Practical Hardware Transactional vEB Trees

Mohammad Khalaji
mohammad.khalaji@uwaterloo.ca
University of Waterloo
Waterloo, Canada

Trevor Brown
trevor.brown@uwaterloo.ca
University of Waterloo
Waterloo, Canada

Khuzaima Daudjee
khuzaima.daudjee@uwaterloo.ca
University of Waterloo
Waterloo, Canada

Vitaly Aksenov
aksenov.vitaly@gmail.com
City, University of London
London, United Kingdom

Abstract

van Emde Boas (vEB) trees are sequential data structures optimized for extremely fast predecessor and successor queries. Such queries are an important incentive to use *ordered* sets or maps such as vEB trees. All operations in a vEB tree are doubly logarithmic in the universe size. Attempts to implement concurrent vEB trees have either simplified their structure in a way that eliminated their ability to perform fast predecessor and successor queries, or have otherwise compromised on doubly logarithmic complexity. In this work, we leverage *Hardware Transactional Memory* (HTM) to implement vEB tree-based sets and maps in which operations are doubly logarithmic in the absence of contention. Our proposed concurrent vEB tree is the first to implement *recursive summaries*, the key algorithmic component of fast predecessor and successor operations. Through extensive experiments, we demonstrate that our algorithm outperforms state-of-the-art concurrent maps by an average of 5× in a moderately skewed workload, and the single-threaded C++ standard ordered map and its unordered map by 70% and 14%, respectively. And, it does so while using two orders of magnitude less memory than traditional vEB trees.

CCS Concepts: • **Computing methodologies** → **Concurrent algorithms**; **Shared memory algorithms**; • **Information systems** → *Data scans*; • **Theory of computation** → **Concurrent algorithms**; **Predecessor queries**.

Keywords: Concurrent Data Structures, vEB Trees, van Emde Boas Trees, Transactional Memory, Lock Elision

PPoPP '24, March 2–6, 2024, Edinburgh, United Kingdom

ACM ISBN 979-8-4007-0435-2/24/03…$15.00
https://doi.org/10.1145/3627535.3638504

ACM Reference Format:
Mohammad Khalaji, Trevor Brown, Khuzaima Daudjee, and Vitaly Aksenov. 2024. Practical Hardware Transactional vEB Trees. In *The 29th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming (PPoPP '24), March 2–6, 2024, Edinburgh, United Kingdom.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3627535.3638504

1 Introduction

There has been significant recent interest in improving the implementations of ordered sets and maps, which are two of the most important abstract data types in computer science. Ordered sets store only keys, while ordered maps (dictionaries) associate each key with a value. These structures typically provide *insert*, *delete*, and *search* operations. Crucially, they also offer *successor* and *predecessor* queries that are popular, e.g., for data management [4, 48]. These two operations are the key difference between ordered and unordered sets and maps, and they are the primary reason to consider using an ordered set or map.

Most of the concurrent ordered sets and maps that have been proposed offer operations with logarithmic runtimes (ignoring contention). Binary search trees, B-tree variants, and skip lists fall into this well-explored category.

A few concurrent maps have been proposed with doubly logarithmic runtimes. Most notably, interpolation search trees offer amortized expected $O(\lg \lg n)$ complexity, but only for a limited set of distributions. Skip tries, on the other hand, offer $O(\lg \lg u)$ complexity for arbitrary distributions, where u is the size of the universe of keys, but they are complex, and there are no publicly available implementations.

van Emde Boas (vEB) trees are alternative doubly logarithmic data structures well known in the sequential setting for their extremely fast predecessor and successor operations. vEB trees are recursive data structures, where the root for a universe of size u has $\sqrt{u}$ children (often called *clusters*), each of which is a vEB tree of universe size $\sqrt{u}$. Additionally, the root has a special *summary* vEB tree of universe size $\sqrt{u}$. (These smaller vEB trees also have summaries and clusters, and so on, recursively.) Unfortunately, the standard vEB tree requires $O(u)$ space which renders it practically unusable for large universes.

This work addresses this key pitfall of vEB trees. Our vEB tree implementations use orders of magnitude less memory than traditional vEB trees. This is achieved by (1) dynamically allocating nodes on-demand, and (2) tuning the base case where the recursion stops.

Prior attempts to implement vEB trees in the concurrent setting have substantially compromised their capabilities in order to avoid the difficulties of producing a full-featured concurrent implementation. Such implementations fail to guarantee $O(\lg \lg u)$ complexity for their operations (even in the absence of contention) and/or do not guarantee correct results for successor/predecessor queries [25, 30, 31]. Some data structures have borrowed ideas from vEB trees but do not offer doubly logarithmic complexity for their operations as well [42].

Our contributions in this paper are efficient concurrent implementations of vEB tree-based sets and maps using Intel's *Hardware Transactional Memory* (HTM) instructions. Importantly, our concurrent vEB set and map implementations guarantee the classical time complexity of $O(\lg \lg u)$ for the standard operations including successor and predecessor queries, in the absence of contention. Our implementations use *Transactional Lock Elision* (TLE) to synchronize threads and to guarantee that operations are linearizable. We start by giving a naive concurrent implementation of a vEB tree using TLE, and iteratively improve it with a sequence of optimizations.

It is remarkable that TLE can be used to produce a fast concurrent implementation of a data structure with such extensive recursive substructure. Implementing vEB trees with conventional locking or lock-free techniques would be exceptionally difficult. Using HTM in a straightforward way to implement this sophisticated data structure results in a surprisingly performant algorithm. The resulting synchronization overhead is so low that our best implementation exceeds the single threaded performance of the C++ standard library's ordered map by up to 70% and its *unordered map* by up to 14%. Moreover, extensive experiments using both uniform and Zipfian workloads show that our vEB map implementation is up to about 5× and 3× faster than state-of-the-art concurrent binary search tree and (a,b)-tree implementations, respectively.

The contributions of this paper are as follows: (1) To the best of our knowledge, we present the first concurrent vEB tree that implements recursive summaries, which are crucial for fast successor and predecessor operations. (2) In the absence of contention, the insert, delete, search, successor, and predecessor operations complete in $O(\lg \lg u)$ steps. Moreover, our algorithm is simpler than prior work that offers similar runtimes. (3) We offer up to two orders of magnitude improvement in memory usage compared to traditional vEB sets, addressing their key drawback.

2 Background

In this section, we provide a brief overview of the van Emde Boas (vEB) tree together with Hardware Transactional Memory (HTM) and Transactional Lock Elision (TLE).

2.1 van Emde Boas Trees

Most well-known data structures for maintaining a dynamic ordered set of integers are based on binary search trees or B-trees. These data structures usually offer operations: search, insert, delete, ***predecessor***, and ***successor***. In a set containing n keys, these operations typically have a runtime complexity of $O(\lg n)$. However, the van Emde Boas (vEB) tree is different in that the complexity is defined with respect to the *universe size*, similar to tries. The universe is the set of integers $\{0, \ldots, u-1\}$, the members of which can be inserted into the data structure. For simplicity, the universe size is often assumed to be a power of two ($u = 2^{\lg u}$). A sequential vEB tree implements all aforementioned operations with a runtime complexity of $O(\lg \lg u)$.

The vEB tree is a recursive data structure [43, 44]. Each sub-tree of a vEB tree with universe size u is a vEB tree itself with a smaller universe size, specifically $\sqrt{u}$. (And sub-trees one level deeper have universe size $\sqrt{\sqrt{u}}$.) The recursion bottoms out at a *base case* of $u = 2$, where the vEB tree is simply a bit vector of size 2 without any children. Consequently, there are $\lg \lg u$ levels in a vEB tree. This recursion results in integer universe sizes at every level when $\lg u$ is a power of two. However, this rarely is the case, with $u = 2^{48}$ as an example. In this case, there are $\sqrt[\uparrow]{u}$ sub-trees, where $\sqrt[\uparrow]{u} = 2^{\lceil \frac{\lg u}{2} \rceil}$, and each sub-tree has universe size $\sqrt[\downarrow]{u} = 2^{\lfloor \frac{\lg u}{2} \rfloor}$. In addition, the vEB tree (and each sub-tree, recursively) has a summary vEB tree. The summary for a vEB tree with universe size u has universe size $\sqrt[\downarrow]{u}$. [1]

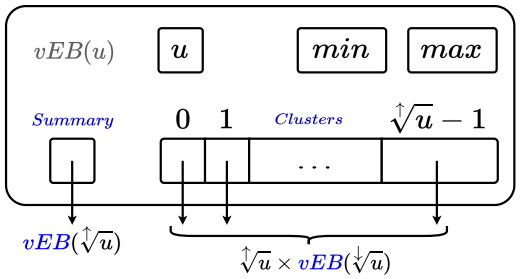


Figure 1. A vEB node with universe size u

Sequential vEB Tree Implementation. Figure 1 depicts the root of a vEB tree with universe size u [13]. At this node, the universe size is stored alongside *min* and *max* fields that keep track of the smallest and largest key currently contained in the sub-tree rooted at the node. The root contains an array of $\sqrt[\uparrow]{u}$ pointers to clusters, where each cluster is the root of a

[1] To be clear, each node in a vEB tree has a summary tree, and each summary tree is a vEB tree, so each node in a summary tree has its own smaller summary tree, and so on recursively.

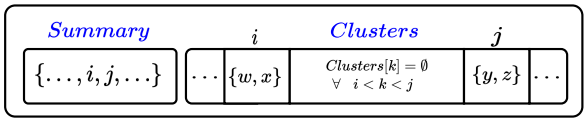


Figure 2. How data is stored in a vEB set

vEB tree with universe size $\sqrt[]{u}$. Additionally, the root has a pointer to a summary with universe size $\sqrt[]{u}$ that keeps track of non-empty clusters. The summary plays a crucial role in predecessor and successor queries in the vEB tree.

The minimum value in each cluster is stored directly in the *min* field of that cluster, and it is not stored in any of its sub-clusters. The same is *not* true for the maximum value. The *max* field essentially stores a duplicate key that exists solely to accelerate successor and predecessor operations, and the real key is stored in a sub-cluster.

An **insert** operation in a vEB tree with universe size u proceeds as follows. If the key to be inserted is smaller than the current *min* field, it becomes the new *min* key, and the old *min* key is recursively inserted in a sub-cluster. The particular sub-cluster to insert into is determined by the $\lceil \frac{\lg u}{2} \rceil$ most significant bits (*high* bits) of the key, and the $\lfloor \frac{\lg u}{2} \rfloor$ least significant bits (*low* bits) form the key that is recursively inserted into the sub-cluster. If that sub-cluster was previously empty, its index is inserted into the summary to reflect the fact that it is now non-empty. In the base case of the recursion, there are no clusters, and only two keys are possible, which occupy *min* and *max*. At the end of the recursive call, the *max* fields of all nodes visited are updated, as appropriate.

A **search** operation is similar to insert, except for the fact that it stops the recursion when it encounters an empty cluster, or finds the key it is searching for in the *min* or *max* fields of a node.

A **delete** operation is more complicated because if a thread deletes a *min* or *max* key, it must find a suitable replacement. Moreover, if it deletes the only key in a cluster, it must delete the cluster's index from the summary.

The **successor** operation requires additional discussion regarding how a vEB tree stores its elements. Take Figure 2 as an example: The set $\{\ldots, w, x, y, z, \ldots\}$ where $w < x < y < z$ is stored in a vEB tree. Out of the n elements present in this set, w and x are stored in the i-th cluster, while y and z are stored in the j-th cluster. As a result, since the i-th and j-th clusters are non-empty, $\{i, j\} \subseteq Summary$. Recall that each cluster is itself a vEB tree with universe size $\sqrt[]{u}$, and the summary is a vEB set too, with a universe size of $\sqrt[]{u}$.

In order for a thread to identify the successor of x, which is y, the first place to search is the cluster where x itself is stored. However, in this example, the thread will not find y in that cluster. Consequently, the thread needs to find the minimum key of the next *occupied* cluster to return as the answer. Without consulting the summary, this would be an expensive $O(\sqrt[]{u})$ operation. However, it can recursively retrieve the successor of i (which is j) from the summary with a runtime complexity of $O(\lg \lg \sqrt[]{u})$, and then return the minimum of the j-th cluster in an additional $O(1)$ steps. The **predecessor** operation is similar, but it uses the *max* field of nodes instead of *min*. We have provided pseudocodes of these operations in the supplementary material.

2.2 Hardware Transactional Memory (HTM)

Transactional Memory (*TM*) [27] is a concurrency control mechanism that aims to simplify the development of concurrent programs by providing an interface that allows programmers to define blocks of instructions as *transactions* that either commit and take effect atomically or abort and have no effect on shared memory. TM can be implemented in software [19, 23, 26, 39, 40] or in hardware [27].

Hardware transactions offer *atomicity* and *serializability*, and incur minimal overhead to the existing procedures and mechanisms in the processor. HTM was commercialized by IBM and Intel in the 2010s [29, 47]. There has been a significant amount of work on how to best utilize HTM in different setups and on the strengths and weaknesses of the mechanism [1, 5, 10, 17, 20, 21, 47]. The work in this paper is implemented on top of Intel's Transactional Synchronization Extensions (`TSX`).

Using the appropriate APIs, a transaction is started by invoking `_xbegin`, and it is committed by invoking `_xend`. It can also be explicitly aborted by invoking `_xabort`. Intel's best-effort implementation of HTM does not guarantee any transactions to commit, and the system can abort them at any time. Every transaction abort is accompanied by an abort reason provided by the hardware or the developer in the case of explicit aborts. Two of the most important abort reasons are *conflict* and *capacity*. Conflict aborts are the result of two threads accessing the same cache line, which means that false sharing is especially important. Capacity aborts, on the other hand, are caused by the exhaustion of shared resources within the HTM system, such as exceeding the L1 cache capacity or the associativity of a particular cache set.

2.3 Transactional Lock Elision (TLE)

HTM is a powerful mechanism that facilitates the implementation of correct concurrent data structures, offering more simplicity compared to traditional lock-based or lock-free methods. However, some performance pitfalls associated with it are easy to slip into. If transactions are aborted frequently, the naive workaround of repeating until commit can lead to significant performance degradation and progress concerns. Transactional Lock Elision (TLE) is a mechanism that aims to address this issue [16, 18]. The idea behind TLE is that a transaction that fails repeatedly will eventually acquire a global lock and execute alone in the system to ensure progress. The code path where the threads are making progress through hardware transactions is sometimes called

```
1  retriesLeft ← 35
2  while retriesLeft > 0
3      wait until globalLock is not held
4      if _xbegin() == SUCCESS
5          // control jumps to the else case on abort
6          if globalLock is held
7            _xabort(EXPLICIT)
8          result ← criticalSection(...)
9          _xend()
10         return result
11     else
12         // control jumps here on abort
13         retriesLeft ← retriesLeft - 1
14
15 // fallback path
16 acquire(globalLock)
17 result ← criticalSection(...)
18 release(globalLock)
19 return result
```

Algorithm 1: Typical TLE implementation

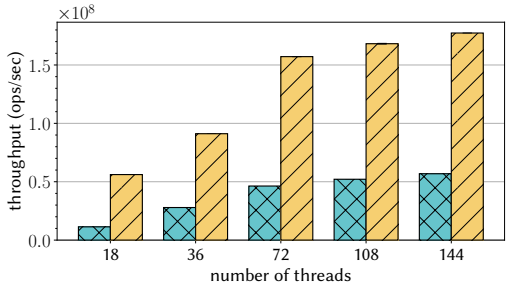
the *fast path*. Conversely, the code path where the global lock is held by one thread is usually called the *fallback* path, since it acts as a last resort for threads to make progress.

Algorithm 1 shows how TLE is typically implemented. The call to `_xbegin` at line 4 begins a transactional region. The thread running this code "subscribes" to the global lock at line 6 by reading its value, meaning that if the lock becomes acquired by another thread sometime in the future, the transaction will be aborted. If control reaches line 10, it means that the transaction has been committed, and any of its modifications to shared memory will be visible to others. On the other hand, if a transaction is aborted explicitly at line 7, or if it is aborted by hardware, then control jumps to the else block at line 12. In this else block, the thread can determine the reason for the abort and act accordingly. Each thread will attempt a transaction up to a fixed number of times, 35 in this example, ensuring the global lock is not held by any thread before each attempt. If none of those attempts succeed, the thread executes the fallback code path between the calls to *acquire* and *release* at lines 16 and 18, performing its task as the lone thread making progress.

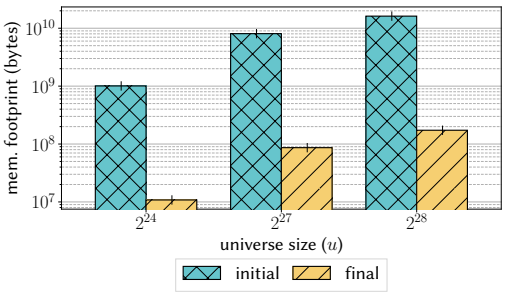
3 Our Algorithm

The high-level idea of our algorithm is to use a slightly modified sequential van Emde Boas tree implementation and wrap it in a Transactional Lock Elision (TLE) block. This entails implementing each operation by replacing the `criticalSection` in Algorithm 1 with a sequential implementation of the operation. However, the most straightforward way of doing this results in poor performance and memory efficiency. To address these issues, we propose a set of algorithmic improvements that are built on top of one another. Each algorithmic improvement is designed to address a specific problem with the previous algorithm.

Figure 3 depicts the speedup and memory savings of our final vEB set implementation compared to a naive implementation using TLE. This figure shows result of an experiment where up to 144 threads perform operations on the same tree



(a) Performance. Uniform 50% update workload. $u = 2^{28}$.



(b) Memory usage for $\frac{u}{2}$ keys. **Log-scaled** y-axis.

Figure 3. Comparing our initial and final vEB sets.

concurrently. The threads perform 50% searches and 50% updates on random keys following a uniform distribution for a fixed period of time, and total throughput (number of successful operations per second) is reported alongside the memory footprint of the tree (in bytes). (Further detail on the experimental methodology appears in Section 4.) The final implementation yields a 210% speedup (with 72 threads) and uses orders of magnitude less memory. The rest of this section is devoted to two goals, (1) describing the sequence of algorithmic improvements that led to this final implementation and (2) exploring the impact of converting our best-performing set to a map on performance and memory.

3.1 Naively TLE-Wrapped vEB

Memory allocation often aborts hardware transactions [15]. Traditional vEB trees pre-allocate all necessary memory when they are first constructed [13]. Thus, one straightforward way to build a HTM based vEB tree is to first pre-allocate all memory and construct a *skeleton* tree and then use TLE to perform operations on this tree. As depicted by bars labeled `naive` in Figure 6, this algorithm scales reasonably well as the number of threads increases. However, it suffers from a major issue.

Since the entire tree is pre-allocated at the beginning, the memory footprint of this implementation is proportional to the universe size without any regard for the actual number of keys present in the set. This issue is a substantial barrier to the incorporation of vEB trees in most real world applications. Even with cheaply accessible large memories, allocating the entire tree before using it is undesirable and

```
1 void allocateClusterIfNeeded(VEB vEB, long high)
2   if (vEB.clusters[high] == nil)
3     long clusterSize = ∜vEB.u;
4     vEB.clusters[high] = popFromThePool(clusterSize);
5     rememberToRefill(clusterSize);
```

Algorithm 2: Thread-local object pool usage

wasteful, especially for sparse sets. Consequently, our first algorithmic improvement addresses this issue.

3.2 Dynamically Allocated vEB

To this end, we propose a dynamically allocated vEB tree. The idea is to allocate, on demand, only nodes that are needed. With this approach, the memory footprint of the set is no longer directly proportional to u (although it is at least $\Omega(\sqrt{u})$ since the root node is that large), and sparser sets can use significantly less memory than dense sets.

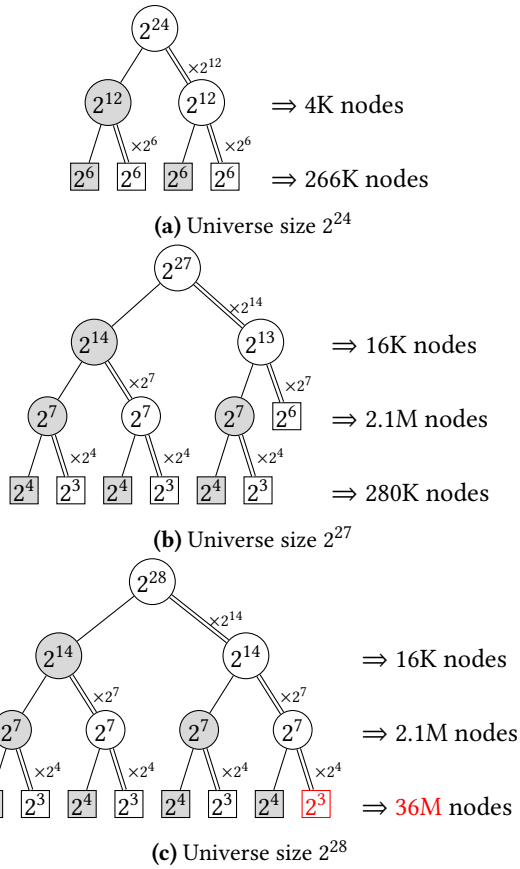
However, as mentioned earlier, memory allocation often aborts transactions, which makes it difficult to allocate memory only as needed. To overcome this challenge, we use a simple memory pooling mechanism.

In our implementation, there are thread-local pools of $N \geq 2$ nodes per *size class*, where N is configurable. The pool elements are stored in C++ standard vectors. The size classes are defined by the universe sizes at each level of recursion in the vEB tree. The reason behind a minimum pool size of 2 per size class is that, in an insert operation, a thread could possibly add a new cluster and a new summary node, of the same size, to the structure in the same transaction. In our formative experiments, we did not observe any meaningful differences in performance between $N = 2$, $N = 5$, and $N = 10$.

When inserting a key, the inserting thread needs access to the corresponding cluster. Recall that the index of that cluster is determined by taking the most significant $\lceil \frac{\lg u}{2} \rceil$ bits of the key. Before accessing the cluster, it checks if the cluster is already allocated. If not, it retrieves a new cluster from the thread-local pool and attaches it to the node (Algorithm 2). The same procedure is followed when the thread needs access to the summary node (when inserting the very first key to a cluster). After the insert operation is completed, either transactionally or through the fallback path, we replace any nodes that were removed from the thread-local pools by retrieving their size classes that were recorded by the `rememberToRefill` procedure. The refilling is performed outside transaction boundaries to prevent transaction aborts caused by memory allocation.

In the case of deleting a key, the thread needs to determine whether it is deleting the only key in the node. If so, it should return the node to the pool for later reuse. To limit the amount of memory used by a pool, we set an upper bound on the number of nodes that it can contain.

As we demonstrate later, this dynamically allocated vEB set consumes considerably less memory compared to the naive implementation, especially in sparsely filled sets, where empty nodes and sub-trees are not fully allocated. In our figures, we refer to this variation of our vEB sets as `dynamic`.



(a) Universe size 2^{24}

(b) Universe size 2^{27}

(c) Universe size 2^{28}

Figure 4. Cut-off vEB tree structure for three different universe sizes. Gray nodes are summary nodes. Square-shaped nodes are leaves. Only one cluster is represented per node, while the number of clusters is denoted by edge labels. For example, on Figure (a) the root node contains 2^{12} cluster nodes of size 2^{12}.

3.3 Cut-off vEB

Dynamically allocating vEB nodes on-demand led to considerable memory savings. However, as Figure 6 shows, it did not have a substantial impact on throughput. We observed that recursing until $u = 2$ is inefficient on modern hardware when loading a single word necessitates loading an entire cache line. A more efficient vEB tree would tune the base case of the recursion to use the processor cache more efficiently. Our next optimization does exactly this, terminating recursion once a node with universe size $u \leq 64$ is encountered. Different options for this threshold are discussed in the supplementary material.

Leaf nodes with $u \leq 64$ use a bitmap for storing keys, and they have a min and max field to comply with the original

(a) Before **(b)** After

Figure 5. Modifying the way the root node is divided

vEB semantics. Figure 6 shows the effects of this improvement on throughput (referred to as `cut-off`). In a vEB set with universe size $u = 2^{27}$, cutting off at the described level results in a 3× throughput compared to the dynamically allocated naive vEB set. This improvement also has a substantial positive impact on the memory footprint of the set, which is discussed further next.

3.4 Modifying The Root

Cutting off the recursion at a certain level of the tree ($u \leq 64$ in our case) yields significant improvements in both throughput and memory footprint. However, there is an unexpected performance drop induced by doubling the universe size from 2^{27} to 2^{28} (compare the `cut-off` bars in Figures 6b and 6c).

To get to the bottom of this unexpected performance drop, we need to take a closer look at how a vEB tree is structured, and how it is being affected by our implementation of the cut-off. Figure 4 compares the structure of a cut-off vEB tree with universe sizes 2^{24}, 2^{27}, and 2^{28}.

The reason for the performance drop boils down to the fact that the universe size $u = 2^{28}$ creates a relatively large number of leaf nodes (see Figure 4c), where the universe size of those leaves is small compared to those of vEB trees with $u = 2^{24}$ or $u = 2^{27}$ (Figures 4a and 4b). There are 36 million leaf nodes with universe size $u = 2^3$ or $u = 2^4$ in a vEB tree with $u = 2^{28}$, but the number of such nodes in a vEB tree with $u = 2^{27}$ is not nearly that large, and leaf nodes that small do not even exist in a vEB tree with $u = 2^{24}$.

To address the performance drop caused by the overwhelming number of small nodes, we first attempted to modify the internal nodes on one level above the leaves in a way that forced the clusters to be of size 64. For example, a vEB node with universe size 2^7 was divided to 2^1 clusters of universe size 64 (rather than 2^4 clusters of universe size 2^3). Although this approach yielded some improvement (reported in the supplementary material), it did not fully resolve the issue, which led us to a different approach.

While the aforementioned approach tackled the issue at the leaf level, our new approach attempts to resolve the problem by modifying how the root node is structured. Given that the size 2^{24} does not have the overhead of leaf nodes smaller than 64 (see Figure 4a), the idea is to divide the root node into $\frac{u}{2^{24}}$ clusters of size 2^{24}. More generally, the nodes are divided such that the number of clusters is a power of $2^6 = 64$.

Figure 5 depicts how the root modification is implemented for a root node of size 2^{28}. It is worth mentioning that if we stick with this method for universe sizes from 2^{25} to 2^{48}, the division derived for the root node with $u = 2^{48}$ is exactly the same as how the original vEB tree would be structured ($\frac{2^{48}}{2^{24}} = 2^{24}$ clusters of size 2^{24}).

Figure 6c shows how modifying the root (`new-root` in the legend) this way has a positive impact on the performance of sets with the universe size of 2^{28} (75% more throughput with 144 threads). The performance implications of root modification on sets with universe sizes that do not induce an overwhelming number of small nodes are sometimes slightly negative (see Figures 6a and 6b), but it is a small price to pay for a scalable vEB structure with larger universe sizes.

3.5 Correctness

We briefly sketch why our TLE-based data structure is correct. In all variants of our vEB sets, each operation is performed in either a hardware transaction or in the fallback path. The changes made by an operation running in a hardware transaction take effect atomically when the transaction commits. In the event that a transaction aborts, its changes are discarded and are never visible to other threads. An operation running on the fallback path does not execute concurrently with any other operation. To see why, note that an operation on the fallback path acquires the global lock, at which point all concurrent hardware transactions are automatically aborted by the hardware, because the lock state has changed (see Line 3 of Algorithm 1). Therefore, operations running in the fallback path are serialized. At any given time, one can have either a single transaction running on the fallback path (alone), or potentially many transactions running in hardware. In both cases, all vEB operations appear to take effect atomically, either because one operation is running alone in the system, or because multiple operations are running in hardware transactions (and atomicity is provided directly by the hardware). The result is a (family of) linearizable vEB set(s).

3.6 Adding Values To Our Set

So far, we have demonstrated the performance effects of our algorithmic improvements on sets, i.e., data structures that support only keys. Since we are using HTM, it is not difficult to modify our vEB set to a vEB map so that we can leverage our optimizations in the design of a map as well.

To achieve a vEB map, two simple modifications need to be made. Recall that the minimum key at each node is stored directly in the *min* field of the node, as opposed to being recursively inserted into its corresponding cluster. (Note that *max* is stored in the cluster instead.) First, we need to add a single value to every internal (non-leaf) node to store the value associated with the minimum value. Second, we

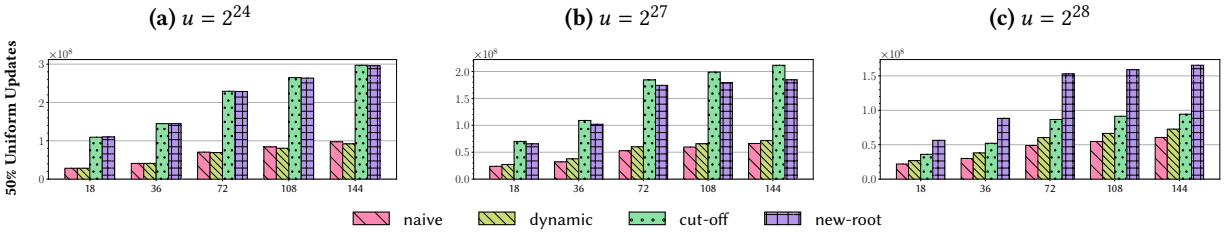


Figure 6. Throughput comparison of our vEB sets. $\frac{u}{2^{10}}$ keys are inserted before measurements. x-axes: number of threads.

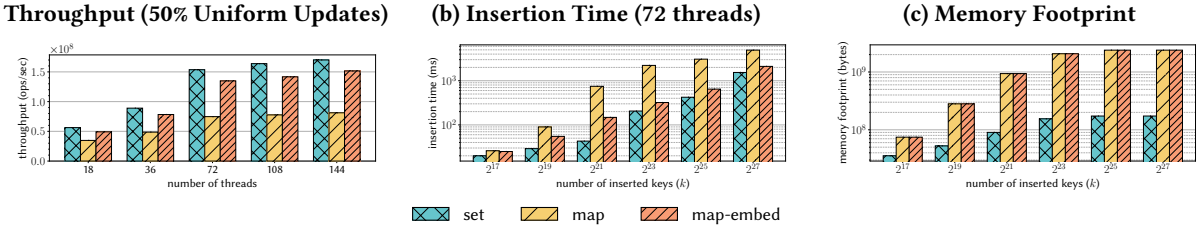


Figure 7. From vEB set to vEB map, $u = 2^{28}$. y-axes for insertion time and memory footprint are **log-scaled**.

need to add an array of values to each leaf. The size of the array will be equal to the universe size of the leaf. Note that summaries in the vEB map are still implemented as sets, which means that these modifications need not be applied to summary nodes.

Adding values to our vEB set induces some last level cache (LLC) misses that diminish performance, but there is a possible improvement that can alleviate this issue. The trivial way of adding an array of values to each leaf node is to add a pointer, which points to the values array, to the node. Although this approach keeps the node small, it causes the values array to be allocated in a different memory location than the node itself, which could lead to an extra cache miss when accessing a value. Alternatively, the values array can be embedded in the leaf node itself, but C/C++ do not allow variable-sized structs or classes. To circumvent this issue, we defined a node class that contains all fields *except* for the values array. Rather than allocating an instance of this class, we allocated a raw block of memory (i.e., `char *`) large enough to contain the class and the values array, and casted it to the class type. We then accessed the values array using macros to perform direct accesses, as appropriate, to the trailing memory after the end of the class definition that semantically represents the values array. This reduced LLC misses and improved performance significantly, as results in the supplementary material demonstrate.

Figure 7 depicts how taking the best version of our vEB set and making the aforementioned modifications to it affects performance, insertion time, and memory footprint of a data structure with $u = 2^{28}$. It is shown in Figure 7a that embedding the values array in the node has a significant positive impact on the throughput of the map (with $\frac{u}{2^{10}}$ keys pre-inserted). Moreover, embedding the values array in the node improves the time needed to insert different quantities of uniformly random keys in an insert-only workload (Figure 7b). For example, it takes 72 threads 4.9 seconds to insert 2^{27} key-value pairs into a vEB map with a universe size of 2^{28} if the values array is not embedded in the leaf itself. On the other hand, embedding the values array results in the same operation taking 2.1 seconds. We use this best performing version of vEB map when evaluating against other concurrent dictionaries in the next section.

4 Evaluation

Our experiments were executed using the publicly available benchmark, *SetBench* [11]. The experiments are run on a 4-socket machine with Intel's 2.2 GHz Xeon Gold 5220 processors. Each socket has 18 cores, and there are 2 hyperthreads per core, adding up to 144 threads. Threads in our experiments are pinned in a way that the first 72 threads are divided between sockets 2, 3, 4, and 1, in that order (18 threads per socket)s. The next 72 threads use hyperthreads on the same sockets in the same order. We begin with the second socket to delay the negative interactions between HTM and system processes on the first socket as much as possible. Moreover, we delay the utilization of hyperthreads because hyperthreading induces L1 cache sharing, which diminishes the cache capacity available for each transaction thereby increasing the rate of aborts. The implementations were compiled using `G++` version 9.3.0 and `-O3` optimization level. `jemalloc` was used for fast memory allocation, and memory pages were interleaved across the working sockets using `numactl`.

We compare our best-performing vEB map structure (see end of Section 3) with some of the most prominent concurrent maps available:

- **Elim-ABT** [41]: A concurrent (a,b)-tree designed for highly skewed write-heavy workloads.
- **C-IST** [11]: A doubly logarithmic interpolation search tree. The branching factor (fanout) in this tree is $\sqrt{u}$,

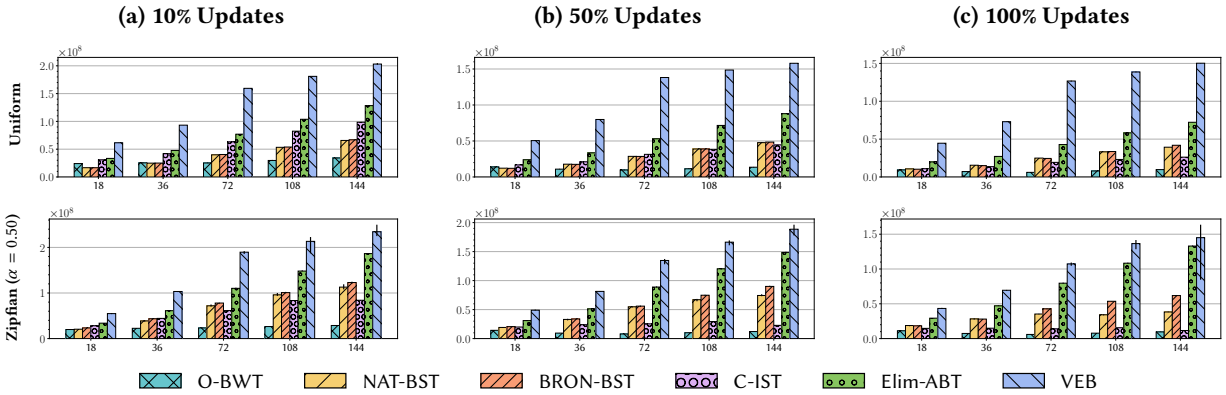


Figure 8. Performance. $u = 2^{28}$. $\frac{u}{2^{10}}$ keys inserted before the measurements. x-axes: number of threads. y-axes: throughput.

which makes it the most structurally similar data structure to our vEB map.

- **NAT-BST** [34]: A lock-free binary search tree.
- **BRON-BST** [7]: An optimistic concurrency control binary search tree.
- **O-BWT** [45]: The open source implementation of Microsoft's BW-Tree [32].

We also compared our vEB map to the concurrent BST by David et al. and the concurrent (a,b)-tree by Brown [12, 14], but we do not report on the results here since our implementation performed significantly better than these comparison trees on all workloads.

In our comparisons, we fix the universe size to $u = 2^{28}$, and the number of inserted keys prior to beginning the measurements to $\frac{u}{2^{10}}$, while experimental results with $\frac{u}{2}$ and $\frac{u}{2^{13}}$ prefilling are included in the supplementary material.[2] The operations performed on all of the data structures are searches, inserts, and deletes, except for the experiments in Section 4.6, where we evaluate the performance of successor queries. We have read-heavy workloads in our experiments where 90% of the operations are searches, as well as workloads with a high percentage of writes (50% and 100%). For random keys generated as operands, we use (1) uniform keys where each key is selected with equal probability ($\frac{1}{u}$), and (2) Zipfian keys with a skew factor of $\alpha = 0.50$ [24]. To save space, we sometimes omit uniform workloads in favour of Zipfian ones, because skewed workloads are more charitable to competing data structures. Experimental results with different α values are reported in the supplementary material.

[2]The choice of $\frac{u}{2^{10}}$ is somewhat arbitrary, but we note increasing the denominator in this fraction has two effects: clusters in our vEB sets will be more sparsely populated, and the total number of keys inserted will be smaller (relative to the universe size). In particular, we expect larger denominators would be charitable to other competing algorithms, as we expect our vEB set(s) to perform worse as sparsity increases.

4.1 Performance

In this section, we present the performance comparison of the vEB map with the competing data structures by measuring the throughput (operations executed per second) for our different workloads. Each data point on all graphs represents the average value over 5 independent runs with error bars around the means showing the minimum and maximum measured values.

Uniform Distribution. For a uniformly distributed workload, shown in the first row of Figure 8, our proposed vEB map performs substantially faster than its competitors. For update-only workloads executing with 144 threads, the vEB map is twice as fast as its closest competitor, Elim-ABT. In a read-heavy situation (90% searches, 10% inserts and deletes), our proposal offers a 1.58× speedup compared to Elim-ABT. Compared to C-IST, the most structurally similar data structure, our vEB map performs 2.06× faster in the read-heavy workload, and the advantage increases to 5.73× in the update-only setting. O-BWT, a popular data structure for databases, was the slowest data structure in most of our experiments.

Zipfian Distribution. Depicted in the second row of Figure 8, this set of experiments includes the same workloads in terms of update to read ratio but with a fairly skewed Zipfian key-generator (skew factor $\alpha = 0.50$). For workloads with 10% and 50% updates, the vEB map delivers about 25% speedup compared to Elim-ABT with 144 threads, with the advantage being bigger when there is less contention (10% updates). For write-only workloads, the vEB map is 9% faster than Elim-ABT. Decreasing α results in a less skewed distribution, leading to an increase in our vEB map's advantage. As α increases beyond 0.50, this performance gap narrows as expected due to the following reasons. First, by increasing the skew factor, we introduce a small set of *hot* keys that are frequently written to. As a result, HTM transactions begin to abort more often and the fallback path is invoked more frequently, which means more threads are serialized. Second, Elim-ABT is specifically designed to perform well under highly contended write-heavy workloads while our

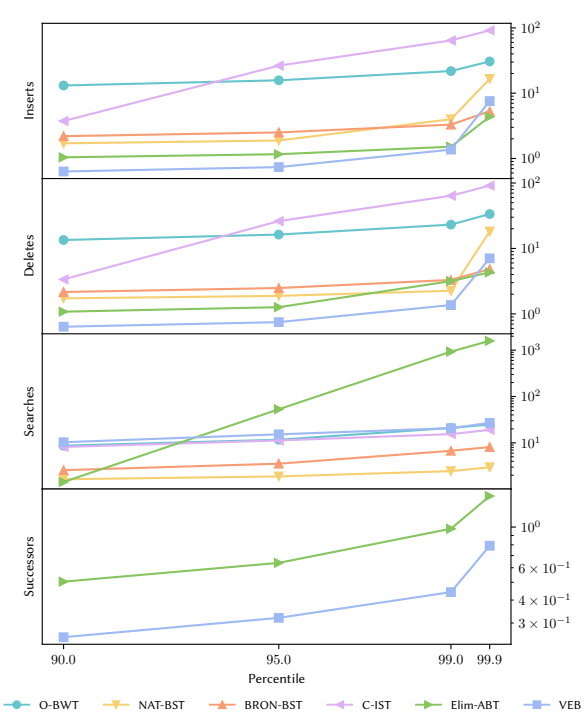


Figure 9. Per-operation latency in microseconds. **Log-scaled** y-axis. 72 Threads running a 50% updates workload with a Zipfian distribution ($\alpha = 0.50$), except for the lowermost graph, which is explained in Section 4.6.

vEB map shows better performance in moderately skewed workloads despite the fact that it is not specifically engineered for skewed distributions. Regarding the other data structures, our vEB map is 90% faster in the worst case (versus BRON-BST in the read-heavy workload), and 15× faster in the best case (versus O-BWT in the write-only workload). Experimental results with a more diverse set of skew factors are included in the supplementary material. Briefly, our vEB map outperforms other data structures in all workloads where $\alpha = 0.60$. In addition, it is still faster in 10% and 50% update workloads where $\alpha = 0.70$. In extreme cases where $\alpha = 0.80$ or $\alpha = 0.99$, however, it cannot compete with Elim-ABT in 50% and 100% updates workloads, but it is faster than other competing data structures except for NAT-BST in some workloads.

Operation Latencies. We measured operation latencies in a 50% update workload with a Zipfian distribution ($\alpha = 0.50$). Results appear in the top three charts of Figure 9, where it can be observed that the latency of search operations is relatively low in BST structures, while inserts and deletes have a higher latency compared to Elim-ABT and vEB. This observation shows how the well-studied cache issues in concurrent binary search trees affect their overall performance by slowing down updates.

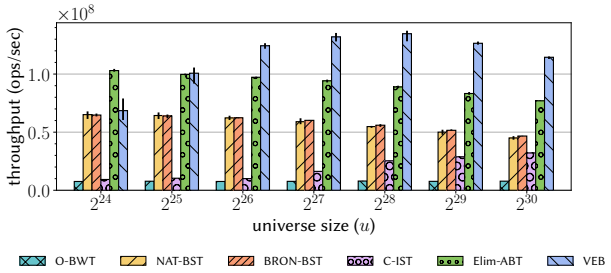


Figure 10. The effect of universe size on performance. $\frac{u}{2^{10}}$ keys pre-inserted. 50% updates workload with a Zipfian distribution ($\alpha = 0.50$).

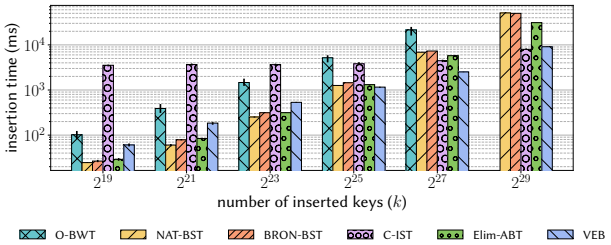


Figure 11. Insertion time (ms). **Log-scaled** y-axis.

Performance Limitations. In this section, we briefly discuss some limitations to our vEB map's performance.

Firstly, the vEB map's performance struggles with heavily skewed workloads. Heavily skewed workloads naturally induce more memory contention than uniform workloads (since some data is hot). HTM and TLE are known to suffer under high contention, as increased contention leads to more aborts and more transactions running on the fallback path. A data structure like Elim-ABT is specifically designed to perform well under highly contended write-heavy workloads, and it outperforms our vEB map in such cases.

Secondly, our vEB map thrives with large, densely populated universes, but struggles to compete with other data structures when the universe size is small. Figure 10 shows the performance of our vEB map compared with other data structures with different universe sizes. In these experiments, we have fixed the thread count to 72, and we have inserted $\frac{u}{2^{10}}$ keys to the data structures prior to the measurements. The workload consists of 25% inserts and 25% deletes, as well as 50% searches (lookups) of keys that follow a Zipfian distribution with $\alpha = 0.50$. As is evident from the figure, the vEB map is outperformed by Elim-ABT when the universe size equals 2^{24}, even when the workload is not heavily skewed.

4.2 Insertion Time

We fix the number of threads to 72 and measure the time it takes to insert k key-value pairs into structures with a key range (universe size) of $u = 2^{30}$, where $k = 2^{19}, 2^{21}, \ldots, 2^{29}$. The keys are generated using a uniform distribution and are inserted in random order.

As Figure 11 shows, in sparser settings, different variants of concurrent binary search trees (NAT-BST and BRON-BST)

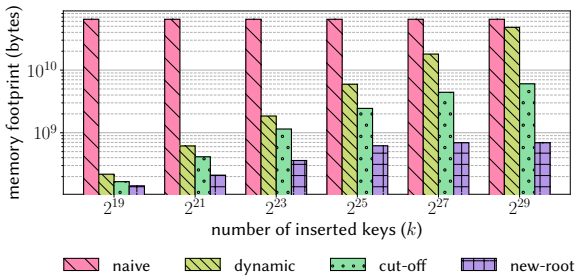


Figure 12. Memory footprint (bytes). **Log-scaled** y-axis. $u = 2^{30}$.

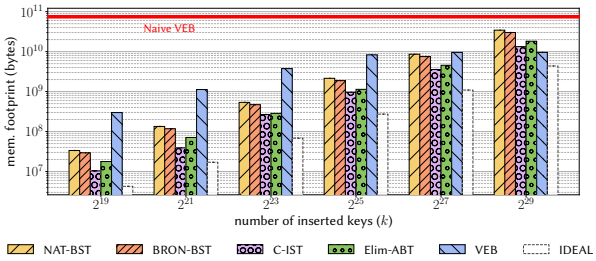


Figure 13. Memory footprint (bytes). **Log-scaled** y-axis. $u = 2^{30}$.

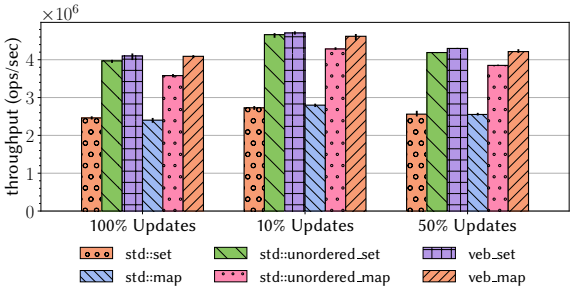


Figure 14. Single-threaded performance compared with C++ standard containers. Zipfian ($\alpha = 0.50$) workloads.

perform better than the other algorithms. However, contrary to our expectations, O-BWT performs poorly, and we were unable to obtain the data point for $k = 2^{29}$ because of segmentation faults in its open-source implementation. Due to its target for databases, good performance for bulk inserts is expected. Nevertheless, its performance might be better with sequential, ordered inserts rather than randomly ordered ones. As expected, our vEB map performs best when the universe is densely populated ($k = 2^{25}$ and higher).

4.3 Memory Footprint of vEB Sets

In this section, we evaluate the impact of our algorithmic improvements on the memory footprint of vEB sets. The experimental setup is identical to that of Section 4.2, but instead of key-value pairs, only keys are inserted since we are comparing sets rather than maps. The results are shown in Figure 12.

The first observation is that the memory footprint of the naive implementation of vEB sets remains constant regardless of the data structure's density. Mere dynamic allocation of nodes impacts the memory footprint substantially, with the sparsest data point being 99.6% more memory efficient than that of the naive implementation. Additionally, cutting off the recursion at $u = 64$ is also significantly impactful on memory efficiency. This technique results in a 75% improvement over its predecessor (dynamically allocated vEB sets without cut-off) when $k = 2^{29}$. The same argument could be made for the root modification technique, which is about 49% more efficient than its predecessor in the densest configuration. However, the root modification technique is not as impactful on the memory footprint in universe sizes with a relatively small number of small nodes, which is to be expected (per Figure 4).

4.4 Memory Footprint of vEB Map vs. Competitors

Having demonstrated the memory advantages of our algorithmic improvements on vEB sets, we now compare the memory footprint of our vEB map with other concurrent maps. The results are shown in Figure 13. The experiment is set up similarly to the last one. Recall that we used k to denote the number of inserted keys. Two extra data points are added for reference: **Naive VEB** depicts the memory footprint of a traditional (without any of our improvements) vEB map of universe size $u = 2^{30}$ [13], and **IDEAL** is the memory footprint of having a k-bit bitmap for the keys, and an array of k 8-byte values, which is a very optimistic implementation. Note that the IDEAL data points refer to a hypothetical implementation where there is no structure to the data and up to k key-value pairs can be stored. In such a configuration, efficient implementation of successor and predecessor queries is impossible. Note that the O-BWT implementation available to us did not have a mechanism for calculating the memory footprint, so results are unavailable for that data structure.

The vEB structure sacrifices memory for speed, but as Figure 13 depicts, our root-modified, cut-off vEB map uses 99.9% and 87.1% less memory compared to the naive vEB map in the sparsest and the densest settings respectively. Moreover, as the set of inserted keys becomes denser, the memory footprint size of our vEB map draws closer to the memory footprint of other competitors, even those that use memory proportional to n rather than u.

4.5 Single-threaded Performance

For this set of experiments, we compare the throughput of our vEB sets and maps with the standard ordered and unordered sets and maps available in the C++ programming language. Since those containers are not thread safe, the experiments are executed with a single thread. The universe size is fixed to 2^{24}, and 2^{16} keys (or key-value pairs for maps, respectively) are inserted into the data structures before the measurement begins. The keys generated for the operations follow a Zipfian distribution with $\alpha = 0.50$. Figure 14 shows

the results for workloads with 10%, 50%, and 100% updates. As in previous experiments, performance tends to decrease when the update rate increases.

For the update-only workload, our vEB set performs 66% better than the standard ordered set and is better than the standard unordered set (a hash table!) by 3%. The vEB map is also 70% faster than its ordered competitor, and it is 14% faster than the standard unordered map.

4.6 Successor Queries

The vEB tree offers $O(\lg \lg u)$ successor and predecessor queries, in the absence of contention. These queries make it attractive for many applications. In this section, we compare the performance of successor queries against Elim-ABT, the closest competitor in the previous experiments. Even though Elim-ABT is an ordered map, its publicly available implementation does not offer predecessor and successor operations. To make a comparison possible, we implemented a successor operation for Elim-ABT, but it is *not* linearizable. Thus, our implementation of the successor operation for Elim-ABT is unfairly advantaged by being extremely fast because it does not have to perform any consistency validations or retries for correctness. A correct implementation would be slower in practice because of necessary synchronization overheads.

To sanity check our successor and predecessor implementations, we implemented and executed some stress tests that inserted multiples of a constant number into the data structure: $c \cdot k \; \forall k$. We then ran workloads that queried the data structure for successors and predecessors of randomly generated keys, and compared the results to expected answers. We conducted these stress tests on both vEB maps and Elim-ABT maps. (The latter would produce incorrect results in the presence of updates, but there were no updates in this workload.)

Having completed our sanity checks, we evaluated performance using the following workload. We fixed the universe size to $u = 2^{28}$ and initially inserted 2^{23} uniformly distributed keys into the data structure. In this workload, the threads perform 98% successor queries, and 2% inserts and deletes. The keys generated in the workload follow a Zipfian distribution with $\alpha = 0.50$ to simulate moderate skew. Figure 15 compares the performance of our vEB's linearizable successor operation to the Elim-ABT's (unfair) non-linearizable successor operation. The vEB consistently outperforms the Elim-ABT, by up to 56%, although its advantage shrinks at the highest thread count. Note that **Elim-ABT achieves its high performance by sacrificing correctness**, which makes our vEB map's advantage more significant. The vEB map's advantage is also evident in the "Successors" plot in Figure 9, which reports operation latencies for the same workload as shown in Figure 2.

Although our implementation of the vEB map does not support range queries, it is worth mentioning that they can

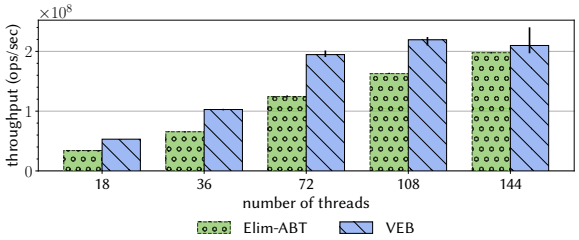


Figure 15. Successor operation performance. Elim-ABT is **non-linearizable**. The workload consists of 98% successor queries and 2% updates that follow a Zipfian distribution ($\alpha = 0.50$).

be implemented as a sequence of successor operations performed in a transaction. The performance of such a range query would depend on the length of the query (i.e., both the size of the range queried and how densely populated the range is). We expect small range queries to be fast, but large range queries to be prone to interruption by concurrent updates.

5 Related Work

The literature on concurrent sets and maps is rich. We limit our discussion to work that is most closely related to our proposal in this paper.

Binary Search Trees. A wide variety of concurrent binary search trees have been implemented and extensively evaluated [3]. The first provably correct concurrent BST was proposed by Ellen et al. [22]. They presented a lock-free unbalanced BST that uses *flagging* and *marking* to implement *helping* to guarantee lock-free progress. Natarajan and Mittal improved upon this work by *flagging* and *tagging edges* rather than nodes, increasing possible concurrency, alongside other improvements **(NAT-BST)** [34]. Bronson et al. proposed an optimistic lock based AVL tree **(BRON-BST)** [7]. David et al. presented a methodology for implementing optimistic lock based data structures, and implemented an unbalanced BST as an example of their methodology [14]. Many other implementations of concurrent BSTs have also appeared [9, 28, 35, 38].

B-Trees. Braginsky and Petrank were the first to propose concurrent B-Trees based on lock-free linked lists of key-value pairs [6]. Brown presented a lock-free (a,b)-tree, based on wait-free synchronization primitives `LLX` and `SCX` [8, 9, 12]. This (a,b)-tree was shown to be significantly faster than the concurrent BSTs mentioned above [9]. Srivastava and Brown proposed another concurrent (a,b)-tree **(Elim-ABT)**. It introduced a technique called *publishing elimination* in which, at a high level, each node has a publishing area where threads that are trying to insert and delete the same key meet up with one another and eliminate each other's operations. This technique makes it well-suited for highly skewed workloads [41]. The BW-Tree is a lock-free B+tree, and it is a popular data structure in the database community

[32]. We use the optimized version of BW-Tree proposed by Wang et al. **(O-BWT)** in our experiments [45].

Tries. Willard introduced x-fast-tries and y-fast-tries, data structures that support doubly logarithmic predecessor operations, similar to vEB trees [46]. However, they would be significantly harder to implement than our vEB map, since they require dynamic universal hashing or cuckoo hashing. As a result, to the best of our knowledge, concurrent implementations have not appeared, and their performance has not been well studied empirically. Moreover, the leaves of an x-fast-trie are binary search trees, which are known to utilize processor caches poorly compared to data structures with fat nodes (containing many keys).

Oshman and Shavit introduced skip tries, taking inspiration from x-fast-tries and y-fast-tries [36]. A skip trie is a concurrent variant of a y-fast-trie in which skip lists are used instead of binary search trees (resulting in amortized expected doubly logarithmic runtimes). A skip trie would be a natural competitor for our vEB map in our experiments, but we are not aware of any fast or stable publicly available implementation.

Distribution-aware data structures. There has been some study of distribution-aware data structures in the concurrent setting. Brown et al. proposed a concurrent interpolation search tree (**C-IST**) that offers doubly-logarithmic runtime and is robust to certain types of distributional skew, but as observed in our experiments, its performance degrades in update-heavy workloads [11]. Skip lists are probabilistic data structures that are robust to distributional skew [37] (although not distribution *aware* per se). Aksenov et al. proposed the Splay list, which is a concurrent skip list that continually moves frequently-accessed keys to make them more efficient to access [2].

vEB Trees. Kulakowski presented *cvEB*, a vEB-like concurrent data structure using locks [30]. However, it does not offer doubly logarithmic complexity, and the maximum and predecessor operations are not implemented in it. The same author also introduced *dcvEB*, the dynamic version of *cvEB* with some modifications and improvements [31]. The stated runtime complexity is doubly logarithmic, but there is no publicly available implementation of the algorithm for us to compare against. Moreover, as the paper states, *dcvEB* allows its successor operation to occasionally return *incorrect values*, which would make any performance comparisons unfair.

Guo and Suda proposed a lock-free concurrent vEB-like data structure, but they fix the branching factor at 64, and the summary structures are not recursive. Consequently, the presented data structure is neither doubly logarithmic nor can it offer fast successor and predecessor queries. The code for this data structure is not publicly available, so we could not include it in our experiments.

Mayr et al. proposed a vEB set designed for SIMD architectures [33]. They presented a method that optimizes the *construction* of vEB trees for GPUs, and they compared their work with binary search trees on graphical processors using Nvidia's CUDA.

6 Conclusion

In this paper, we introduced practical vEB tree-based sets and maps. We leveraged Hardware Transactional Memory to achieve simple, low overhead thread synchronization, and we used Transactional Lock Elision to guarantee progress. Our data structures show excellent single threaded performance, outperforming C++ standard ordered and even unordered sets and maps. In the concurrent setting, our vEB map is on average 5× faster than state-of-the-art concurrent binary search trees and (a,b)-trees. Our work's high performance is accompanied by orders of magnitude less memory consumption compared to traditional vEB structures.

To the best of our knowledge, our work is the first concurrent implementation of vEB trees that implements recursive summaries and, as a result, in the absence of contention, obtains doubly logarithmic runtime complexity for its predecessor and successor queries.

Acknowledgments

This work was supported by the Natural Sciences and Engineering Research Council of Canada (NSERC) Collaborative Research and Development grant: 539431-19, the Canada Foundation for Innovation John R. Evans Leaders Fund (38512) with equal support from the Ontario Research Fund CFI Leaders Opportunity Fund, NSERC Discovery Program Grant: 2019-04227, NSERC Discovery Launch Grant: 2019-00048, and the University of Waterloo.

References

[1] Yehuda Afek, Amir Levy, and Adam Morrison. 2014. Software-improved hardware lock elision. In *ACM Symposium on Principles of Distributed Computing, PODC '14, Paris, France, July 15-18, 2014*. ACM, 212–221. https://doi.org/10.1145/2611462.2611482

[2] Vitaly Aksenov, Dan Alistarh, Alexandra Drozdova, and Amirkeivan Mohtashami. 2020. The Splay-List: A Distribution-Adaptive Concurrent Skip-List. In *34th International Symposium on Distributed Computing, DISC 2020, October 12-16, 2020, Virtual Conference (LIPIcs, Vol. 179)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 3:1–3:18. https://doi.org/10.4230/LIPIcs.DISC.2020.3

[3] Maya Arbel-Raviv, Trevor Brown, and Adam Morrison. 2018. Getting to the Root of Concurrent Binary Search Tree Performance. In *2018 USENIX Annual Technical Conference, USENIX ATC 2018, Boston, MA, USA, July 11-13, 2018*. USENIX Association, 295–306. https://www.usenix.org/conference/atc18/presentation/arbel-raviv

[4] Nicole Bidoit, Melanie Herschel, and Katerina Tzompanaki. 2014. Query-Based Why-Not Provenance with NedExplain. In *Proceedings of the 17th International Conference on Extending Database Technology, EDBT 2014, Athens, Greece, March 24-28, 2014*. OpenProceedings.org, 145–156.

[5] Jayaram Bobba, Kevin E. Moore, Haris Volos, Luke Yen, Mark D. Hill, Michael M. Swift, and David A. Wood. 2008. Performance Pathologies

in Hardware Transactional Memory. *IEEE Micro* 28, 1 (2008), 32–41. https://doi.org/10.1109/MM.2008.11

[6] Anastasia Braginsky and Erez Petrank. 2012. A lock-free B+tree. In *24th ACM Symposium on Parallelism in Algorithms and Architectures, SPAA '12, Pittsburgh, PA, USA, June 25-27, 2012*. ACM, 58–67. https://doi.org/10.1145/2312005.2312016

[7] Nathan Grasso Bronson, Jared Casper, Hassan Chafi, and Kunle Olukotun. 2010. A practical concurrent binary search tree. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2010, Bangalore, India, January 9-14, 2010*. ACM, 257–268. https://doi.org/10.1145/1693453.1693488

[8] Trevor Brown, Faith Ellen, and Eric Ruppert. 2013. Pragmatic primitives for non-blocking data structures. In *ACM Symposium on Principles of Distributed Computing, PODC '13, Montreal, QC, Canada, July 22-24, 2013*. ACM, 13–22. https://doi.org/10.1145/2484239.2484273

[9] Trevor Brown, Faith Ellen, and Eric Ruppert. 2014. A general technique for non-blocking trees. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '14, Orlando, FL, USA, February 15-19, 2014*. ACM, 329–342. https://doi.org/10.1145/2555243.2555267

[10] Trevor Brown, Alex Kogan, Yossi Lev, and Victor Luchangco. 2016. Investigating the Performance of Hardware Transactions on a Multi-Socket Machine. In *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures, SPAA 2016, Asilomar State Beach/Pacific Grove, CA, USA, July 11-13, 2016*. ACM, 121–132. https://doi.org/10.1145/2935764.2935796

[11] Trevor Brown, Aleksandar Prokopec, and Dan Alistarh. 2020. Non-blocking interpolation search trees with doubly-logarithmic running time. In *PPoPP '20: 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, San Diego, California, USA, February 22-26, 2020*. ACM, 276–291. https://doi.org/10.1145/3332466.3374542

[12] Trevor Alexander Brown. 2017. *Techniques for Constructing Efficient Lock-free Data Structures*. Ph. D. Dissertation. University of Toronto, Canada. http://hdl.handle.net/1807/80693

[13] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2009. *Introduction to Algorithms, 3rd Edition*. MIT Press. http://mitpress.mit.edu/books/introduction-algorithms

[14] Tudor David, Rachid Guerraoui, and Vasileios Trigonakis. 2015. Asynchronized Concurrency: The Secret to Scaling Concurrent Search Data Structures. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2015, Istanbul, Turkey, March 14-18, 2015*. ACM, 631–644. https://doi.org/10.1145/2694344.2694359

[15] Dave Dice, Tim Harris, Alex Kogan, and Yossi Lev. 2015. The Influence of Malloc Placement on TSX Hardware Transactional Memory. *CoRR* abs/1504.04640 (2015). arXiv:1504.04640 http://arxiv.org/abs/1504.04640

[16] Dave Dice, Alex Kogan, and Yossi Lev. 2016. Refined transactional lock elision. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2016, Barcelona, Spain, March 12-16, 2016*. ACM, 19:1–19:12. https://doi.org/10.1145/2851141.2851162

[17] Dave Dice, Alex Kogan, Yossi Lev, Timothy Merrifield, and Mark Moir. 2014. Adaptive integration of hardware and software lock elision techniques. In *26th ACM Symposium on Parallelism in Algorithms and Architectures, SPAA '14, Prague, Czech Republic - June 23 - 25, 2014*. ACM, 188–197. https://doi.org/10.1145/2612669.2612696

[18] David Dice, Yossi Lev, Mark Moir, and Daniel Nussbaum. 2009. Early experience with a commercial hardware transactional memory implementation. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2009, Washington, DC, USA, March 7-11, 2009*. ACM, 157–168. https://doi.org/10.1145/1508244.1508263

[19] David Dice, Ori Shalev, and Nir Shavit. 2006. Transactional Locking II. In *Distributed Computing, 20th International Symposium, DISC 2006, Stockholm, Sweden, September 18-20, 2006, Proceedings (Lecture Notes in Computer Science, Vol. 4167)*. Springer, 194–208. https://doi.org/10.1007/11864219_14

[20] Nuno Diegues and Paolo Romano. 2014. Self-Tuning Intel Transactional Synchronization Extensions. In *11th International Conference on Autonomic Computing, ICAC '14, Philadelphia, PA, USA, June 18-20, 2014*. USENIX Association, 209–219. https://www.usenix.org/conference/icac14/technical-sessions/presentation/diegues

[21] Nuno Diegues, Paolo Romano, and Luís E. T. Rodrigues. 2014. Virtues and limitations of commodity hardware transactional memory. In *International Conference on Parallel Architectures and Compilation, PACT '14, Edmonton, AB, Canada, August 24-27, 2014*. ACM, 3–14. https://doi.org/10.1145/2628071.2628080

[22] Faith Ellen, Panagiota Fatourou, Eric Ruppert, and Franck van Breugel. 2010. Non-blocking binary search trees. In *Proceedings of the 29th Annual ACM Symposium on Principles of Distributed Computing, PODC 2010, Zurich, Switzerland, July 25-28, 2010*. ACM, 131–140. https://doi.org/10.1145/1835698.1835736

[23] Pascal Felber, Christof Fetzer, Patrick Marlier, and Torvald Riegel. 2010. Time-Based Software Transactional Memory. *IEEE Trans. Parallel Distributed Syst.* 21, 12 (2010), 1793–1807. https://doi.org/10.1109/TPDS.2010.49

[24] Jim Gray, Prakash Sundaresan, Susanne Englert, Kenneth Baclawski, and Peter J. Weinberger. 1994. Quickly Generating Billion-Record Synthetic Databases. In *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data, Minneapolis, Minnesota, USA, May 24-27, 1994*. ACM Press, 243–252. https://doi.org/10.1145/191839.191886

[25] Ziyuan Guo and Reiji Suda. 2019. Lock-free concurrent van Emde Boas Array. (2019).

[26] Maurice Herlihy, Victor Luchangco, Mark Moir, and William N. Scherer III. 2003. Software transactional memory for dynamic-sized data structures. In *Proceedings of the Twenty-Second ACM Symposium on Principles of Distributed Computing, PODC 2003, Boston, Massachusetts, USA, July 13-16, 2003*. ACM, 92–101. https://doi.org/10.1145/872035.872048

[27] Maurice Herlihy and J. Eliot B. Moss. 1993. Transactional Memory: Architectural Support for Lock-Free Data Structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture, San Diego, CA, USA, May 1993*. ACM, 289–300. https://doi.org/10.1145/165123.165164

[28] Shane V. Howley and Jeremy Jones. 2012. A non-blocking internal binary search tree. In *24th ACM Symposium on Parallelism in Algorithms and Architectures, SPAA '12, Pittsburgh, PA, USA, June 25-27, 2012*. ACM, 161–171. https://doi.org/10.1145/2312005.2312036

[29] Christian Jacobi, Timothy J. Slegel, and Dan F. Greiner. 2012. Transactional Memory Architecture and Implementation for IBM System Z. In *45th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2012, Vancouver, BC, Canada, December 1-5, 2012*. IEEE Computer Society, 25–36. https://doi.org/10.1109/MICRO.2012.12

[30] Konrad Kulakowski. 2014. A concurrent van Emde Boas array as a fast and simple concurrent dynamic set alternative. *Concurr. Comput. Pract. Exp.* 26, 2 (2014), 360–379. https://doi.org/10.1002/cpe.2995

[31] Konrad Kulakowski. 2015. Dynamic concurrent van Emde Boas array. *CoRR* abs/1509.06948 (2015). arXiv:1509.06948 http://arxiv.org/abs/1509.06948

[32] Justin J. Levandoski, David B. Lomet, and Sudipta Sengupta. 2013. The Bw-Tree: A B-tree for new hardware platforms. In *29th IEEE International Conference on Data Engineering, ICDE 2013, Brisbane, Australia, April 8-12, 2013*. IEEE Computer Society, 302–313. https://doi.org/10.1109/ICDE.2013.6544834

[33] Benedikt Mayr, Alexander Weinrauch, Mathias Parger, and Markus Steinberger. 2021. Are van Emde Boas trees viable on the GPU?. In *2021 IEEE High Performance Extreme Computing Conference, HPEC 2021, Waltham, MA, USA, September 20-24, 2021*. IEEE, 1–7. https:

//doi.org/10.1109/HPEC49654.2021.9622837

[34] Aravind Natarajan and Neeraj Mittal. 2014. Fast concurrent lock-free binary search trees. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '14, Orlando, FL, USA, February 15-19, 2014*. ACM, 317–328. https://doi.org/10.1145/2555243.2555256

[35] Aravind Natarajan, Lee Savoie, and Neeraj Mittal. 2013. Concurrent Wait-Free Red Black Trees. In *Stabilization, Safety, and Security of Distributed Systems - 15th International Symposium, SSS 2013, Osaka, Japan, November 13-16, 2013. Proceedings (Lecture Notes in Computer Science, Vol. 8255)*. Springer, 45–60. https://doi.org/10.1007/978-3-319-03089-0_4

[36] Rotem Oshman and Nir Shavit. 2013. The SkipTrie: low-depth concurrent search without rebalancing. In *ACM Symposium on Principles of Distributed Computing, PODC '13, Montreal, QC, Canada, July 22-24, 2013*, Panagiota Fatourou and Gadi Taubenfeld (Eds.). ACM, 23–32. https://doi.org/10.1145/2484239.2484270

[37] William W. Pugh. 1990. Skip Lists: A Probabilistic Alternative to Balanced Trees. *Commun. ACM* 33, 6 (1990), 668–676. https://doi.org/10.1145/78973.78977

[38] Arunmoezhi Ramachandran and Neeraj Mittal. 2015. A Fast Lock-Free Internal Binary Search Tree. In *Proceedings of the 2015 International Conference on Distributed Computing and Networking, ICDCN 2015, Goa, India, January 4-7, 2015*. ACM, 37:1–37:10. https://doi.org/10.1145/2684464.2684472

[39] Bratin Saha, Ali-Reza Adl-Tabatabai, and Quinn Jacobson. 2006. Architectural Support for Software Transactional Memory. In *39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-39 2006), 9-13 December 2006, Orlando, Florida, USA*. IEEE Computer Society, 185–196. https://doi.org/10.1109/MICRO.2006.9

[40] Nir Shavit and Dan Touitou. 1995. Software Transactional Memory. In *Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing, Ottawa, Ontario, Canada, August 20-23, 1995*. ACM, 204–213. https://doi.org/10.1145/224964.224987

[41] Anubhav Srivastava and Trevor Brown. 2022. Elimination (a, b)-trees with fast, durable updates. In *PPoPP '22: 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, Seoul, Republic of Korea, April 2 - 6, 2022*. ACM, 416–430. https://doi.org/10.1145/3503221.3508441

[42] Ibrahim Umar, Otto Johan Anshus, and Phuong Hoai Ha. 2015. DeltaTree: A Locality-aware Concurrent Search Tree. In *Proceedings of the 2015 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems, Portland, OR, USA, June 15-19, 2015*. ACM, 457–458. https://doi.org/10.1145/2745844.2745891

[43] Peter van Emde Boas. 1977. Preserving Order in a Forest in Less Than Logarithmic Time and Linear Space. *Inf. Process. Lett.* 6, 3 (1977), 80–82. https://doi.org/10.1016/0020-0190(77)90031-X

[44] Peter van Emde Boas, R. Kaas, and E. Zijlstra. 1977. Design and Implementation of an Efficient Priority Queue. *Math. Syst. Theory* 10 (1977), 99–127. https://doi.org/10.1007/BF01683268

[45] Ziqi Wang, Andrew Pavlo, Hyeontaek Lim, Viktor Leis, Huanchen Zhang, Michael Kaminsky, and David G. Andersen. 2018. Building a Bw-Tree Takes More Than Just Buzz Words. In *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*. ACM, 473–488. https://doi.org/10.1145/3183713.3196895

[46] Dan E. Willard. 1983. Log-Logarithmic Worst-Case Range Queries are Possible in Space Theta(N). *Inf. Process. Lett.* 17, 2 (1983), 81–84. https://doi.org/10.1016/0020-0190(83)90075-3

[47] Richard M. Yoo, Christopher J. Hughes, Konrad Lai, and Ravi Rajwar. 2013. Performance evaluation of Intel® transactional synchronization extensions for high-performance computing. In *International Conference for High Performance Computing, Networking, Storage and Analysis, SC'13, Denver, CO, USA - November 17 - 21, 2013*. ACM, 19:1–19:11. https://doi.org/10.1145/2503210.2503232

[48] Qiang Zeng, Xiaorui Jiang, and Hai Zhuge. 2012. Adding Logical Operators to Tree Pattern Queries on Graph-Structured Data. *Proc. VLDB Endow.* 5, 8 (2012), 728–739. https://doi.org/10.14778/2212351.2212355

A Artifact Description

The artifacts for this paper are available at the following link: https://zenodo.org/records/10493788.

The artifact requires `GCC 9.3.0` and `GNU Make 4.2.1`, as well as `python3`. We have run our experiments on `ubuntu 20.04`.

We have created a set of Python scripts that generate graphs for different workloads. The workloads include the comparison of different vEB set variations, the comparison of vEB sets and maps, as well as the comparison of vEB maps and other structures. We have done our best to make it straightforward to mix and match the workloads and the parameters. Here is a brief description of each script and its role in the artifact:

- `01-sets-comparison.py`: This script reproduces Figure 6, while the workload and the prefilling amount is customizable. If `PAPI` is enabled, LLC cache miss performance will be compared too.
- `02-sets-vs-maps.py`: This script reproduces Figures 7a and 7c, and offers LLC cache performance comparison between vEB sets and maps.
- `03-veb-vs-others-uniform.py`: This script reproduces the first row of Figure 8. Similar to other scripts, the prefilling denominator, universe size, and thread count are easily customizable in the script.
- `04-veb-vs-others-zipfian.py`: This script reproduces the second row of Figure 8. The α parameter of the Zipfian distribution can be modified to experiment with less or more skewed workloads.
- `05-successor-vs-nonlin-pub.py`: This script reproduces Figure 15.
- `06-veb-vs-others-memory-and-insertion-time.py`: This script reproduces Figures 11 and 13.
- `07-veb-vs-std.py`: This script reproduces Figure 14.

More information about the artifact can be found in the `README.md` file.