of inserted KV pairs (Number of equations)
m	Number of integers (variables) in the value table
$A_j[t]$	The t-th integer of the j-th array in the value table
$S_j[t]$	The set of all KV pairs hashed to $A_j[t]$
$C_j[t]$	Size of the set $S_j[t]$

TABLE II: Key Symbols Used in This Paper.

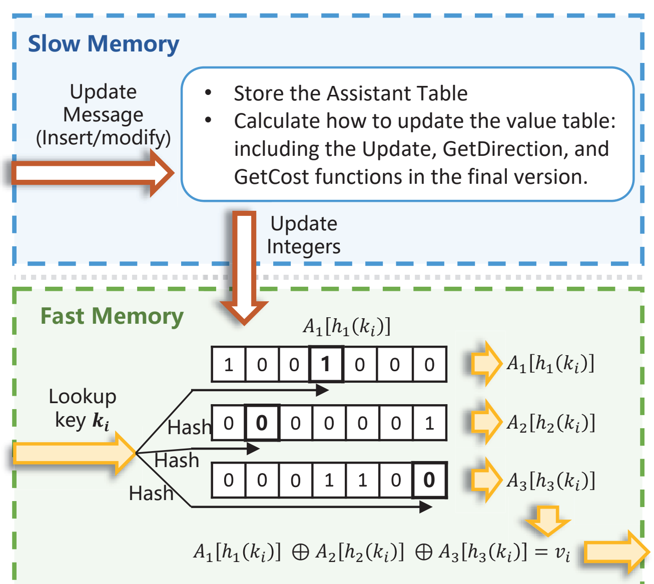


Figure 1: Overview that covers the hierarchical structure and lookup workflow with dynamic updates, and an example of a value table where the width is 7 and each integer is 1 bit.

III. System Overview for VisionEmbedder

In this section, we present the overview for VisionEmbedder: the workflow for lookup and dynamic updates, and the components of the data structure. We illustrate these aspects in Figure 1 and list key symbols in Table II.

Like other VO tables, VisionEmbedder is designed for a hierarchical storage structure. It has two tables, a compact *Value Table* in the fast space and a large *Assistant Table* in the slow space.

Workflow of Lookup and Dynamic Updates. VisionEmbedder supports various operations, including looking up the value of a key, inserting a new KV pair, modifying the value of an existing key, with the latter two categorized as dynamic updates. Similar to existing work, VisionEmbedder prioritizes lookup performance, focusing on throughput and latency, over the efficiency of dynamic updates. Thus, when looking up a key's value, only the fast space's value table needs to be accessed. This approach significantly enhances lookup performance by circumventing the slower space. For dynamic updates, the procedure starts with accessing and updating the assistant table in the slow space, figuring out what needs to be changed in the value table, and then updating the value table.

Value Table. The structure of the value table is as previously described (refer to Section II), comprising three arrays of integers. An equation is established for each KV pair:

$$A_1[h_1(k_i)] \oplus A_2[h_2(k_i)] \oplus A_3[h_3(k_i)] = v_i \tag{1}$$

When looking up key k_i, the computation $A_1[h_1(k_i)] \oplus A_2[h_2(k_i)] \oplus A_3[h_3(k_i)]$ is performed to obtain the result v. If the key has already been inserted, then the lookup result is guaranteed to be correct. Conversely, the result is a meaningless number, leaving the user unaware of the key's absence. Lookup optimization can be achieved by parallelizing hash function computations and simultaneous integer reads, enhancing throughput and latency.

Assistant Table. In the assistant table, located in the slow space with ample space, for each integer (denoted as $A_j[t]$) of the value table A, the assistant table records the number of keys mapped to $A_j[t]$ using a counter $C_j[t] = \sum_{\forall k_i}[h_j(k_i) = t]$. It also keeps track of the set of keys mapped to each integer in a bucket

$$S_j[t] = \{\langle k_i, v_i\rangle | h_j(k_i) = t.\}$$

This design ensures that the value table, located separately from the slow space, is effectively supported by the detailed record-keeping of the assistant table.

IV. Update Algorithms of VisionEmbedder

In this section, we start by proposing a simple strategy for updates, which allows for fast addition or change of key-value pairs. However, this method needs a larger table, resulting in about 140% more space usage. Then, we present a more refined update method, Vision Update. This approach is not only fast but also more space-efficient, requiring only 30% extra space compared to the static construction. This extra space is a reasonable trade-off for the benefit of dynamic updating in various scenarios. Lastly, we introduce other operations of VisionEmbedder.

A. Simple Update Algorithm

Inserting or modifying a KV pair is referred to as a dynamic update. Each such update for a key k_i results in the formation of a new equation, as outlined in Equation (1). If this new equation does not happen to hold, it becomes necessary to modify at least one of the integers from the set $S_{All} = \{A_1[h_1(k_i)], A_2[h_2(k_i)], A_3[h_3(k_i)]\}$. The main challenge in this process is minimizing the impact of these modifications on the other equations.

Our simple strategy is guided by two principles: (1) Limit the number of modified integers to reduce the ripple effect on other equations; (2) Quickly determine the integer to be modified to ensure high update performance. Following that, when an update requires changes to the equation, we choose to modify just one integer. This integer is selected randomly to boost the update process's speed and efficiency. This method is inspired by the 'randomly kick' technique found in Cuckoo hashing, which has rapid and effective decision-making in modifications.

The update process for a key-value pair consists of three steps, executed recursively: **Step 1:** Identifying a potential integer to modify, **Step 2:** Modifying this chosen integer, and **Step 3:** Updating all other keys impacted by this modification. They are outlined in Algorithm 1.

The `Update Function` is designed to accept three parameters: key, $value$, and S_{Fix}. Its function is to update the value of key to the new $value$. The parameter S_{Fix} serves as an auxiliary tool, indicating one integer that not be modified, thus preventing the algorithm from repeatedly modifying the same integer.

Initially, the user provides the key and $value$ to be modified, with S_{Fix} starting null. We then compute the hashes to locate the three integers linked to key, which are gathered in S_{All}. If S_{Fix} is not empty, it is excluded from S_{All}. Following this, we employ a decision function to choose one of the integers from S_{All} for modification.

Step 1: In the simple update algorithm, the decision function, named GetDirection, selects an integer at random, with equal probability. This selected integer is designated as $A_j[t]$.

Step 2: We then modify $A_j[t]$ according to the formula $A_j[t] = value \oplus \bigoplus_{x \in S_{All} \setminus \{A_j[t]\}} x$. This modification ensures that a lookup for the key will now correctly return the updated $value$.

Step 3: After modifying the integer $A_j[t]$, we use the assistant table to identify all KV pairs associated with it by $S_j[t]$, which includes all pairs hashed to $A_j[t]$. Then, the set for further updates is thus $S_j[t] \setminus \{\langle key, value\rangle\}$. Each pair in this refined set is then updated using the same function, ensuring that $A_j[t]$ is not modified again, which would otherwise result in an infinite loop. The update process is deemed complete when no additional keys or equations require updating.

Algorithm 1: Update Procedure

Procedure `Update`$(key, value, S_{Fix})$:
  $S_{All} \leftarrow \{A_1[h_1(key)], A_2[h_2(key)], A_3[h_3(key)]\}$;
  $A_j[t] \leftarrow$ `GetDirection`$(S_{All} \setminus S_{Fix})$;
  $A_j[t] \leftarrow value \oplus \bigoplus_{x \in S_{All} \setminus \{A_j[t]\}} x$;
  for $\langle k, v\rangle \in S_j[t] \setminus \{\langle key, value\rangle\}$ **do**
    `Update`$(k, v, \{A_j[t]\})$;

Function `GetDirection`(S):
  return $RandomSelect(S)$;

B. Vision Update Algorithm

The simple update algorithm is effective only when the number of inserted KV pairs is small within a fixed value table size. This is because it lacks sophistication in choosing which integer to modify, leading to an ever-increasing number of modifications and making it difficult to terminate the process.

To enable the insertion of more KV pairs and to minimize the number of modified integers for faster updates, a more advanced decision-making process is necessary. This process entails assessing the impact of modifying each integer and selecting the one with the least impact. A straightforward approach is to use the count of equations linked to an integer, indicated by $C_j[t]$ in the assistant table, as a measure of its modification cost. Then, our decision function selects the integer with the lowest $C_j[t]$.

However, this method is still not advanced enough. Firstly, if two candidate integers have the same $C_j[t]$, the method struggles to determine the better option. Secondly, an integer (denoted as a) with a smaller $C_j[t]$ (where $C_j[t] > 0$) doesn't necessarily imply fewer future modifications. For instance, integer a might be linked to only one equation (i.e., $C_j[t] = 1$), but the other two integers in that equation could be associated with many more equations. While modifying a impacts just one equation initially, the subsequent steps could affect several others.

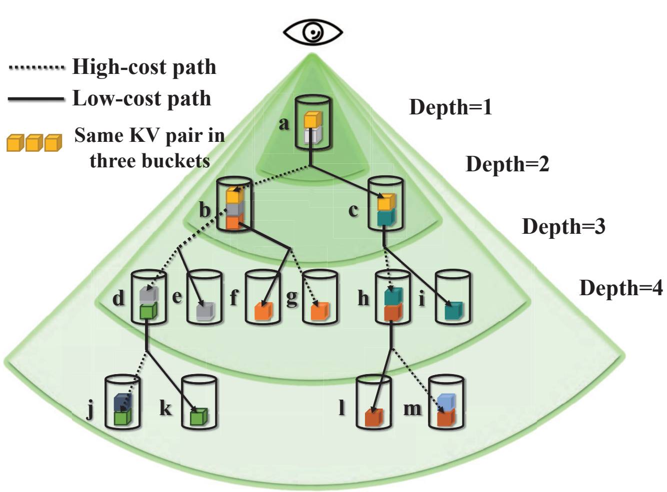


Figure 2: An example of finding a low-cost modification path.

Our method involves looking ahead a few steps to better estimate the modification cost. We predict the impact of modifications multiple steps forward, evaluate the costs of different modification paths, and then select the most efficient one.

Example. Before delving into the details of the algorithm, we present a specific example in Figure 2 to facilitate readers' understanding of the update process. In this figure, each bucket $S_j[t]$ in the assistant table corresponds to an integer in the value table, and the cubes inside the bucket are KV pairs mapped to that integer. Cubes of the same color in the figure represent identical KV pairs. Each key corresponds to an equation, so in our example, integers 'a', 'b', and 'c' are linked to the same equation marked by the yellow cube. Integers connected to the same equation are joined by line segments. Dashed lines represent modification paths with higher costs, while solid lines represent those with lower costs. If 'a' has been modified, we need to choose between modifying 'b' and 'c' to satisfy the equation with the yellow cube. We calculate the modification costs for both 'b' and 'c'. When evaluating 'c', the function recursively assesses the costs of modifying 'h' and 'i'. Thus, the cost for modifying 'c' involves two integers: 'c' and 'i'. Modifying 'b' would require changing 'e', and either 'f' or 'g', resulting in three integers. Therefore, we choose to modify 'c' and 'i', concluding the update process.

Cost Estimation. The essence of our vision update is to accurately estimate the cost of modifying an integer, aiming for nearly optimal modification decisions. As detailed in Algorithm 2 (function GetCost), the total cost of modifying an integer includes the base cost of "1" for the integer itself, plus the number of equations impacted by this modification. To begin, we identify equations linked to integer a that have not yet been updated. For each of these equations, we have a choice to modify one of two integers. We then recursively call our estimation function on these options and choose the one with the lower estimated cost. This chosen cost is added to the total cost associated with modifying integer a, enabling us to determine how to modify a to minimize overall impact.

Our cost calculation process is designed to be finite; we limit the recursion to a maximum depth, $MaxDepth$, to avoid excessive decision-making time. If the current recursion reaches $MaxDepth$, we use the number of equations related to the current integer a (indicated by $C_j[t]$) as the cost estimate. This approach ensures a balance between decision accuracy and efficiency.

Vision Update. To this end, we can devise a new decision function based on our estimation of modification costs (as outlined in the pseudo-code "GetCost"). This function operates by selecting the variable with the lowest estimated cost of modification from two (or three) options. By integrating this GetDirection function into the previously mentioned Update workflow, we arrive at the complete algorithm for VisionEmbedder (as detailed in Algorithm 2).

Update Speed Optimization— Dynamic Depth. To optimize overall performance, we have developed a mixed strategy that dynamically adjusts the $MaxDepth$ parameter in response

Algorithm 2: Our Final Algorithm

Function GetCost $(Depth, a, key, value)$**:**
  if $Depth = MaxDepth$ **then**
    return $C_j[t]$;
  $TotalCost \leftarrow 0$;
  for $\langle k, v\rangle \in S_j[t] \setminus \{\langle key, value\rangle\}$ **do**
    $S_{All} \leftarrow \{A_1[h_1(k)], A_2[h_2(k)], A_3[h_3(k)]\}$;
    $\{b, c\} \leftarrow S_{All} \setminus \{a\}$;
    $CostB \leftarrow$ GetCost $(Depth + 1, b, k, v)$;
    $CostC \leftarrow$ GetCost $(Depth + 1, c, k, v)$;
    $TotalCost \leftarrow TotalCost + \min(CostB, CostC)$;
  return $TotalCost + 1$;

Function GetDirection $(S, key, value)$**:**
  return $\arg\min_{x\in S}$ GetCost *(1, x, key, value)* ;

Procedure Update-SingleThread $(key, value, S_{Fix})$**:**
  $S_{All} \leftarrow \{A_1[h_1(key)], A_2[h_2(key)], A_3[h_3(key)]\}$;
  $A_j[t] \leftarrow$ GetDirection $(S_{All} \setminus S_{Fix}, key, value)$;
  $A_j[t] \leftarrow value \oplus \bigoplus_{x\in S_{All}\setminus\{A_j[t]\}} x$;
  for $\langle k, v\rangle \in S_j[t] \setminus \{\langle key, value\rangle\}$ **do**
    Update-SingleThread $(k, v, \{A_j[t]\})$;

Procedure ConcurrentBranches $(key, value, \mathbf{S}_\Delta)$**:**
  $S_{All} \leftarrow \{A_1[h_1(key)], A_2[h_2(key)], A_3[h_3(key)]\}$;
  $A_j[t] \leftarrow$ GetDirection $(S_{All} \setminus \mathbf{S}_\Delta, key, value)$;
  $\mathbf{ReadLock}(\mathbf{A_j[t]})$;
  $\mathbf{S}_\Delta \leftarrow \mathbf{S}_\Delta \cup \mathbf{A_j[t]}$;
  for $\langle k, v\rangle \in S_j[t] \setminus \{\langle key, value\rangle\}$ **do**
    ConcurrentBranches (k, v, S_Δ) ;

Procedure Update-Concurrent $(key, value)$**:**
  $S_{All} \leftarrow \{A_1[h_1(key)], A_2[h_2(key)], A_3[h_3(key)]\}$;
  $\mathbf{WriteLock}(\mathbf{A_k[h_k(key)], k \in \{1, 2, 3\}})$;
  Update $S_k[h_k(key)], C_k[h_k(key)] (k \in \{1, 2, 3\})$ in the Assistant Table;
  $V_\Delta = value \oplus \bigoplus_{x\in S_{All}} x$;
  $\mathbf{WriteUnLock}(\mathbf{S_{All}})$;
  $S_\Delta \leftarrow \emptyset$;
  ConcurrentBranches $(key, value, S_\Delta)$;
  for $A_j[t] \in S_\Delta$ **do**
    Atomic Update $A_j[t] \leftarrow A_j[t] \oplus V_\Delta$;
    $\mathbf{ReadUnLock}(\mathbf{A_j[t]})$;

to the insertion of more KV pairs. This approach effectively balances the trade-off between accommodating additional KV pairs and managing the time expended in the update process.

Looking ahead more steps (i.e., a greater $MaxDepth$) increases the success probability of inserting more KV pairs, thereby enhancing VisionEmbedder's space efficiency. Space efficiency, defined as $\frac{\text{Total size of all values}}{\text{Total size of the value table}} = \frac{n}{m}$, is a crucial metric. Higher space efficiency means more KV pairs can be accommodated.

However, a deeper lookahead also requires more processing time. Therefore, we adjust the $MaxDepth$ based on the current level of space efficiency. Specifically:

- When space efficiency is less than 0.2, we set $MaxDepth = 1$. This minimal depth allows for faster updates when the space is less utilized.
- For space efficiency in the range of $[0.2, 0.4)$, we increase the depth to $MaxDepth = 2$. This intermediate depth offers a balance between update speed and space utilization.
- When space efficiency exceeds 0.4, we set $MaxDepth = 3$. This maximum depth is used to maximize space efficiency, allowing for the insertion of the greatest number of KV pairs.

By employing this strategy, we achieve a balance between rapid updates and high space efficiency, ensuring that VisionEmbedder operates effectively under varying space conditions. Here, space efficiency is not equivalent to the load factor of ordinary hash tables. When the key and value are the same length, a VO table with 0.5 space efficiency consumes the same space as a ordinary hash table with 100% load factor.

Update Failure. When the update process does not complete quickly, for example when the Update function loops more than 50 times, this is defined as an update failure. If space efficiency is below 0.6, we consider this failure as random and suggest reconstructing the table with a different hash function. If not, we report a lack of space and advise the user to remove some entries or resize the table.

Concurrency. VisionEmbedder has a high performance design in multithreading. The primary challenge lies in resolving conflicts arising when multiple threads are responsible for updating different keys. We conducted a detailed analysis of conflict scenarios and applied locking minimally to mitigate the impact of locks.

Initially, we reduce frequent reads from the value table A during updates through an equivalent modification. When inserting a KV pair and modifying $A_j[t]$, we note that the modification increment is fixed. At the start, changing integer a to a' with an increment equal to $value \oplus \bigoplus_{x\in S_{All}} x$, denoted as V_Δ, ensures the new equation is valid. For other equations like $a \oplus b \oplus c = v$, modifying b to $a' \oplus b' \oplus c = v$ or c to $a' \oplus b \oplus c' = v$ works since their increments satisfy: $b \oplus b' = a \oplus a'$, $c \oplus c' = a \oplus a'$. Thus, recording V_Δ initially avoids frequent reads from A. All integers requiring modification (modification path) are added to a set S_Δ, which are all incremented by V_Δ after the search concludes.

The update process is divided into two parts: 1) Adding a new key to the Assistant Table and calculating the XOR value increment V_Δ needed for modifying integers. 2) Finding the modification path (S_Δ) and modifying it according to V_Δ. Each integer $A_j[t]$ and its Assistant Table entries $S_j[t]$ and $C_j[t]$ are protected by a dedicated Reader-Writer lock (Shared_Mutex), termed a "unit." In part 1, we apply a "write lock" to three units, exclusive to that unit and mutually exclusive with other threads' read and write locks, released at the end. In part 2, a read lock (compatible with other threads' read locks) is applied to each unit in the modification path. Updates to $A_j[t]$ are made with atomic write operations (without needing a write lock), and the read lock is then released.

This design ensures high performance and correctness: 1) Part 2 is the most time-consuming, but read locks allow different threads' part 2 to run entirely independently. Part

1's scope, which cannot be reduced, hardly allows concurrent operations with any operation within other threads' parts 1 and 2. 2) Except for GetCost, the described locks provide adequate protection. GetCost is only affected by part 1, possibly leading to a suboptimal update direction. However, its occurrence is low enough ($O(T/n)$, where T is the number of threads) to be manageable when n is significantly larger than T. For smaller n, we implemented a search backtrack feature in the code (not shown in the pseudocode) to avoid failures caused by inaccurate GetCost.

C. Other Operations

Delete Operation. Since VisionEmbedder, like other VO tables, returns a meaningless value for the alien key, the deletion of KV pairs does not need to modify the value table in the fast space. We only need to update the assistant table, including subtracting 1 from the counters of the three positions indexed by this key, and deleting this key from the table that records the set of keys mapped to each position. After deletion, the removed KV pair no longer occupies space or affects subsequent updates.

Reconstruct Operation. When an update failure occurs, users have the option to reconstruct the entire table. This involves changing all hash functions and then reconstructing both the assistant table and the value table. The construction of the value table can either use the existing static construction method of Bloomier or our dynamic update scheme to insert KV pairs one by one.

V. Theoretical Analysis

In this section, we present the theoretical analysis for VisionEmbedder. Our focus is on two main achievements: the space efficiency we can attain and our very low failure probability. The key results we have proven are as follows:

1) **High Space Efficiency.** VisionEmbedder with MaxDepth=1 can successfully perform dynamic updates, i.e., stop in amortized constant time, when the space usage is greater than 1.756L per L-bit value. Here, 1.756 represents $\frac{m}{n}$, indicating the space required to encode per 1-bit value and is inverse to space efficiency ($\frac{n}{m}$).
2) **Low Failure Rate.** For n consecutive insertions, the probability of encountering an update failure is $O(\frac{1}{n})$.

A. High Space Efficiency

The goal of analyzing Space Efficiency is to find a threshold of n/m (the ratio of equations to variables), below which the update algorithm of VisionEmbedder can successfully operate, specifically, stopping in amortized constant time. In detail, when updating a KV pair, VisionEmbedder selects the variable with the lowest modification cost among three for adjustment to meet the equation. However, this modified variable impacts other equations. If this chain of effects leads to a progressively decreasing number of variables needing modification, then the count of variables to be adjusted exponentially declines, reaching zero within amortized constant time, ensuring a successful VisionEmbedder update. If not, the VisionEmbedder update is likely to never terminate.

Consider a scenario where n pairs have been inserted into VisionEmbedder which contains m variables/integers/buckets. Given that each key is mapped to 3 buckets, the total number of hashed positions is $3n$. With each key having an equal probability $\frac{1}{m}$ of being hashed to any bucket, and considering the independence of each bucket, the number of keys hashed to a single bucket can be represented by a random variable X. We assume X follows a Poisson distribution with parameter $\lambda = 3n/m$, i.e., $X \sim Pois(\lambda)$.

The update process involves recursively modifying the bucket with the lowest modification cost to satisfy the equations. Changing a bucket's value necessitates considering all keys hashed to this bucket in subsequent recursions. The number of keys hashed to the chosen bucket is crucial in this recursion. The algorithm selects the bucket with the minimum number of hashed keys. We denote the number of hashed keys in the selected bucket as X_{min}. The algorithm converges if and only if the expected value $E[X_{min}]$ is less than 1.

Theorem 1. *If the ratio of m/n exceeds 1.756, the update algorithm (MaxDepth=1) is expected to converge.*

Proof.

$$E\left[X_{min}\right] = \sum_{k=1}^{n} k \times P\left(X_{min} = k\right) \tag{2}$$

Since $P(X_{min} = i) = P(X_{min} \geq i) - P(X_{min} \geq i+1)$, from the above equation we have $E[X_{min}] = \sum_{k=1}^{n} P(X_{min} \geq k)$.

Since $P(X = k) = \frac{\lambda^k}{k!}e^{-\lambda}$ as $X \sim Pois(\lambda)$, we have

$$P(X_{min} \geq k) = P(X \geq k)^2 = (\sum_{i=k}^{\infty} \frac{\lambda^i}{i!}e^{-\lambda})^2$$
$$E[X_{min}] = \sum_{k=1}^{n} P(X_{min} \geq k) = \sum_{k=1}^{n} \left(\sum_{i=k}^{\infty} \frac{\lambda^i}{i!}e^{-\lambda}\right)^2 \tag{3}$$

According to Equation (3), the convergence depends on the parameter $\lambda = \frac{3n}{m}$. $E[X_{min}]$ and parameter λ have positive correlation. Therefore, there must exist a threshold λ' such that $E[X_{min}] < 1$ if $\lambda < \lambda'$. We conduct a numerical simulation to solve for λ' and the result shows that $\lambda' \approx 1.709$. The corresponding $\left(\frac{m}{n}\right)' = 1.756$. Thus, if the ratio of m/n exceeds 1.756, the update algorithm is expected to converge. □

B. The Probability of Update Failure

In this section, we analyze the probability of update failure, which includes two cases. The first case is when the equation itself has no solution. The second case is when the update process chooses an incorrect path, creating an endless loop and leading to failure. We prove that both scenarios happen with a likelihood of $O\left(\frac{1}{n}\right)$, as detailed in Theorem 2 and

Theorem 3. This proves that the chance of an update failure is also $O\left(\frac{1}{n}\right)$.

For the first case where the equation has no solution, we calculate a basic unsolvable situation known as collision error. Although this is a simple case, it has been demonstrated in existing studies to have the highest probability of occurrence compared to other unsolvable scenarios, which are considered mathematically negligible [16], [20]. Therefore, demonstrating that the probability of a collision error is $O\left(\frac{1}{n}\right)$ can prove that the probability of the first scenario (the equation being unsolvable) is also $O\left(\frac{1}{n}\right)$.

A collision error refers to the situation where two different keys are hashed to the same three variables, which means the VisionEmbedder structure cannot find a solution. The formal definition is as follows:

Definition 1. Collision Error. *Given m representing the total number of buckets (assuming m is divisible by 3 for simplicity), a key k_i is hashed to three buckets indexed as k_{i1}, k_{i2}, k_{i3}. Each k_{ij} falls within the range $[(j-1)*m/3, j*m/3-1]$ for $j=1,2,3$. If two different keys k_a, k_b hash to the same indices $k_{aj}=k_{bj}$ for $j=1,2,3$, it results in a collision error.*

Lemma 1. *For two keys, the probability of an error collision is denoted by $P(E)$ and calculated as*

$$P(E) = \left(\frac{m}{3} \times \left(\frac{1}{\frac{m}{3}}\right)^2\right)^3 = \left(\frac{3}{m}\right)^3 \tag{4}$$

Theorem 2. *For n consecutive insertions, the probability of encountering a collision error is $O\left(\frac{1}{n}\right)$.*

Proof. With n keys independently hashed to 3 buckets, and referring to Equation (4), the probability of no collision is

$$\begin{aligned} P(no\ collision) &= (1-P(E))^{C_n^2} \approx \left(1-\left(\frac{3}{m}\right)^3\right)^{\frac{n^2}{2}} \\ &= \left(1-\left(\frac{3}{m}\right)^3\right)^{\left(\frac{m}{3}\right)^3\left(\frac{3}{m}\right)^3\times\frac{n^2}{2}} \\ &\approx e^{-\left(\frac{3}{m}\right)^3\times\frac{n^2}{2}} = 1-O\left(\frac{n^2}{m^3}\right) \end{aligned} \tag{5}$$

According to Formula 5, for n consecutive insertions, the probability of encountering a collision error is $O\left(\frac{1}{n}\right)$. □

The second case is an endless loop leading to update failure. We follow the situation described in Theorem 1, which states that the update procedure will eventually converge. It is reasonable to assume that an update process requires z modifications, where z is a constant and does not depend on n.

Lemma 2. *For one update, the probability of the endless loop is $O(\frac{1}{n^2})$.*

Proof. The modification to a bucket requires all keys hashed to this bucket to be further updated. There are two candidate buckets to select and modify, because the third one has just been modified. An endless loop occurs if both candidate buckets have already been modified in this update. Since hash operations are independent, we can assume that each modification is independent from the others. Given m buckets and n keys, the probability of an endless loop for each modification is $P(F_1)=\frac{1}{n^2}$. As there are z modifications in total and they are all independent, the probability of an endless loop, $P(F_2)$, is $(1-\frac{1}{n^2})^z \approx 1-(1-z\times\frac{1}{n^2}) = \frac{z}{n^2} = O(\frac{1}{n^2})$, assuming z is constant. □

Theorem 3. *For t consecutive updates, the probability of the endless loop is $O(\frac{t}{n^2})$.*

Proof. Considering t consecutive update, the probability of update failure is $1-(1-\frac{z}{n^2})^t \approx 1-(1-t\times\frac{z}{n^2}) = \frac{zt}{n^2}$. Since z is constant, the probability of update failure for t consecutive update is $O(\frac{t}{n^2})$. □

VI. Experiment Results

In this section, we compare our algorithm with other Value-Only solutions, including Bloomier [8], Othello [9], Ludo [21], and Coloring Embedder [10]. Regarding space cost and failure probability, we conducted experiments on a CPU server, demonstrating that it can achieve $1.58L$ bits per L-bit value and a failure probability of $O(1/n)$. Additionally, we evaluated the lookup and update performance, as well as robustness, across various datasets. To showcase versatility across multiple platforms, we also present a case study on FPGA platforms to demonstrate its suitability for specialized hardware.

A. Experiment Setup

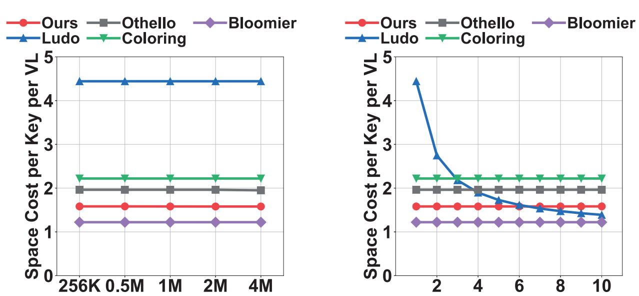
1) Methodology: We compare VisionEmbedder with prior art from five aspects: space cost, the frequency of update failure, update performance, look performance, and robustness against datasets. It's worth noting that we use the frequency of update failures to assess the stability of these algorithms during update operations. Furthermore, we employ both throughput and latency as metrics to evaluate the performance of lookup and update operations. The specific definitions of key metrics are as follows:

- **Throughput:** Million Operations Per Second (Mops). We use *Throughput* to evaluate the average lookup/update speed.
- **Latency**. We use the percentiles of *latency* to evaluate to performance of lookup/update operations. The tail latency can show the update performance when the data structures are nearly full.
- **Space Cost** = $\frac{\text{the space of the value table}}{\text{the number of KV pairs}\times\text{the value length}}$. We use the space cost incurred per bit of value encoded to evaluate the space efficiency of each algorithm. Lower space cost indicates an algorithm has better space efficiency.

2) Datasets: We use synthetic random datasets and three real-world datasets for experiments. The synthetic random datasets consist of varying numbers of KV pairs with different value lengths in our experiments. These datasets are sufficiently persuasive since our algorithm does not utilize any distribution characteristics of the key-value pairs. Randomly creating these pairs makes sure our results are consistently good across all dataset distributions, matching or surpassing the outcomes of other datasets. It's important to note that malicious activities, such as stealing hash functions to deliberately create collisions, are outside the scope of this paper. We establish some terminologies for clarity: "dataset size" refers to the number of KV pairs and the "value length" is L. We vary the L from 1 bit to 10 bit to study how the L influences the performance of algorithms. We use three real-world datasets in experiments, including:

- **MACTable**. This dataset is drawn from the MAC table file in [22], which consists of 2731 distinct KV pairs. The key is a MAC address and the value represents the type field (static or dynamic). The value length of MACTable is 1 bit.
- **MachineLearning**. This dataset is a dataset for binary classification tasks from UCI machine learning repository [23]. Each KV pair represents an entry in the training set and the value is the label of the entry. The dataset size is 359874 and the value length is 1 bit.
- **DBLP**: This dataset is drawn from DBLP [24]. We use the "key" attribute as the key. The value represents whether a record is from a journal or a conference. The dataset consists of 829119 distinct KV pairs and the value length is 1.

3) Implementation: We implement VisionEmbedder in C++. During all of the experiments, we use the well-known MurmurHash [25] as the hash function in VisionEmbedder. We utilized the open-source implementation of prior art and fixed their bugs. The parameters of Othello, Ludo, and Color are configured according to their original papers. Specifically, by default, Bloomier, Othello, Color, Ludo, and VisionEmbedder consume space of 1.23L*(n+100), 2.3L*n, 2.22L*n, (3.76 + 1.05L)*n bits, and 1.7L*n bits for L-bit values, respectively. We perform our experiments on a server with an 18-core CPU (Intel® Core™ i9-10980XE @3.00 GHz) and 128 GB memory. We deploy our algorithm on the FPGA platform as a case study.



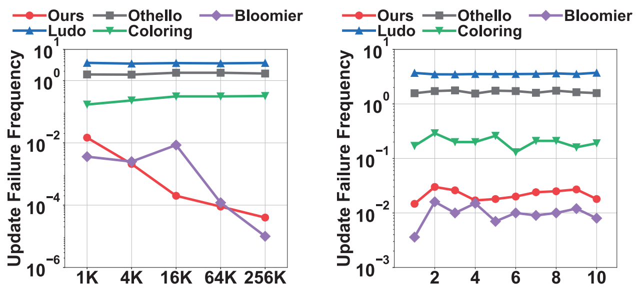
(a) Varying n (b) Varying L

Figure 3: Space cost under different dataset size & value length. The y-axis is $\frac{\text{the space of the value table}}{\text{the number of KV pairs*the value length}}$, indicating the storage cost incurred per bit of value encoded.

B. Space Cost Comparison

We assess the space cost by the minimum fast space required by these five algorithms to function effectively. This minimal space was determined by initially providing each algorithm with ample space, then incrementally reducing this space until the update failure frequency exceeded five times throughout the entirety of the data insertion.

Figure 3 illustrates that our algorithm, VisionEmbedder, requires the least space (1.58 bits) for 1-bit values. The $1.58L$, compared to the existing $2.2L$, reduces redundancy towards the optimal L by 50%. By default, we will set the space for VisionEmbedder at 1.7L to achieve the best overall performance. Ludo's efficiency, at 3.76 + 1.05L bits, may be better for L values over 6 bits, but our approach, VisionEmbedder, remains valuable for larger L sizes for two reasons: it can improve Ludo's space efficiency to approximately 3.1 + 1.05L bits by replacing its internal Othello component, and it has a significantly lower update failure probability.



(a) Varying n (b) Varying L

Figure 4: Update failure frequency.

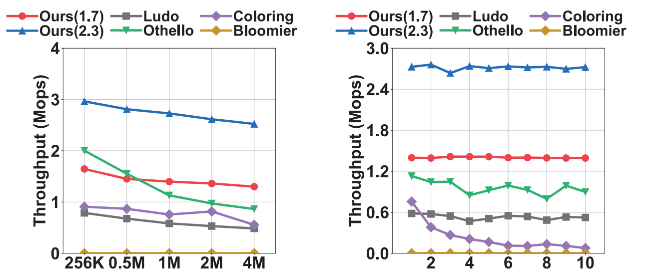
C. Update Failure Frequency Comparison

Existing VO solutions suffer from the update failure, which requires changing the hash seed and reconstructing the entire data structure, a process that is extremely time-consuming. We first assess the frequency of update failures, and then in § VI-D, we measure the impact of update failures on update throughput and latency. To assess these algorithms' update failure frequency, we inserted the entire dataset and recorded how often each algorithm had to reconstruct itself. This process was repeated 100,000 times to ensure reliable results. For space occupancy, we used the aforementioned default settings.

Figure 4 illustrates that our algorithm, VisionEmbedder, experiences far fewer update failures compared to other synamic solutions. This demonstrates that VisionEmbedder offers more stable insertions, fewer update failures, and, consequently, better dynamic performance. At the same time, it can be observed that the update failure rate of VisionEmbedder approximately decreases linearly with the increase in dataset size n, which is consistent with our theoretical expectations. The reason Bloomier performs well at smaller values of n is that its space overhead includes an added constant of 100, as recommended in the original paper, to ensure a significant success rate even when n is small.

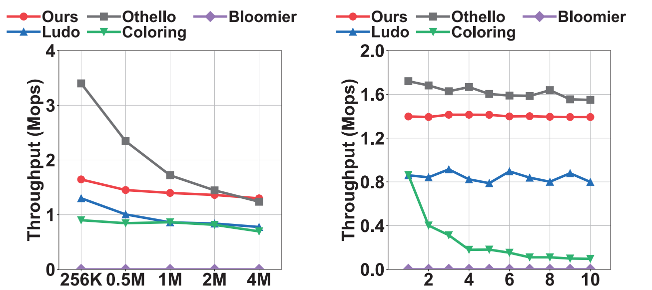
D. Update Performance Comparison

Figure 5 shows that VisionEmbedder achieves the best throughput among all algorithms. Compared to Othello, Vi-



(a) Varying n (b) Varying L

Figure 5: Overall update throughput including reconstruction.



(a) Varying n (b) Varying L

Figure 6: Update throughput excluding reconstruct time

sionEmbedder (utilizing 2.3L of space) is faster at the same space cost. Under its default configuration, VisionEmbedder uses less space, making updates more challenging and slower, and is therefore comparable to Othello in performance. VisionEmbedder with the default configuration still outperforms Othello under large-scale datasets.

The aforementioned throughput includes the time spent on reconstructions due to update failures. However, in Figure 6, we have excluded the time for reconstructions. The results show that the throughput of Othello, Color, and Ludo has improved to a certain extent because they have a higher probability of requiring reconstruction.

The distribution of update latency are shown in Figure 7. The tail latency of Othello, Ludo, and Color is significantly higher than that of VisionEmbedder. Users employing these algorithms may have to endure significant latency inflation, with a high likelihood of encountering such issues. On the other hand, VisionEmbedder exhibits a relatively lower probability of significant tail latency.

E. Lookup Performance Comparison

Figure 8 show that the lookup throughputs of VisionEmbedder and Othello are comparable, and both are faster than Ludo, Color, and Bloomier.

Figure 8(a) shows the performance of various algorithms under different values of n when L=1. The results indicate that at smaller values of n, VisionEmbedder outperforms Othello due to its smaller space allowing for better cache utilization. As n increases, Othello gains an advantage because it requires only two memory accesses, compared to our three. These factors affect throughput, making VisionEmbedder and Othello comparable. From an algorithm design perspective, Bloomier

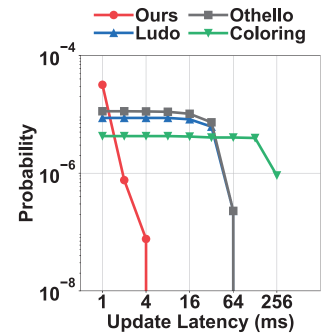
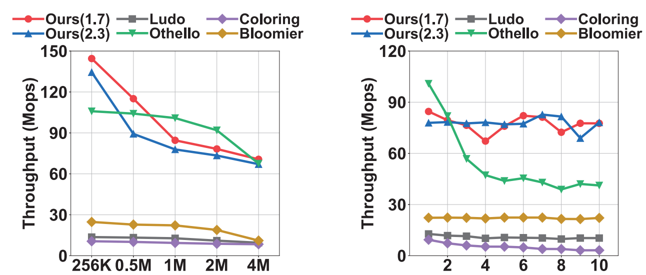


Figure 7: The distribution curve of update latency with $n = 64k(2^{16}), L = 1$. The figure illustrates the probability of update latency exceeding a certain time (on the x-axis).



(a) Varying n (b) Varying L

Figure 8: Lookup throughput.

and Vision should achieve similar lookup speeds, as Vision is an enhancement of Bloomier for dynamic updates. However, due to poor implementation of existing Bloomier, the actual throughput is not high.

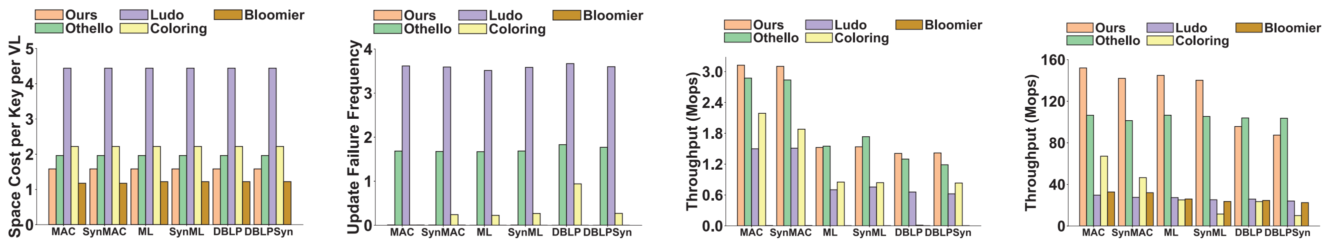
Figure 8(b) shows that as L increases with n fixed at 1M, the lookup performance of Vision, Bloomier, and Ludo remains essentially unchanged, while Othello and Color show a significant decrease in throughput. This is because their time consumption is directly proportional to L.

F. Robustness

Robustness against datasets. We perform experiments across three real-world datasets and three corresponding synthetic datasets of the same scale. These test datasets are derived from existing work, with their queries sampled from the key set according to a Zipf distribution with $\alpha = 1.0$, since the datasets did not include queries. The queries in the synthetic datasets are uniformly distributed. Figure 9 shows that whether the dataset is synthetic or not does not affect VisionEmbedder's space cost and update performance. When query keys are not uniformly random but skewed, the presence of cache leads to a slight improvement in lookup throughput.

Stability. We demonstrate that the performance of VisionEmbedder remains stable under the influence of randomness arising from varying hash seeds:

- Speed. We conducted update and lookup operations on VisionEmbedder using various hash seeds. The results, as depicted in Figure 10 and Figure 11, indicate that VisionEmbedder maintains stable performance across dif-

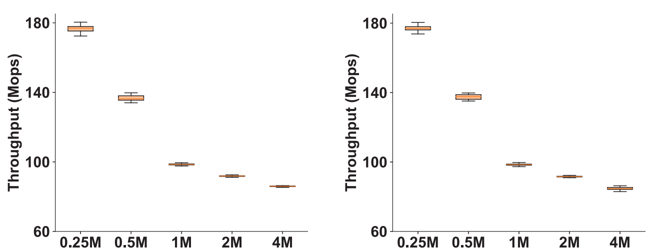


(a) Space cost. (b) Updata failure frequency. (c) Update throughput. (d) Lookup throughput.

Figure 9: Robustness against datasets. The Syn$X$ refers to the synthetic dataset of the same scale as dataset X.

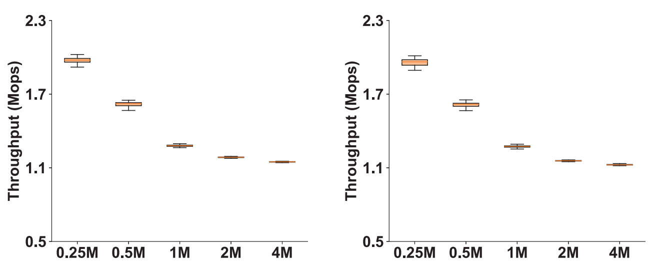
ferent hash seeds, even with changing dataset sizes and value lengths.

- Space Efficiency. We alter hash seeds to test the space consumption of VisionEmbedder, employing a methodology similar to that of Figure 3. The experiment results, as shown in Figure 12, indicate that the hash seed has nearly no impact on space efficiency.



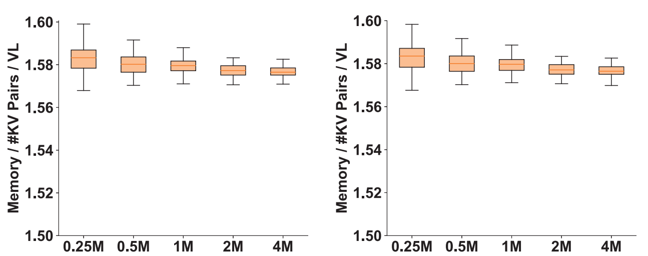
(a) L=1 (b) L=8

Figure 10: Lookup throughput with different hash seeds.



(a) L=1 (b) L=8

Figure 11: Update throughput with different hash seeds.



(a) 1-bit VL (b) 8-bit VL

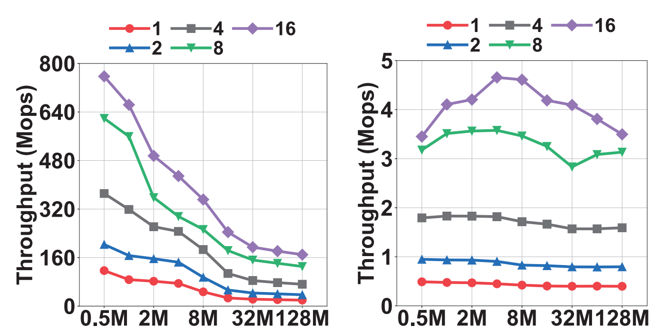
Figure 12: Space cost with different hash seeds. The y-axis is $\frac{\text{the space of the value table}}{\text{the number of KV pairs*the value length}}$, indicating the storage cost incurred per bit of value encoded.

G. Deletion Performance

Similar to Figure 5, we test the deletion performance of VisionEmbedder at n=256k, 512k, 1M, 2M, 4M, achieving throughputs of 6.60, 5.62, 5.35, 5.10, 4.92 MOPS, respectively. At n=256k, with space usage set to 1.7L, 1.9L, 2.1L, and 2.3L, the throughputs are 6.60, 6.61, 6.53, and 6.24, respectively. Overall, deletion throughput is lower than that for lookups but higher than for updates, mainly depending on the access speed of slow memory.

H. Multi-threading

We assess the performance of VisionEmbedder's multi-threaded concurrency by varying the data scale under 1 to 16 threads, evaluating the throughput for updates and lookups separately. The results (Figure 13) demonstrate that both update and lookup operations in VisionEmbedder are well-suited for acceleration using multithreading. At a data scale of 1M, 2, 4, 8, and 16 threads achieved respective speedups of 1.96, 3.84, 7.37, and 8.61 times for update acceleration, and 1.91, 3.65, 6.41, and 7.61 times for lookup acceleration. We observed that the multithreaded version's update failures and space cost did not show noticeable changes compared to the single-threaded version, with no differences found in relevant tests, hence the corresponding experiment figures are not shown. Note that the update performance of VisionEmbedder under 1 thread is worse than the result in Figure 5 due to the overhead caused by multithreading.



(a) Lookup. (b) Update.

Figure 13: Update and lookup throughput with different number of threads.

I. A Case Study: FPGA Implementation

We implement the VisionEmbedder algorithm on an FPGA platform as a case study, to demonstrate that VisionEmbedder can indeed be implemented in hierarchical structures beyond CPU servers. The FPGA integrated with the platform is

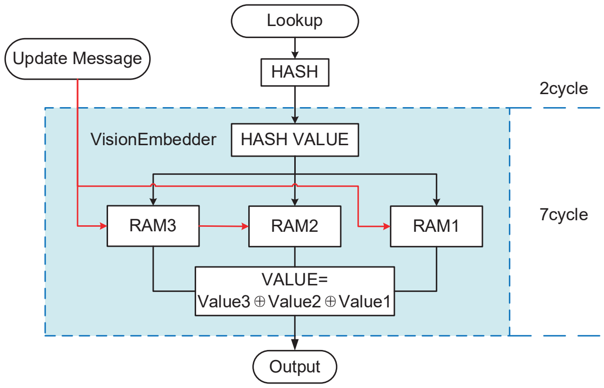


Figure 14: FPGA Implementation Architecture.

XCVU13P-L2FLGA2577 with 1728000 CLB LUTS, 3456000 CLB Register, and 2688 Block RAM.

1) Architecture Design (Figure 14): The scheme mainly consists of 2 hardware modules: hash module and VisionEmbedder module. The lookup procedure first processes the input key with CRC-32 in the hash module. VisionEmbedder module then locates 3 positions in RAM, combines them using exclusive OR operation, and outputs the value stored corresponding to the key. The insert procedure consists of two phases. In phase I, the control plane (slow space in CPU) conducts the insert algorithm and generates several update messages $(location,\ value)$ to indicate the modification in fast space (FPGA implementation). In phase II, fast space updates RAM accordingly.

TABLE III: FPGA Implementation Results.

Module	CLB LUTS	CLB Register	Block RAM	Frequency (MHz)
Hash	76	66	0	279.64
VisionEmbedder	505	631	385	279.64
Total	581	697	385	279.64
Usage	0.03%	0.02%	14.32%	

2) Evaluation Result: Table III shows the evaluation result of FPGA-based VisionEmbedder. According to the synthesis report, when the depth of RAM is $2E19$, $i.e.$, it can store about 0.95 million KV pairs with 8-bit value, the clock frequency of our implementation in FPGA is 279.64 MHz, meaning the throughput of lookup can be 279.64 Mops. Meanwhile, the logic usage is 0.03% and the space usage is 14.32%.

VII. Related Work

We classify existing works into two categories: 1) Key-stored Solutions, and 2) Value-Only (VO) tables. Our focus in this paper is on the latter, which we further divide into static solutions that do not support incremental updates and dynamic solutions that do.

Static VO solutions include Bloomier filter [8] and Perfect Hashing methods [26]–[30]. Among these, Bloomier filters are noted for their high memory efficiency, but they suffer from a long update time of $O(n)$. Our value table bears resemblance to Bloomier filter's $Table_1$, yet we do not employ a mask. The similarity stops there. Beyond this similarity, the innovative aspect of our work lies in the method of updating the table. Like Bloomier filters, most perfect hashing methods do not support constant time updates. Although recent developments in perfect hashing, such as the MapEmbed [31], allow for dynamic updates, they represent a special case of perfect hashing that stores keys.

Dynamic VO solutions include Othello [9], Ludo [21] and Color [10]. They can update in amortized constant time. Different from Bloomier, their approach is to select only two variables per key to establish equations. This approach makes constant time update possible, but introduces a high probability of update failure: With constant probability, the equations have no feasible solution. In this case, when inserting a specific KV pair, the algorithm should change the hash function and rebuild the whole table. We call this update failure. The update failure causes some insert operations to experience a long pause, and pause the lookup. The rebuilt is unacceptable for real-time applications. Our work, as a dynamic VO solution, can operate independently as well as become a component of other data structures such as ChainedFilter [32].

Key-stored Solutions are not the focus of our study, as they are less space efficient in scenarios where keys are long and values are short. Compared to VO tables, their main advantage lies in the ability to detect outliers, meaning they can return a ”non-existent” response when queried for keys that have not been inserted. In contrast, VO tables respond to queries for such outlier keys with a random answer. Typical key-stored solutions include RocksDB [33], [34], Redis [35], Memcached [36], Twemcache [37], and more [38]–[49].

VIII. Conclusion

This paper presents VisionEmbedder, the first value-only key-value lookup solution with amortized constant update time and $O(\frac{1}{n})$ failure probability. Compared with existing solutions, it reduces the update failure by n times, saves 50% redundant memory from $2.2L$ bits to $1.6L$ bits, and achieves comparable update/lookup speed. We prove our results by rigorous mathematical analysis and extensive experiments. We also implement VisionEmbedder in FPGA.

Acknowledgment

We would like to thank the anonymous reviewers for their constructive feedback. Thanks to Tianbao Zhou and Bowen Ye for their help during the revision stage. This work was supported in part by the National Key R&D Program of China (No. 2022YFB2901504), in part by the National Natural Science Foundation of China (NSFC) (No. U20A20179, 62372009), and in part by the China Postdoctoral Science Foundation (No. 2023TQ0010, GZC20230055).

References

[1] Fenggang Wu, Ming-Hong Yang, Baoquan Zhang, and David H.C. Du. Ac-key: Adaptive caching for lsm-based key-value stores. In *USENIX ATC '20*, 2020.

[2] Hagar Meir, Dmitry Basin, Edward Bortnikov, and et al. Oak: A scalable off-heap allocated key-value map. In *PPoPP '20*, 2020.

[3] Yu-Pei Liang, Tseng-Yi Chen, Ching-Ho Chi, and et al. Enabling a B^+-tree-based data management scheme for key-value store over smr-based sshd. In *DAC'20*, 2020.

[4] Yongkun Li, Zhen Liu, Patrick PC Lee, and et al. Differentiated key-value storage management for balanced i/o performance. In *USENIX ATC'21*, 2021.

[5] Chanwoo Chung, Jinhyung Koo, Junsu Im, and et al. Lightstore: Software-defined network-attached key-value drives. In *ASPLOS '19*, 2019.

[6] Yandong Mao, Eddie Kohler, and Robert Tappan Morris. Cache craftiness for fast multicore key-value storage. In *EuroSys '12*, 2012.

[7] Roxana Geambasu, Amit A Levy, Tadayoshi Kohno, and et al. Comet: An active distributed key-value store. In *OSDI' 10*, pages 323–336, 2010.

[8] Bernard Chazelle, Joe Kilian, Ronitt Rubinfeld, and et al. The bloomier filter: an efficient data structure for static support lookup tables. In *SODA'04*, 2004.

[9] Ye Yu, Djamal Belazzougui, Chen Qian, and et al. Memory-efficient and ultra-fast network lookup and forwarding using othello hashing. *IEEE/ACM Trans. Netw.*, 26(3), 2018.

[10] Yang Tong, Dongsheng Yang, Jie Jiang, and et al. Coloring embedder: A memory efficient data structure for answering multi-set query. In *ICDE' 19*, 2019.

[11] Mathy Vanhoef, Célestin Matte, Mathieu Cunche, Leonardo S Cardoso, and Frank Piessens. Why mac address randomization is not enough: An analysis of wi-fi network discovery mechanisms. In *Proceedings of the 11th ACM on Asia conference on computer and communications security*, pages 413–424, 2016.

[12] Yi Liu, Shouqian Shi, Minghao Xie, Heiner Litz, and Chen Qian. Smash: Flexible, fast, and resource-efficient placement and lookup of distributed storage. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 7(2):1–22, 2023.

[13] Ye Yu, Jinpeng Liu, Xinan Liu, Yi Zhang, Eamonn Magner, Erik Lehnert, Chen Qian, and Jinze Liu. Seqothello: querying rna-seq experiments at scale. *Genome biology*, 19:1–13, 2018.

[14] The source codes and theoretical analysis related to visonembedder. https://github.com/VisonEmbedder/VisionEmbedder.

[15] Thomas Mueller Graf and Daniel Lemire. Xor filters: Faster and smaller than bloom and cuckoo filters. *Journal of Experimental Algorithmics (JEA)*, 25:1–16, 2020.

[16] Michael T Goodrich and Michael Mitzenmacher. Invertible bloom lookup tables. In *2011 49th Annual Allerton Conference on Communication, Control, and Computing (Allerton)*, pages 792–799. IEEE, 2011.

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

[18] Michael Mitzenmacher. The power of two choices in randomized load balancing. *IEEE Transactions on Parallel and Distributed Systems*, 12(10):1094–1104, 2001.

[19] Philippe Flajolet, Daniele Gardy, and Loÿs Thimonier. Birthday paradox, coupon collectors, caching algorithms and self-organizing search. *Discrete Applied Mathematics*, 39(3):207–229, 1992.

[20] Michael Molloy. The pure literal rule threshold and cores in random hypergraphs. 2004.

[21] Shouqian Shi and Chen Qian. Ludo hashing: Compact, fast, and dynamic key-value lookups for practical network systems. *POMACS.*, 4(2), 2020.

[22] Hassel library. https://bitbucket.org/peymank/hassel-public.

[23] Youtube comedy slam preference data data set. https://archive.ics.uci.edu/ml/datasets/YouTube+Comedy+Slam+Preference+Data.

[24] dblp: computer science bibliography. http://dblp.org/xml/release/dblp-2017-09-03.xml.gz.

[25] Murmur hashing source codes. https://github.com/aappleby/smhasher/blob/master/src/MurmurHash3.cpp.

[26] Djamal Belazzougui, Paolo Boldi, Rasmus Pagh, and et al. Monotone minimal perfect hashing: searching a sorted table with o (1) accesses. In *SODA'09*, 2009.

[27] Bohdan S Majewski, Nicholas C Wormald, George Havas, and et al. A family of perfect hashing methods. *The Computer Journal*, 39(6), 1996.

[28] Martin Dietzfelbinger, Anna Karlin, Kurt Mehlhorn, and et al. Dynamic perfect hashing: Upper and lower bounds. *SIAM Journal on Computing*, 23(4), 1994.

[29] Emmanuel Esposito, Thomas Mueller Graf, and Sebastiano Vigna. Recsplit: Minimal perfect hashing via recursive splitting. In *ALENEX '20*, 2019.

[30] Marco Genuzio, Giuseppe Ottaviano, and Sebastiano Vigna. Fast scalable construction of (minimal perfect hash) functions. In *SEA '16*, 2016.

[31] Yuhan Wu, Zirui Liu, Xiang Yu, Jie Gui, Haochen Gan, Yuhao Han, Tao Li, Ori Rottenstreich, and Tong Yang. Mapembed: Perfect hashing with high load factor and fast update. 2021.

[32] Haoyu Li, Liuhui Wang, Qizhi Chen, Jianan Ji, Yuhan Wu, Yikai Zhao, Tong Yang, and Aditya Akella. Chainedfilter: Combining membership filters by chain rule. *Proceedings of the ACM on Management of Data*, 1(4):1–27, 2023.

[33] Rocksdb. https://rocksdb.org/.

[34] Siying Dong, Andrew Kryczka, Yanqin Jin, and Michael Stumm. Evolution of development priorities in key-value stores serving large-scale applications: The {rocksdb} experience. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*, pages 33–49, 2021.

[35] Redis. http://redis.io/.

[36] Memcached. http://memcached.org/.

[37] Juncheng Yang, Yao Yue, and KV Rashmi. A large-scale analysis of hundreds of in-memory key-value cache clusters at twitter. *ACM Transactions on Storage (TOS)*, 17(3):1–35, 2021.

[38] Yuchen Ren, Jinyu Xie, Yunhui Qiu, Hankun Lv, Wenbo Yin, Lingli Wang, Bowei Yu, Hua Chen, Xianjun He, Zhijian Liao, et al. A low-latency multi-version key-value store using b-tree on an fpga-cpu platform. In *2019 29th International Conference on Field Programmable Logic and Applications (FPL)*, pages 321–325. IEEE, 2019.

[39] Baptiste Lepers, Oana Balmau, Karan Gupta, and Willy Zwaenepoel. Kvell: the design and implementation of a fast persistent key-value store. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 447–461, 2019.

[40] Jingyu Zhou, Meng Xu, Alexander Shraer, Bala Namasivayam, Alex Miller, Evan Tschannen, Steve Atherton, Andrew J Beamon, Rusty Sears, John Leach, et al. Foundationdb: A distributed unbundled transactional key value store. In *Proceedings of the 2021 International Conference on Management of Data*, pages 2653–2666, 2021.

[41] Hao Chen, Chaoyi Ruan, Cheng Li, Xiaosong Ma, and Yinlong Xu. {SpanDB}: A fast,{Cost-Effective}{LSM-tree} based {KV} store on hybrid storage. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*, pages 17–32, 2021.

[42] Badrish Chandramouli, Guna Prasaad, Donald Kossmann, Justin Levandoski, James Hunter, and Mike Barnett. Faster: A concurrent key-value store with in-place updates. In *Proceedings of the 2018 International Conference on Management of Data*, pages 275–290, 2018.

[43] Ting Yao, Yiwen Zhang, Jiguang Wan, Qiu Cui, Liu Tang, Hong Jiang, Changsheng Xie, and Xubin He. {MatrixKV}: Reducing write stalls and write amplification in {LSM-tree} based {KV} stores with matrix container in {NVM}. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 17–31, 2020.

[44] Junsu Im, Jinwook Bae, Chanwoo Chung, Sungjin Lee, et al. {PinK}: High-speed in-storage key-value store with bounded tails. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 173–187, 2020.

[45] Yijie Zhong, Zhirong Shen, Zixiang Yu, and Jiwu Shu. Redesigning high-performance lsm-based key-value stores with persistent cpu caches. In *2023 IEEE 39th International Conference on Data Engineering (ICDE)*, pages 1098–1111. IEEE, 2023.

[46] Huajun He, Ruiyuan Li, Sijie Ruan, Tianfu He, Jie Bao, Tianrui Li, and Yu Zheng. Trass: Efficient trajectory similarity search based on key-value data stores. In *2022 IEEE 38th International Conference on Data Engineering (ICDE)*, pages 2306–2318. IEEE, 2022.

[47] Fan Yang, Youmin Chen, Youyou Lu, Qing Wang, and Jiwu Shu. Aria: Tolerating skewed workloads in secure in-memory key-value stores. In *2021 IEEE 37th International Conference on Data Engineering (ICDE)*, pages 1020–1031. IEEE, 2021.

[48] Junkai Liang and Yunpeng Chai. Cruisedb: An lsm-tree key-value store with both better tail throughput and tail latency. In *2021 IEEE 37th International Conference on Data Engineering (ICDE)*, pages 1032–1043. IEEE, 2021.

[49] Xingsheng Zhao, Song Jiang, and Xingbo Wu. Wipdb: A write-in-place key-value store that mimics bucket sort. In *2021 IEEE 37th International Conference on Data Engineering (ICDE)*, pages 1404–1415. IEEE, 2021.