Efficient concurrent search trees using portable fine-grained locality

Phuong Hoai Ha, Otto J. Anshus, Ibrahim Umar

Abstract—Concurrent search trees are crucial data abstractions widely used in many important systems such as databases, file systems and data storage. Like other fundamental abstractions for energy-efficient computing, concurrent search trees should support both high concurrency and fine-grained data locality in a platform-independent manner. However, existing portable fine-grained locality-aware search trees such as ones based on the van Emde Boas layout (VEB-based trees) poorly support concurrent update operations while existing highly-concurrent search trees such as non-blocking search trees do not consider fine-grained data locality. In this paper, we first present a novel methodology to achieve both portable fine-grained data locality and high concurrency for search trees. Based on the methodology, we devise a novel locality-aware concurrent search tree called GreenBST. To the best of our knowledge, GreenBST is the first practical search tree that achieves both portable fine-grained data locality and high concurrency. We analyze and compare GreenBST energy efficiency (in operations/Joule) and performance (in operations/second) with seven prominent concurrent search trees on a high performance computing (HPC) platform (Intel Xeon), an embedded platform (ARM), and an accelerator platform (Intel Xeon Phi) using parallel micro-benchmarks (Synchronobench). Our experimental results show that GreenBST achieves both the best energy efficiency and performance on all the different platforms. GreenBST achieves up to 50% more energy efficiency and 60% higher throughput than the best competitor in the parallel benchmarks. These results confirm the viability of our new methodology to achieve both portable fine-grained data locality and high concurrency for search trees.

1 INTRODUCTION

As energy efficiency is emerging as one of the most important metrics in designing computing systems [26], [41], [45], [57], data should be organized and accessed in an energy efficient manner. Unlike conventional locality-aware data structures and algorithms that only concern whether the data is on-chip (e.g., in cache) or not (e.g., in DRAM), new energy-efficient data structures and algorithms must consider data locality in finer-granularity: where on chip the data is stored. It is estimated that for chips using the 10nm technology, the energy gap between accessing data in nearby on-chip memory (e.g., data in SRAM) and accessing data across the chip (e.g., on-chip data at the distance of 10mm), will be as much as 75x (2pJ versus 150pJ), whereas the energy gap between accessing on-chip data and accessing off-chip data (e.g., data in DRAM) will be only 2x (150pJ versus 300pJ) [18]. Therefore, in order to construct energy efficient software systems, data structures and algorithms should support not only high parallelism but also fine-grained data locality [18].

Concurrent search trees are crucial data structures that are widely used as a back-end in many important systems such as databases (e.g., SQLite [29]), filesystems (e.g., Btrfs [58]), and schedulers (e.g., Linux’s Completely Fair Scheduler (CFS)), among others. However, existing highly concurrent search trees do not consider fine-grained locality. Non-blocking concurrent search trees (e.g., [14], [22], [23], [59]) and Software Transactional Memory (STM)-based search trees (e.g., [2], [12], [17], [21]) among others) have been regarded as the state-of-the-art concurrent search trees. The prominent highly concurrent search trees widely used in several benchmark distributions are the concurrent red-black trees developed by Oracle Labs [21] and the concurrent AVL trees developed by Stanford [12]. These highly concurrent search trees, however, do not take into account fine-grained data locality.

Existing fine-grained locality-aware search trees poorly support concurrency and are usually platform-dependent, limiting their portability across different platforms. For example, Intel Fast [30] and Palm [59] are optimized for a specific platform. Concurrent B-trees (e.g., B-link tree [32]) only perform well if their parameter $B$ is chosen correctly. More recent research on system- and database- oriented concurrent search trees [15], [33], [54], [55], [57], [45] has produced some excellent examples of cache-conscious concurrent search trees. Unfortunately, none of the research addresses the issue of portability, because they were mostly developed and evaluated for a specific platform.

Portable fine-grained locality can be theoretically achieved using cache-oblivious (CO) methodology [25]. In the CO methodology, an algorithm is categorized as cache-oblivious for a two-level memory hierarchy if it has no variables that need to be tuned with respect to hardware parameters, such as cache size and cache-line length, in order to optimize its cache complexity, assuming that the optimal off-line cache replacement strategy is used. If a cache-
oblivious algorithm is optimal for an arbitrary two-level memory, the algorithm is also optimal for any adjacent pair of available levels of the memory hierarchy [10], [29]. Therefore, cache-oblivious algorithms are expected to be locality-optimized irrespective of variations in memory hierarchies, enabling portable fine-grained locality.

Portable fine-grained data locality for sequential search trees can be theoretically achieved using the van Emde Boas (vEB) layout [24], which is analyzed using ideal cache (CO) models [25]. The vEB layout has inspired several cache-oblivious sequential search trees such as cache-oblivious B-trees [7], [8] and cache-oblivious binary trees [11]. The vEB-based trees recursively arrange related data in contiguous memory locations, minimizing data transfer between any two adjacent levels of the memory hierarchy (see Section 3.1 for details).

However, existing vEB-based trees poorly support concurrent update operations and have high overhead and large memory footprints. Inserting or deleting a node may result in relocating a large part of the tree in order to maintain the vEB layout. Bender et al. [9] discussed the problem and provided important theoretical designs of concurrent vEB-based B-trees. Nevertheless, we have found that these designs are not very efficient in practice due to the actual overhead of maintaining necessary pointers as well as their large memory footprint (see Section 8.3).

Our practical insight is that it is unnecessary to keep the entire vEB-based tree in a single contiguous block of memory. In fact, allocating a contiguous block of (virtual) memory for a vEB-based tree does not guarantee a contiguous block of physical memory. Modern OSes and systems utilize different sizes of continuous physical memory blocks, for example, in the form of pages and cache-lines. A contiguous block in virtual memory can be translated into several pages with gaps in physical memory (e.g., RAM); a page can be cached by several cache lines with gaps at any level of cache. This is one of the motivations for our new bounded ideal cache model (see Section 2.2). The upper bound on the contiguous block size can be obtained easily from any system (e.g., page-size), which is platform-independent. In fact, the memory transfer complexity of our search operation in the new bounded ideal cache model is independent of the upper bound values (see Lemma 3.1 in Section 3.2).

1.1 Our contributions
In this paper, we investigate whether it is practical to achieve both fine-grained data locality and portability in concurrent search trees and if so, whether portable fine-grained data locality actually improves energy efficiency and performance compared with conventional coarse-grained data locality used in B-trees. To the best of our knowledge, there is no previous experimental study of how portable fine-grained data locality actually influences energy efficiency and performance in concurrent search trees. Such a study is necessary nowadays when multilevel memory hierarchies are becoming more prominent in commodity systems. Modern CPUs tend to have at least three levels of caches.

Our contributions are fourfold:
1) We have devised a new bounded ideal cache model (or BCO) (see Section 2) and a novel concurrency-aware vEB layout (or CvEB) that makes the vEB layout suitable for highly concurrent data structures with update operations (see Section 5).
2) Based on the new concurrency-aware vEB layout, we have devised a new portable fine-grained locality-aware concurrent search tree called GreenBST (see Sections 3 and 5). To the best of our knowledge, GreenBST is the first practical search tree that achieves both portable fine-grained data locality and high concurrency. GreenBST is open source and available at: https://github.com/uit-agc/GreenBST.
3) We have analyzed and compared GreenBST performance (in operations/second) and energy efficiency (in operations/Joule) with seven prominent concurrent search trees (see Table 1) on a high performance computing (HPC) platform (Intel Xeon), an embedded platform (ARM), and an accelerator platform (Intel Xeon Phi) (see Table 2 in Section 7 using parallel micro-benchmarks (Synchrobench [27]). Our experimental results show that GreenBST achieved up to 50% more energy efficiency and 60% higher throughput than the best competitor on the commodity HPC, embedded and accelerator platforms. Unlike platform-dependent search trees that are optimized for a specific platform (e.g., Fast [30] and Palm [39]), GreenBST is platform-independent and performance-portable.
4) We have provided insights into how portable fine-grained data locality actually influences energy efficiency and performance in concurrent search trees (see Section 5). Among our findings are: i) portable fine-grained data locality is able to reduce data movement, resulting in lower energy consumption and higher performance across HPC, embedded and accelerator platforms; ii) reducing the (hidden) overhead (e.g., pointers) in the tree structure allows a larger portion of real data in each memory transfer, resulting in better energy efficiency; and iii) for multicore platforms, efficient concurrency control is necessary for energy-efficient data structures, in addition to locality-awareness.

2 BOUNDED IDEAL CACHE MODEL
In order to devote locality-aware concurrent search trees, we need theoretical execution models that promote both data locality and concurrency. In this section, we present a new execution model called bounded ideal cache model (BCO) that promotes both data locality and concurrency. Unlike previous models such as the I/O model [3] and the ideal cache model (CO) [25] that promote only data locality, the new BCO model enables new designs of concurrent data structures that achieve both data locality and high concurrency (see concurrency-aware vEB layout (Section 5) and GreenBST (Section 4)).

Before presenting the new BCO model, we first provide some background on relevant models, namely the ideal cache model (CO) [25]. These models enable the analysis of data transfer between two levels of the memory hierarchy. Lower data transfer complexity implies better data locality and, therefore, higher energy efficiency since energy consumption caused by data transfer dominates the total energy consumption [18].

2.1 Ideal cache model
The ideal cache model was introduced by Frigo et al. in [25], which is similar to the I/O model [3] except that the block size $B$ and memory size $M$ are unknown. Using the same analysis as the

3. A performance-portable tree is one that achieves high performance across a variety of platforms.

3. B-link tree is a highly-concurrent B-tree variant and it’s still used as a backend in popular database systems such as PostgreSQL (https://github.com/postgres/postgres/blob/master/src/backend/access/nbtree/README)
We define bounded ideal cache (e.g., register size, cache line size), obtaining a single upper bound. Therefore, without prior knowledge about a memory hierarchy, algorithms that can utilize fine-grained data locality (e.g., at cache level) are optimal for any adjacent pair of a multi-level memory. This feature enables designing algorithms that perform better and are more robust than the cache-aware algorithms [10].

Note that in the ideal cache model, B-tree is no longer optimal because of the unknown block size. Instead, the vEB-based trees [7], [8], [9], [11] are optimal in the model. We refer the readers to [10], [25] for a more comprehensive overview of the I/O model and the ideal cache model.

### 2.2 Bounded ideal cache model

We define bounded ideal cache model (BCO) to be the ideal cache model (CO) with an extension that an upper bound on the unknown memory block size is known in advance. This extension is inspired by the fact that although the exact block size at each level of the memory hierarchy is architecture-dependent (e.g., register size, cache line size), obtaining a single upper bound for all the block sizes (e.g., register size, cache line size, and page size) is architecture-independent. For example, the page size obtained from the operating system is such an upper bound.

Unlike the CO model, the new BCO model facilitates designing concurrent data structures and algorithms that support not only fine-grained data locality but also high concurrency. The BCO model inherits fine-grained data locality from the CO model while achieving higher concurrency by the new ability of organizing data in smaller memory chunks of known size. Without knowing an upper bound on the unknown memory block size, we must organize data for unknown block size that can be extremely large. We will elaborate on the advantage of the BCO model in designing concurrency-aware vEB layout in Section 3.2.

Moreover, the new BCO model maintains the key feature of the original CO model [25]. First, temporal locality (i.e., reuse of the same data located in cache) is exploited perfectly as there are no constraints on cache size M in the BCO model. Since the Least Recently Used (LRU) cache replacement policy with cache size \((1 + \varepsilon)M\), where \(\varepsilon > 0\), is almost as good as the optimal offline cache replacement policy with cache size \(M\) [40], an optimal cache replacement policy can be assumed in the BCO model. Second, analysis for a simple two-level memory is applicable for an unknown multilevel memory (e.g., registers, L1/L2/L3 caches and memory). Namely, an algorithm that is optimal in terms of data transfer for a two-level memory is asymptotically optimal for an unknown multi-level memory. This feature enables designing algorithms that can utilize fine-grained data locality (e.g., at cache levels) in the multi-level memory hierarchy of modern architectures.

<table>
<thead>
<tr>
<th>#</th>
<th>Algorithm</th>
<th>Ref</th>
<th>Description</th>
<th>Synchronization</th>
<th>Affiliation</th>
<th>Data structure</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>CITRUS</td>
<td>[5]</td>
<td>RCU-based search tree</td>
<td>lock-based</td>
<td>Technion</td>
<td>binary-tree</td>
</tr>
<tr>
<td>3</td>
<td>LFIBST</td>
<td>[56]</td>
<td>Non-blocking binary search tree</td>
<td>lock free</td>
<td>UT Dallas</td>
<td>binary-tree</td>
</tr>
<tr>
<td>4</td>
<td>BSTTK</td>
<td>[39]</td>
<td>Portably scalable concurrent search tree</td>
<td>lock-based</td>
<td>EPFL</td>
<td>binary-tree</td>
</tr>
<tr>
<td>5</td>
<td>LYBTREE</td>
<td>[22]</td>
<td>Concurrent B-tree (B-link tree)</td>
<td>lock-based</td>
<td>CMU</td>
<td>b-tree</td>
</tr>
<tr>
<td>6</td>
<td>ABTREE</td>
<td>[13]</td>
<td>Lock-free (a,b)-tree</td>
<td>lock-free</td>
<td>Technion</td>
<td>(a,b)-tree</td>
</tr>
<tr>
<td>7</td>
<td>BWTREE</td>
<td>[24]</td>
<td>Lock-free Bw-tree</td>
<td>lock-free</td>
<td>CMU</td>
<td>b-tree</td>
</tr>
<tr>
<td></td>
<td>new GreenBST</td>
<td></td>
<td>Locality aware concurrent search tree</td>
<td>lock-based</td>
<td>this paper</td>
<td>b-tree</td>
</tr>
</tbody>
</table>

Table 1: List of evaluated concurrent tree algorithms. These algorithms are sorted by synchronization type.

I/O model, an algorithm is categorized as cache-oblivious if it has no variables that need to be tuned with respect to cache size and cache-line length, in order to optimize the data transfer complexity. An optimal cache-oblivious algorithm for a two-level memory is also optimal for any adjacent pair of a multi-level memory. Therefore, without prior knowledge about a memory hierarchy, a cache-oblivious algorithm can automatically adapt to the memory hierarchy with multiple levels. It has been reported that cache-oblivious algorithms perform better and are more robust than the cache-aware algorithms [10].

We first define the notations that will be used to elaborate on the improvement:

- \(b_i\) (unknown): block size (in terms of the number of nodes) at level \(i\) of the memory hierarchy (like \(B\) in the I/O model [3]), which is unknown as in the ideal cache model [25]. When the specific level \(i\) of the memory hierarchy is irrelevant, we use notation \(B\) instead of \(b_i\) in order to be consistent with the I/O model.
- \(U\) (known): the upper bound (in terms of the number of nodes) on the block size \(b_i\) of all levels \(i\) of the memory hierarchy.
- GNode: the largest recursive subtree of a vEB-based search tree that contains all \(U\) nodes (see dashed triangles of height \(2^k\) in Figure 2b). GNode is a fixed-size tree-container with the vEB layout.
- "level of detail" \(k, k \in \mathbb{N}\), is a number representing a partition of a vEB-based tree into recursive subtrees of height at most \(2^k\).
- \(T\): size and height of the whole tree in terms of basic nodes (not in terms of GNodes).

### 3 Concurrency-aware vEB layout

In this section, we present a new concurrency-aware van Emde Boas layout (vEB), an improvement to the conventional van Emde Boas layout (vEB). The new vEB layout is based on the new BCO model presented in Section 2.

We define the notations that will be used to elaborate on the improvement:

- \(b_i\) (unknown): block size (in terms of the number of nodes) at level \(i\) of the memory hierarchy (like \(B\) in the I/O model [3]), which is unknown as in the ideal cache model [25]. When the specific level \(i\) of the memory hierarchy is irrelevant, we use notation \(B\) instead of \(b_i\) in order to be consistent with the I/O model.
- \(U\) (known): the upper bound (in terms of the number of nodes) on the block size \(b_i\) of all levels \(i\) of the memory hierarchy.
- GNode: the largest recursive subtree of a vEB-based search tree that contains all \(U\) nodes (see dashed triangles of height \(2^k\) in Figure 2b). GNode is a fixed-size tree-container with the vEB layout.
- "level of detail" \(k, k \in \mathbb{N}\), is a number representing a partition of a vEB-based tree into recursive subtrees of height at most \(2^k\).
- \(T\): size and height of the whole tree in terms of basic nodes (not in terms of GNodes).

#### 3.1 Conventional van Emde Boas (vEB) layout

The conventional van Emde Boas (vEB) layout has been introduced in [24] and widely used in cache-oblivious data structures [7], [8].
1 will affect the other subtrees we cannot use dynamic node allocation via pointers as in highly dynamic vEB layout (or CvEB layout for concurrency-aware data structures with update operations, we introduce a novel dynamic vEB layout (see Lemma 3.1) while the CvEB layout supports high concurrency with update operations. For example, when subtree W in Figure 2a is full, the CvEB layout enables allocating a new subtree X and linking X to W as in k-ary search trees. By incorporating pointers, the CvEB layout enables highly concurrent (update) operations on subtrees (e.g., by using universal methodologies for implementing highly concurrent data structures [28]).

Lemma 3.1. For any upper bound \( U \) of the unknown memory block size \( B \), a search in a complete binary tree with the new concurrency-aware vEB layout achieves the optimal memory transfer \( O(\log_2 N) \), where \( N \) and \( B \) are the tree size and the unknown memory block size in the ideal cache model [25], respectively.

Proof. (Sketch) Figure 2b illustrates the proof. Let \( k \) be the coarsest level of detail such that every recursive subtree contains at most \( B \) nodes, where \( B \) is unknown. Since \( B \leq U \), \( k \leq L \), where \( L \) is the coarsest level of detail at which every recursive subtree (or GNode) contains at most \( U \) nodes. That means there are at most \( 2^L-1 \) subtrees of height \( 2^k \) along the search path in a GNode and no subtree of height \( 2^k \) is split due to the boundary of GNodes. Namely, in Figure 3, each triangle of height \( 2^k \) fits within a dashed triangle of height \( 2^L \).

Because at any level of detail \( i \leq L \) in the CvEB layout, a recursive subtree of height \( 2^k \) is stored in a contiguous block of memory, each subtree of height \( 2^k \) within a GNode is stored in at most \( 2^L \) memory blocks of size \( B \) (depending on the starting location of the subtree in memory). Since every subtree of height \( 2^k \) fits in a GNode (i.e., no subtree is stored across two GNodes), every subtree of height \( 2^k \) in the tree is stored in at most \( 2^L \) memory blocks of size \( B \).

Let \( T \) be the tree height. A search will traverse \( \lceil T/2^k \rceil \) subtrees of height \( 2^k \) and thereby at most \( 2 \lceil T/2^k \rceil \) memory blocks are transferred.

Since a subtree of height \( 2^k+1 \) contains more than \( B \) nodes, \( 2^{k+1} \geq \log_2(B+1) \), or \( 2^k \geq 1/2 \log_2(B+1) \).

We have \( 2^{k-1} \leq N \leq 2^k \) since the tree is a complete binary tree.

This implies \( \log_2 N \leq T \leq \log_2 N + 1 \).

Therefore, the number of memory blocks transferred in a search is \( 2 \lceil T/2^k \rceil = 4 [\log_2 B + 1] = 4 \log_2 B + 4 \log_2 N + 2 \leq O(\log_2 N) \), where \( N \geq 2 \).

Figure 2 illustrates the new concurrency-aware vEB (CvEB) layout. Let \( L \) be the coarsest level of detail such that every recursive subtree contains at most \( U \) nodes. Namely, let \( H \) and \( S \) be the height and size of such a subtree then \( H = 2^L \) and \( S = 2^H - 1 \leq U \). The tree is recursively partitioned into level of detail \( L \) where each subtree represented by a triangle in Figure 3, is stored in a contiguous memory block of size \( U \). Unlike the conventional vEB, the subtrees at level of detail \( L \) are linked to each other using pointers, namely each subtree at level of detail \( k \geq L \) is not stored in a contiguous block of memory. Intuitively, since \( U \) is an upper bound on the unknown memory block size \( B \), storing a subtree at level of detail \( k \geq L \) in a contiguous memory block of size greater than \( U \), does not reduce the number of memory transfers, provided there is perfect alignment. For example, in Figure 2a, traversing from a subtree \( W \) at level of detail \( L \), which is stored in a contiguous memory block of size \( U \), to its child subtree \( X \) at the same level of detail will result in at least two memory transfers: one for \( W \) and one for \( X \). Therefore, it is unnecessary to store both \( W \) and \( X \) in a contiguous memory block of size \( 2U \). As a result, the memory transfer cost of search operations in the new CvEB layout is the same as in the conventional vEB layout (see Lemma 3.1) while the CvEB layout supports high concurrency with update operations. For example, when subtree \( W \) in Figure 2a is full, the CvEB layout enables allocating a new subtree \( X \) and linking \( X \) to \( W \) as in k-ary search trees. By incorporating pointers, the CvEB layout enables highly concurrent (update) operations on subtrees (e.g., by using universal methodologies for implementing highly concurrent data structures [28]).

Although the conventional vEB layout is useful for achieving data locality, it poorly supports concurrent update operations. Inserting (or deleting) a node at position \( i \) in the contiguous block storing the tree may restructure a large part of the tree. For example, inserting new nodes in the full subtree \( W_1 \) (a leaf subtree) in Figure 1 will affect the other subtrees \( W_2, W_3, \ldots, W_m \) because of rebalancing nodes between \( W_1 \) and \( W_2, W_3, \ldots, W_m \) in order to have space for new nodes. Even worse, we will need to allocate a new contiguous block of memory for the whole tree if the previously allocated block of memory for the tree runs out of space [11]. Note that we cannot use dynamic node allocation via pointers as in highly concurrent search trees since at any level of detail, each subtree in the vEB layout must be stored in a contiguous block of memory.

Figure 2: (a) The new concurrency-aware vEB layout. (b) A search path using the concurrency-aware vEB layout.

[9, 11]. Figure 1 illustrates the vEB layout. Suppose we have a complete binary tree of height \( h \). For simplicity, we assume \( h \) is a power of 2, i.e., \( h = 2^k, k \in \mathbb{N} \). The tree is recursively laid out in the memory as follows. The tree is conceptually split between nodes of height \( h/2 \) and \( h/2+1 \), resulting in a top subtree \( T \) and \( m_1 = 2^{h/2} \) bottom subtrees \( W_1, W_2, \ldots, W_{m_1} \) of height \( h/2 \). The \((m_1+1)\) top and bottom subtrees are then located in contiguous memory locations where \( T \) is located before \( W_1, W_2, \ldots, W_{m_1} \). Each of the subtrees of height \( h/2 \) is then laid out similarly to \((m_1+1)\) subtrees of height \( h/4 \), where \( m_2 = 2^{h/4} \). The process continues until each subtree contains only one node, i.e., the finest level of detail, 0.

One of the key features of the vEB layout is that the cost of searching a key located at a leaf in this layout is \( O(\log_2 N) \) memory transfers, where \( N \) is the tree size and \( B \) is the unknown memory block size in the ideal cache model [25]. Namely, the search operation in vEB-based trees is cache-oblivious. The search cost is optimal and matches the search cost of B-trees that requires the memory block size in the ideal cache model [25]. Namely, the cost of searching a key located at a leaf in this layout is \( \log N \) where \( \log \) is the tree size and \( \log \) is the unknown memory block size in the ideal cache model [25], respectively.

3.2 Concurrency-aware vEB layout

In order to make the vEB layout suitable for highly concurrent data structures with update operations, we introduce a novel concurrency-aware dynamic vEB layout (or CvEB layout for short). Our key idea is that if we know an upper bound \( U \) on the unknown memory block size \( B \) as in the BCO model (see Section 2), we can support dynamic node allocation via pointers while maintaining the optimal search cost of \( O(\log_2 N) \) memory transfers (see Lemma 3.1). The assumption on known upper bound \( U \) is inspired by the fact that in practice it is not necessary to keep the entire vEB tree in a single contiguous block of memory (see Section 1). Instead, if the tree is larger than some fixed (known) upper bound \( U \), then we break the tree into smaller vEB trees, each of which is stored in a contiguous block of memory (of size \( S \leq U \)).
4 GreenBST overview

To prove that the new conceptual CvEB layout (see Section 3) is useful for concurrent search trees to achieve both portable fine-grained data locality and high concurrency in practice, we devise a novel locality-aware concurrent search tree based on the new CvEB layout, which is called GreenBST.

A GreenBST $T_G$ consists of $|T_G|$ GNodes of fixed size $U$ (see Figure 3). Each of the GNodes contains a pointer-less binary search tree (BST) $T_i, i = 1, ..., |T_G|$. Nodes of tree $T_i$ are basic nodes which should be distinguished from GNodes. GreenBST provides the following operations: $\text{UPDATE}(v,T_G)$, which adds or removes value $v$ from GreenBST $T_G$, and $\text{SEARCH}(v,T_G)$, which determines whether value $v$ exists in $T_G$. We use term update operation for either insert or delete operation. We assume that duplicate values are not allowed inside the tree and a special value, for example 0, is reserved as the EMPTY value.

4.1 Data structures

The topmost level of GreenBST is represented by struct UNIVERSETYPE (line 17 in Figure 4) that contains pointer root to the basic root-node of the root GNode. GNodes are represented by struct GNODETYPE (line 3 in Figure 4). A GNode $G$ includes a pointer root pointing to $G$’s root, a nodes group of size $U$ that hold values/keys, and a link array of size $U$ that links $G$’s basic leaf-nodes to the roots of $G$’s child GNodes. GNode metadata contains: i) pointer nextRight pointing to the GNode’s right sibling; ii) field highKey containing the lowest value/key of the right sibling GNode; iii) counter rev used for optimistic concurrency [31]; and iv) a lock locked protecting the GNode for update operations. Fields locked, rev, highKey and nextRight are used for GreenBST concurrency control.

Each NODETYPE structure (line 11 in Figure 2) contains i) field value holding a value for guiding the search, or a data value (or key) if it is a leaf-node; and ii) field mark indicating a logically deleted node (if the field is set to true).

Struct MAPTYPE (line 14 in Figure 4), is used to remove pointers from the NODE structure (see Section 5.1 for details).

4.2 Balanced and concurrent tree

GreenBST high-level structure is inspired by the B+tree structure and GreenBST concurrency is inspired by the B-link tree concurrency which provides lock-less search operations [32]. However, unlike the B-link tree, GreenBST is an in-memory tree and uses the new CvEB layout for its GNodes. Moreover, GreenBST uses optimistic concurrency to handle lock-less concurrent search operations even in the occurrences of concurrent update operations.

To keep GreenBST balanced while supporting concurrent updates, the whole tree is built in a bottom-up manner, which is handled by the $\text{UPDATE}$ function (see Figure 6). Meanwhile, the search operation traverses GreenBST in a top-down, left-to-right manner using a combination of the $\text{GNODESEARCH}$ function to find the relevant leaf GNode (see Figure 5) and a binary search to find the relevant node within the found GNode.

Function $\text{UPDATE}$ inserts a given key at a leaf of GreenBST $T_G$, provided key does not exist in the tree (see Figure 6, line 11). It first finds the appropriate leaf GNode to store the key (line 2 and locks the GNode (or its siblings if the GNode has been split) using the $\text{MOVE_RIGHT}$ function (line 4). The $\text{MOVE_RIGHT}$ function serializes concurrent updates by combining right-scanning (line 59) and lock coupling (see lines 56–59). If key is inserted to a node at the last level $H$ of the found GNode (i.e., at the boundary of the GNode), the $\text{UPDATE}$ function will either rebalance the GNode’s embedded tree to reduce its height (line 16) or split the GNode into two GNodes (line 21).

Function $\text{REBALANCE}$(key, $T_i$) is responsible for rebalancing a GNode $T_i$ after an insertion (see Figure 6, line 11). GreenBST rebalance is incremental, significantly reducing the rebalance overhead. GreenBST incremental rebalance is described in detail in Section 5.3.

GreenBST split operation on a GNode $T_v$ distributes the member keys of $T_v$ between $T_v$ and a new GNode $T_s$ as evenly as possible (see Figure 6, line 21). The split operation also updates $T_v$.nextRight to point to $T_s$ (line 28) and fills $T_s$.highKey with $T_s$’s minimum key (line 27). Lastly, $T_s$’s minimum key is inserted into the parent GNode of both $T_v$ and $T_s$ (line 25). If $T_v$ is currently the root GNode, a new root GNode is created and it becomes the new parent of $T_v$ and $T_s$. Note that for a GNode $T_v$ with minimum key $k_{\text{min}}$, a new key less than $k_{\text{min}}$ will be forwarded to one of $T_v$’s left sibling GNodes. The leftmost sibling GNode will host the minimum key that has ever been inserted to GreenBST.

The delete operation in GreenBST is handled by the $\text{UPDATE}$ function which marks the leaf node containing key as deleted (see Figure 6, line 5). Deleted nodes are reclaimed by the rebalance and split operations. The offline memory reclamation techniques used in the B-link tree [32] can be deployed to merge nearly empty GNodes in the case where a large part of the workload is delete operations. GreenBST is, however, designed for workloads...
dominated by search and insert operations.

The search operation in GreenBST is a combination of function GNODESEARCH to find the associated GNode (line 12 in Figure 5) and a binary tree search using cached map within the found GNode (line 7; see Section 5.1 for details). The GNODESEARCH function traverses the tree from the root GNode down to a leaf GNode, at which the search is handed over to the binary search to find the searched key within that leaf GNode.

4.3 Customized concurrency control

GreenBST uses locks and variables nextRight and highKey to coordinate concurrent search and update operations, while counter rev is used for the search optimistic concurrency [31]. When a GNode G needs to be updated, G’s rev counter is increased by one before the update operation starts (lines 8 and 13 in Figure 6). The counter is increased by one again after the update operation finishes (see lines 8 and 37). Note that all update operations happen when the lock is still held (see line 8) and therefore, only one update operation may increase G’s rev counter and update G at a time. The rev counter prevents the search operation from returning a wrong key because of a concurrent update operation (lines 8 and 35 in Figure 5).

GreenBST search uses optimistic concurrency [31] to ensure that the operation always returns the correct answer even if it arrives at a GNode that is undergoing update operation (e.g., insert and delete). First, before starting to traverse a GNode G, a search operation records G’s rev counter (line 14 in Figure 5). Before following a link to a child GNode or returning a link, the search

Figure 5: Function SEARCH in GreenBST. The GNODESEARCH function will return the leaf GNode for finding the searched key. From there, a simple binary search using LEFT and RIGHT functions (see Figure 7) is used to find the key location in the GNode or its right siblings. The SEARCH function utilizes counter rev, pointer nextRight and value highKey to ensure correct results even when executing concurrently with update operations.

Figure 6: Function UPDATE in GreenBST. The insert operation can call the REBALANCE function if needed. The MOVE_RIGHT function serializes concurrent updates by combining right-scanning (see line 55) and lock coupling (see lines 56–59). The correctness proof of GreenBST is presented in Section 6.
5.1 Cached map instead of pointers

Although removing pointers connecting basic nodes in GNode reduces data transfer between memory levels, it raises several challenges. Two of the key challenges are how to properly connect child- and parent-nodes within a GNode and how to establish child-parent paths between GNodes (i.e., inter-GNode links). We address the former in this subsection and the latter in subsection 5.2.

We replace GNode basic (left and right) pointers with functions LEFT and RIGHT that utilize a cached map (see line 14 in Figure 7 and Figure 5). The LEFT and RIGHT functions, given an arbitrary node p and the memory address of its GNode base, return the addresses of p’s left and right child nodes, respectively, and 0 if p has no children (i.e., p is a leaf node of the GNode). The LEFT and RIGHT functions throughout GreenBST share a common cached map instance (Figure 7 line 1). As all GNodes use the same fixed-size vEB layout, only one map instance with size U is needed for all traversing operations. This approach makes GreenBST memory footprint small and keeps the frequently-used map instance in cache.

For example, assume that a GNode has 127 basic nodes. Without using the new cached map, the set of basic (left and right) pointers occupy 2032 bytes (127 × 16 bytes) of memory in a 64-bit operating system, four times the amount of memory used by the actual data (i.e., 127 × 4 bytes = 508 bytes, assuming node’s value is a 4-byte integer). Even if the node value is multiple words in length, pointers are still going to occupy a large part of GNode memory (e.g., even for 64-byte node value, pointers still occupy 20% of GNode memory). Therefore, using pointers is inefficient, because every block transfer between levels of memory carries a significant portion of pointers (or overhead) instead of actual data. The new cached map completely removes the pointer overhead.

Our new mapping approach addresses the drawbacks of previous approaches (e.g., pointers and arithmetic-based implicit addresses) and is unique to the concurrency-aware vEB layout as it exploits the fixed-size feature of GNodes. Previous approaches such as pointers and arithmetic-based implicit addresses in cache-oblivious (CO) trees are found to have weaknesses. Pointer-based approaches induce high overhead in term of data transferred between memory levels: the inclusion of pointers reduces the amount of real data (e.g., keys) in each block transferred. The implicit addresses, which demand arithmetic calculation for every node traversal, induce high computation overhead, especially when the tree is big. Our new mapping approach eliminates both pointer overhead (i.e., no pointers) and computation overhead (i.e., addresses are pre-computed and stored in the cached map). The LEFT(p,base) (resp. RIGHT(p,base)) function only finds p’s index and gets the address of p’s left child (resp. right child) from the cached map (e.g., idx at line 11 and map[idx].left at line 13 in Figure 7).

Note that the mapping approach does not induce memory fragmentation because the approach applies only for an individual GNode. Namely, the cached map is the access structure of an individual GNode, which is used to find the left/ right child of an basic node in the GNode, without need of left/ right pointers. As an GNode is a fixed tree-container of size U and with the well-defined vEB layout, the cached map is fixed. The update operations change only the values of GNode basic nodes (e.g., from EMPTY to an input value in the case of insertion), but do not change GNode structure.

5.2 Inter-GNode connection

To enable traversing from a GNode to its child GNodes, we develop a new inter-GNode connection mechanism. Figure 5 explains how the inter-GNode connection works in a pointer-less search operation. We logically assign binary values to basic edges within GNode so that each path from GNode root to a basic leaf node is represented by a unique bit-sequence. As basic nodes have only left and right conceptual edges, we assign 0 and 1 to the left and right edges, respectively (lines 25 and 30). The bit-sequence is then used as an index in the link array containing pointers to child GNodes (line 23). The maximum size of the bit representation is GNode height or log2(U) bits.

Particularly, each basic leaf node L of a non-leaf GNode has two child GNodes pointed by link[LEFT] and link[RIGHT] where L[LEFT] and L[RIGHT] are computed when the GNodeSearch function traverses GreenBST (lines 25, 29 and 32). For example, if L’s depth is 1 (i.e., GNode has only one basic node) and GNode height maxDepth is 3, L[LEFT] = 0 and L[RIGHT] = (1 << (3 − 1)) = 4 (line 32). Namely, link[0] and link[4] point to L’s left child GNode and L’s right child GNode, respectively.

5.3 Incremental rebalancing

GreenBST is inspired by the incremental rebalance proposed in [11] which significantly reduces the rebalance overhead. However, unlike the approach in [11] that occasionally has to rebalance the whole tree, we only apply the incremental rebalancing on the basic tree embedded in each GNode (line 41 in Figure 6). Note that for workloads dominated by search and insert operations, the high-level tree of GNodes is already balanced because of the bottom-up construction (see Section 4.2). For workloads dominated by delete operations, the high-level tree of GNodes may be unbalanced due to empty GNodes when the merge operation is not implemented.

To briefly explain the idea, we denote density( w ) as the ratio of number of keys inside a subtree T rooted at node w to the maximum number of keys that subtree T can hold (line 45 in Figure 6). For example, a subtree with root w at level H − 3, where H is GNode height, can hold at most 2^{H−1} − 1 keys. If the subtree only contains 3 keys, then density( w ) = 3 / 2^{H−1} = 0.42. Let G_i be the density threshold at level i, we have 0 < G_1 < G_2 < ... < G_H = 1, where H is GNode height. After a new key is inserted at a basic leaf v, we find the nearest ancestor node w of v so that density( w ) ≤ G_{depth( w )} where depth( w ) is the level where w resides, counted from the GNode root (e.g., depth(GNode.root) = 1) (see line 47).

If w is found, we rebalance the subtree rooted at w.
Density thresholds $\Gamma_i, i < H$, are tuning parameters and are set manually. In our GreenBST implementation, $\Gamma_i, i < H$, are set to 0.5.

6 Correctness proof

**Definition 6.1.** A GreenBST is well-formed if the following structural properties hold to all (concurrent) functions:

- The binary search tree (BST) embedded in each GNode is well-formed. Namely, BST has no key duplicates, the left (resp. right) sub-tree of a basic node BN contains keys less than (resp. greater than or equal to) BN.value.
- Links between GNodes are well-formed. Namely, if a GNode $G$ has a nextRight pointer to a sibling GNode $G'$, $G$ contains keys less than $G'$.highKey and $G'$ contains keys greater than or equal to $G$.highKey. If a basic leaf node BN in $G$ has a left (resp. right) pointer to a child GNode $G''$, $G''$ contains keys less than (resp. greater than or equal to) BN.value.
- There are no key duplicates among GNodes.

**Lemma 6.1.** Let $G$ be a GNode in a well-formed GreenBST GT. If the Update function appears atomic to the Search and GNODESEARCH functions, an Update function on $G$ makes GT continue being well-formed.

**Proof.** (Sketch) As the Update function appears atomic to the Search and GNODESEARCH functions (the hypothesis), we only need to prove that an Update function on $G$ makes GT continue being well-formed regardless of whether the function is interfered by another Update function.

**Case 1: no interference on $G$ from another Update function:**
If the Update function $GU$ deletes key (line 5 in Figure 6), it sequentially searches $G$’s BST for the basic node containing key as in conventional sequential BST and then marks the node, if found, as deleted (line 7), making $G$’s BST continue being well-formed. Note that deleted nodes are kept in $G$’s BST to maintain the BST structure until the maintenance operations (e.g., REBALANCE and SPLIT) rebuild the whole subtree.

If $GU$ inserts key without triggering maintenance operations (e.g., REBALANCE or SPLIT), it performs the conventional sequential BST insert operation on $G$’s BST, making $G$’s BST continue being well-formed (line 14).

If $GU$ invokes the REBALANCE function on $G$ (line 16), it searches $G$’s BST for an appropriate basic node $w$ and sequentially rebuilds the subtree rooted at $w$ as a balanced BST using the same keys (lines 41 - 51), maintaining the well-formed properties of $G$’s BST. Moreover, since the BST of inner GNodes $G$ is leaf-oriented, the REBALANCE function does not change the links between $G$ and its child GNodes, making the links continue being well-formed.

If $GU$ triggers the split operation on $G$, splitting $G$ into two GNode $G_1$ (or GNode) and $G_2$ (or newGNode) (lines 21 - 35), we will prove that links between $G_1$, $G_2$ and $G$’s parent GNode $G_P$ are well-formed. Note that since $G$’s BST (resp. $G_2$’s BST) is newly created from the lower half (resp. higher half) of $G$’s sorted keys, $G_1$’s BST and $G_2$’s BST are well-formed and there are no key duplicates between $G_1$ and $G_2$.

Indeed, as $G_2$.highKey == $G$.highKey and $G_2$.nextRight == $G$.nextRight (lines 25 - 26), the $G_2$.nextRight link between $G_2$ and $G$’s right sibling is kept well-formed as before splitting. Similarly, as $G_1$.highKey contains $G_2$’s lowest key $y$ and $G_1$.nextRight points to $G_2$ (lines 27 - 28), the $G_1$.nextRight link between $G_1$ and $G_2$ is well-formed. Note that since no other function has a reference to $G_2$ until $G_2$ is inserted to parent $G_P$ (line 33), the modifications on $G_1$ and $G_2$ during the split (lines 23 - 28) are atomic to other Update functions.

Regarding the links between parent $G_P$ and $G_2$, as $G_2$’s lowest key $y$ and pointer linkprev to $G_2$ are assigned to $G_2$’s basic leaf node $L$ and $L$’s right pointer $G_P$.link[li].right, respectively (lines 37 - 39), the link between $G_P$ and $G_2$ is well-formed (i.e., $G_2$’s keys are greater than or equal to $L$.value). Regarding the link between $G_P$ and $G_1$, because $G_1$ reuses the memory allocated to $G$ and the link between $G_P$ and $G$ is well-formed before the split, the link between $G_P$ and $G_1$ becomes $L$’s left pointer $G_P$.link[li].left and therefore is well-formed (i.e., $G_1$’s keys are less than $L$.value, or $G_2$’s lowest key $y$). Since $GU$ successfully locks $G_2$ (line 34) before invoking the Update function on $G_P$ (lines 12 - 38), the same well-formed proof on $G$ applies for $G_P$.

Note that since $G$’s memory and the link $K$ from $G_P$ to $G$ are re-used for $G_1$, a function that has read $K$ before $G$ is split, will able to access $G_1$ via $K$ and $G_2$ via $G$.nextRight, finding all $G$’s keys.

**Case 2: interference on $G$ from other Update functions:**
We will prove that an Update function $GU$ on $G$ appears atomic to other concurrent Update functions and therefore this case becomes Case 1. Indeed, as $GU$ that is modifying $G$, has successfully locked $G$ (line 4 in Figure 6), other concurrent Update functions on $G$ must wait for $GU$ to finish its modification and unlock $G$ (line 9 19 35 or 38). The linearization point of the Update function in this case is the time point when LOCK($G$) in the MOVE_RIGHT function (line 8 in Figure 6) returns (line 52 or 57).

As GreenBST is initiated as an empty GNode and thus well-formed, Lemma 6.1 implies that GreenBST is always well-formed if we can prove that the Update function appears atomic to the GNODESEARCH and Search functions (cf. Lemmas 6.3 and 6.6).

**Lemma 6.2.** Let to be the time at which a reference to the leaf GNode $L$ that is returned by the GNODESEARCH(key,GreenBST.maxDepth) function (invoked by the Search or Update function), is made (line 7 10 12 or 23 in Figure 5). Any change to $L$ at time $t > t_0$ will be observed (by the Search and Update functions) in either $L$ or one of $L$’s right siblings that are reachable via the nextRight pointers.

**Proof.** (Sketch) Changes to $L$ (e.g., inserting or deleting key) at time $t > t_0$ that cannot be found in $L$, occur when $L$ has been split into two leaf GNodes $L$ and $L'$, and the changes are located in $L'$ (lines 27 - 28 in Figure 6). Note that $L'$ is reachable from $L$ via the $L$.nextRight pointer (line 28). The splitting can occur repeatedly, creating a linked list of sibling GNodes originated from $L$.

To prove that following the nextRight pointers will eventually find any change made to $L$ at time $t > t_0$, we need to prove that the split operation on a GNode $G$ appears atomic to other concurrent Update and Search functions on $G$. Indeed, as the Update function $GU$ that performs the split operation on $G$, has successfully locked $G$ (line 4) and only releases the lock after finishing the split operation (line 35), the split operation appears atomic to other concurrent Update functions. The linearization point of the Update function in this case is the time point when LOCK($G$) in the MOVE_RIGHT function (line 8 in Figure 6) returns (line 52 or 57).

Moreover, as function $GU$ makes counter $G$.rev odd number before the split (line 13) and makes $G$.rev even number again only after finishing the split (line 29), concurrent Search functions $GS$
will wait for the update/split to finish before actually accessing GNode $G$ (lines 13 and 35 in Figure 3). Since the UPDATE function must successfully acquire $G$’s lock (line 14 in Figure 3) before increasing $G$.rev (line 15), only one UPDATE function can increase $G$.rev and therefore odd $G$.rev indicates an ongoing update. Note that if the split function interferes with $GS$ and makes $G$.rev even number again between lines 13 and 35 in Figure 3. $GS$ will discover that $G$.rev has been changed (i.e., $G$.rev ≠ rev) and will then wait for the split operation to finish (line 35 in Figure 3). In this case, the linearization point of the split operation is the time point when the UPDATE function $GU$ is the time point when Increment($G$.rev) returns (Figure 6, line 6 in the case of deletion and line 13 in the case of insertion).

**Lemma 6.4.** Let $t_0$ be the time at which a reference to a leaf GNode $LG$ returned by the GNODESEARCH(key, GreenBST, maxDepth) function is made. If key in the tree, it is in LG at time $t_0$.

**Proof.** (Sketch) According to Lemma 6.3, if key is in the tree, it is in the subtree rooted at LG at time $t_0$. As LG is a leaf GNode (line 15), key, if existing, is in LG at time $t_0$.

**Lemma 6.5.** Let $L$ be the GNode that is returned by the GNODESEARCH(key, GreenBST, maxDepth) function. The key, if existing, is located in the first GNode $fG$ with $fG$.highKey ≥ key in the linked list of $L$’s siblings originated from $L$ (including $L$).

**Proof.** (Sketch) Let $t_0$ be the time at which a reference to the GNode $L$ returned by the GNODESEARCH(key, GreenBST, maxDepth) function is made. According to Lemma 6.4, key, if existing, is located in $L$ at time $t_0$ and therefore key < $L$.highKey at time $t_0$ according to the definition of highKey (line 8 in Figure 4). If $L$ is then split into two leaf GNodes $L$ and $L'$ and key is moved to $L'$. $L$.highKey ≤ key < $L'$.highKey according to the split operation (lines 21–27 in Figure 9). Note that the split operation is atomic to other concurrent functions (see the proof of Lemma 6.2). Arguing similarly for further split operations on $L$ and $L'$, which eventually create a linked list of $L$’s siblings $L → L_1 → … → L_k$ (see Lemma 6.2), we have key located in sibling $L_i$ where $L_{i−1}.highKey ≤ key < L_i.highKey$.

**Lemma 6.6.** The SEARCH function is correct.

**Proof.** (Sketch) We will prove that the result returned by the SEARCH(key, GreenBST, maxDepth) function in Figure 3 is the same as one returned by the conventional sequential binary search, even when the UPDATE function is interfering.

Let $t_0$ be the time at which a reference to the GNode $LG$ returned by the GNODESEARCH function (line 2 in Figure 3) is made. According to Lemma 6.4, key, if existing, is located in LG at time $t_0$.

Let $cG$ be the current leaf GNode that $S$ is visiting. Initially, $cG$ is LG.

**Case 1: no interference on $cG$ from UPDATE functions (i.e., no interference between line 13 and 35 when GNode = G).**

If $key < cG$.highKey, $GS$ traverses $G$’s internal binary search tree (BST) using $key$, reaching the appropriate basic leaf node $LN$ (lines 23–30) and its associated (left or right) pointer $link[bits]$ to a child GNode $G_{k+1}$ at the next GNode level (line 59). As in the conventional binary search tree, $key$, if existing, is located in the subtree rooted at $G_{k+1}$ (see Figure 3 for illustration).

If $key ≥ cG$.highKey, $key$, if in the tree, was moved to one of subtrees rooted at $G_k$. sibling GNodes between the time $GS$ got a reference to $G_k$ (line 12 or 30) and the time $GS$ started to access $G_k$ (line 14). In this case, the next GNode $G_{k+1}$ is $G_k$. first sibling GNode (line 16) whose subtree contains $key$ (if $key$ is in the tree) (see Definition 6.2).

**Case 2: interference on $G_k$ from an UPDATE function $GU$.

In this case, we will prove that function $GU$ appears atomic to function $GS$ and therefore this case becomes Case 1.

Indeed, similar to the proof of Lemma 6.2 as $GU$ makes counter $G_k$.rev odd number during its update (Figure 6, lines 6–8 for deletion, lines 13–17 for insertion without maintenance, lines 13–18 for insertion with rebalance, and lines 13–29 for insertion with split), $GS$ will discover $GU$’s interference during its search via checking counter $G_k$.rev (lines 14–34 in Figure 5) and will wait for $GU$ to finish before actually accessing $G_k$ (line 35). The linearization point of the GNODESEARCH function $GS$ is the time point when the function observes that $G_k$.rev is unchanged and even at line 35 in Figure 5. The linearization point of the UPDATE function $GU$ is the time point when Increment($G_k$.rev) returns (Figure 6, line 6 in the case of deletion and line 13 in the case of insertion).
and \cite{6.5} \(S\) will find the correct right sibling \(sG\) of \(cG\) where \(key\), if existing, is located, by following \texttt{nextRight} pointers and checking if \(key < sG.highKey\) (lines \cite{3.3 - 3.5}). Note that \(S\) will always find a \(sG\) satisfying \(key < sG.highKey\) since \(cG\)'s last (rightmost) sibling \(IG\) has \(IG.highKey = \infty\) (line \cite{8.2} in Figure \cite{4.6}). This case becomes Case 2 for the sibling GNode \(IG\).

\textbf{Lemma 6.7.} The \texttt{UPDATE} function is correct.

\textbf{Proof.} (Sketch) We will prove that the \texttt{UPDATE}(\(key\), \textit{GreenBST}, \textit{maxDepth}) function in Figure \cite{6.5} has the same effect as does the update operation of the conventional sequential binary search trees, even when other \texttt{UPDATE} functions are interfering.

Let \(t_0\) be the time at which a reference to the leaf GNode \(IG\) returned by the \texttt{GNODESEARCH} function (line \cite{3.4} in Figure \cite{4.6}), is made. According to Lemma \cite{6.4} \(key\), if existing, is located in \(IG\) at time \(t_0\).

Let \(cG\) be the current leaf GNode that \(U\) is visiting. Initially, \(cG\) is \(IG\).

\textit{Case 1: no interference on \(cG\) from another \texttt{UPDATE} function since time \(t_0\).} In this case, \(U\) performs either the conventional insert operation on \(cG\)'s internal binary search tree in the case of insertion (line \cite{4.3} in Figure \cite{4.6}) or marks as \texttt{deleted} the basic node containing \(key\) in the case of deletion (line \cite{7.3}), which has a similar effect as does the insert or delete operation of the conventional binary search tree. In addition, if the maintenance conditions (e.g., rebalance or split) are satisfied, the maintenance is performed sequentially without any effect on the correctness (cf. Lemma \cite{6.1}). In this case, the linearization point of the \texttt{UPDATE} function is the linearization point of the \texttt{GNODESEARCH} function invoked at line \cite{3.2} (cf. Lemma \cite{6.3}).

\textit{Case 2: interference without splitting on \(cG\) from other \texttt{UPDATE} functions since time \(t_0\).} As \(U\) has successfully locked \(cG\) (line \cite{3.3}) before performing any insertion or deletion, other concurrent update functions on \(cG\) must wait for \(U\) to finish its modification and unlock \(cG\) (line \cite{5.3} or \cite{5.3}). As \(U\)’s update on \(cG\) is not interfered by other update functions because of locking, \(U\)’s update has a similar effect as does the insert or delete operation of the conventional sequential binary search tree. In this case, the linearization point of the \texttt{UPDATE} function is the time point when \(\texttt{LOCK}(G)\) in the \texttt{MOVE_RIGHT} function invoked at line \cite{4.4} returns (line \cite{5.4} or \cite{5.7}).

\textit{Case 3: interference with splitting on \(cG\) from other \texttt{UPDATE} functions since time \(t_0\).} In this case, according to Lemmas \cite{6.2} and \cite{6.5} \(U\) will find the correct right sibling \(sG\) of \(cG\) where \(key\) should be located, by following \texttt{nextRight} pointers and checking if \(key < sG.highKey\) in the \texttt{MOVE_RIGHT} function invoked at line \cite{4.4} (lines \cite{5.3} - \cite{5.5}). This case becomes Case 2 for the sibling GNode \(sG\).

\textbf{Lemma 6.8.} \textit{GreenBST is deadlock-free.}

\textbf{Proof.} (Sketch) As the \texttt{SEARCH} function is lock-less, we only need to prove that concurrent instances of the \texttt{UPDATE} function lock GNodes in a well-defined order.

Indeed, for GNodes located at different tree levels, the \texttt{UPDATE} function locks them from a GNode with lower level (i.e., child GNode) to a GNode with higher level (i.e., parent GNode) in the case of splitting (lines \cite{5.3} - \cite{5.3} in Figure \cite{4.6}). For GNodes located at the same tree level, the \texttt{UPDATE} function locks them from left to right in the \texttt{MOVE_RIGHT} function by following the \texttt{nextRight} pointers (lines \cite{5.4} - \cite{5.7}).

\begin{table}[h]
\centering
\begin{tabular}{|c|c|c|c|}
\hline
Name & HPC & ARM & MIC \\
\hline
System & Intel Haswell-EF Octa & Samsung Exynos5 Octa & Intel Knights Corner \\
\hline
Processors & Intel Xeon E5-2699 v3 & 1x Samsung Exynos 5410 & 1x Xeon Phi 31S1P \\
\hline
# cores & 18 (36 with hyperthreading) & 4x Cortex A15 & 57 (without hyperthreading) \\
\hline
Core clock & 2.30 GHz & 1.66 GHz (A15) & 1.71 GHz \\
\hline
1.1 cache (per core) & 32/32 KB L1D & 32/32 KB L1D & 32/32 KB L1D \\
\hline
1.2 cache (per processor) & 256 KB × 18 (approx. 4.6 MB) & 2 MB (shared A15) & 512 KB × 57 (approx. 29.1 MB) \\
\hline
Interconnect & 2x 9.6 Gbps Quick Path Interconnect (QPI) & CoreLink Cache Coherent Intercon-nect (CCI) 400 & 5 Gbps Ring Bus Interconnect \\
\hline
Memory & 512 GB DDR4 & 2 GB LPDDR3 & 6 GB DDR5 \\
\hline
 OS & Ubuntu Linux 18.04 LTS & Ubuntu 14.04 (3.103 kernel) & Xeon Phi uOS (2.6.38.8+mpss3.4.2) \\
\hline
Compiler & GNU GCC 7.2.0 (using -O3) & GNU GCC 4.8.2 (using -O3) & Intel C Compiler (ver. 15.0.2) (-O3) \\
\hline
\end{tabular}
\caption{Testing platforms specifications.}
\end{table}

\section{Experimental evaluation}

We ran several different benchmarks to evaluate \textit{GreenBST} performance (operations per second) and energy efficiency (operations per joule). We combined the benchmark results with the last level cache (LLC) and memory profiles to draw a conclusion of whether \texttt{vEB}-based trees such as \textit{GreenBST} provide portable energy efficiency and performance across different platforms.

We evaluated \textit{GreenBST} against the prominent non-blocking and lock-based search trees in the literature (see Table \cite{1.1}), using parallel micro-benchmark suite Synchrobench \cite{27}. Their LLC and memory profiles (see Figures \cite{3.3} and \cite{8.1}) were collected to evaluate the impact of the locality-aware approaches on energy efficiency and performance (see Section \cite{7.1}).

The experimental benchmarks were conducted on an Intel high performance computing (HPC) platform, an ARM embedded platform, and an accelerator platform based on the Intel Xeon Phi architecture (MIC platform) (see Table \cite{2.3}). Scalable memory allocator jemalloc 5.1.0 was used for all the benchmarks on the HPC, ARM and MIC platforms. These benchmarks were repeated at least 5 times to guarantee consistent results.

\textit{GreenBST} is open source and available at: https://github.com/uit-agc/GreenBST.

\subsection{Benchmark setup}

We compared \textit{GreenBST} with seven concurrent search trees (see Table \cite{1.1} using parallel micro-benchmark suite Synchrobench \cite{27}. \textit{GreenBST}’s \(U\) is set to 4095 and \texttt{LYBTREE}’s order (or the maximum number of keys within a node \cite{32}) was set to 64 (or \(\log_2(\text{order}) = 6\)), the best configuration (cf. Figure \cite{4.6}). All running threads were pinned to the available logical cores using \texttt{pthread_setaffinity_np}. If the number of threads was less than the number of logical cores in the first CPU, all threads were pinned to the first CPU. Otherwise, threads were distributed evenly among available CPUs. The first thread to finish its work set a global flag, which caused each thread to terminate after its next operation. The experiments were performed in C/C++.

In order to evaluate the efficiency of \textit{GreenBST} design, we implemented SVEB, a lock-based version of the conventional cache-oblivious search tree using global mutex. Namely, concurrent accesses to the conventional sequential \texttt{vEB}-based tree \cite{11} were controlled by a global mutex.
All tree operations (i.e., search, insert, delete) used random values $v \in (0, init \times 2], v \in \mathbb{N}$ where init was the initial size of trees. The init values were chosen to make the trees partially fit into the last level cache (LLC). The init value for HPC and MIC platforms was $2^{24}$ (i.e., 64MB of keys, approximately 2 times as much as the last level cache (LLC) of the HPC and MIC platforms) and for ARM platform was $2^{20}$ (i.e., 4MB of keys, 2 times as much as the ARM platform’s LLC). Multiple threads performed insertions and deletions until the data structure reached init keys. Due to the space constraints, we present only two cases: i) 90% search and 10% update and ii) 50% search and 50% update. The benchmarks were run with different numbers of cores between the minimum and maximum available cores on the HPC and ARM and MIC platforms.

Energy efficiency metrics (operations/joule) were the number of operations divided by the energy consumption. The ARM platform was equipped with a built-in on-board power measurement system that was able to measure the energy consumption for the A15 cores, A7 cores, and memory continuously in real-time. For the Intel HPC platform, the Intel PCM [1] using built-in CPU counters was used to measure the CPU and DRAM energy consumption. Energy consumption on MIC platform was measured by polling the /sys/class/micras/power interface every 50 milliseconds. The total energy consumed by CPUs and memory system (in Joules) was measured. The measurements started after the tree initialization.

Performance metrics (operations/second) were the number of operations ($rep = 5,000,000$) divided by the maximum time for the threads to finish the whole operations.

7.2 Energy efficiency evaluation

On the Intel HPC platform, GreenBST was more energy efficient than the other trees in all cases except the case of 1 thread (see Figure 8a top bar-charts). In the case of 1 thread, SVEB was slightly more energy efficient than GreenBST because of SVEB simple concurrency control - global mutex. The global mutex, however, prevented SVEB from scaling with the number of cores while GreenBST scaled well with the number of cores. GreenBST was 40% more energy efficient than ABTREE, the best competitor, in the experiment running the 50%-search benchmark with 36 cores (see the right bar-chart). GreenBST energy-efficiency advantage over the other trees comes from the new concurrency-aware vEB layout that reduces data movement between memory levels (e.g., between LLC and DRAM as shown in Figure 8d) while supporting high concurrency. For example, as LYBTREE node is a contiguous array of sorted keys to maximize spatial locality for search operations, insert operations may need to shift many keys in order to have room for a new key, causing high data movement (cf. Figure 8f right chart). The amount of data transferred between CPU last level cache (i.e., L3-cache) and memory in GreenBST is much less than that in the other trees except SVEB (cf. Figure 8d).

On the ARM embedded platform where the A15 processor with 4 cores was used, GreenBST energy efficiency scaled well and was significantly better than those of the other trees (see Figure 8b top bar charts). Note that SVEB energy efficiency decreased significantly and became worse in the case of 4 cores. Contrarily, GreenBST scaled well with the number of cores and was 35% more energy efficient than ABTREE, the best competitor, in the experiment running the 50%-search benchmark with 4 cores (see the right bar-chart).

Note that LFBST update operations did not work on the ARM platform since it required 64-bit pointers while the ARM platform was 32-bit. As a result, LFBST was excluded from the experiments on the ARM platform.

On the Intel MIC accelerator platform, GreenBST was significantly more energy efficient than the other trees (see Figure 8c top bar-chart). GreenBST was 50% more energy efficient than LYBTREE, the best competitor, in the case of 90%-search benchmark with 57 cores.

The results on data movement between processor last-level cache (LLC) and memory on the HPC and MIC platforms provide insights into why GreenBST was more energy efficient than all the other trees in most cases (see Figures 8d). The LLC-DRAM data movements on the HPC platform and the MIC platform were collected using Intel PCM and PAPI library, respectively. GreenBST data transferred between LLC and memory was significantly less than those of the other trees across the platforms, thanks to GreenBST concurrent locality-aware layout CvEB (see Section 5.2). For example, on the HPC platform, GreenBST data transferred between LLC and memory was only half of ABTREE, the best competitor, in the 50%-search benchmark using 36 cores (see Figures 8d). Moreover, GreenBST’s memory footprint is smaller than those of LYBTREE, CITRUS, LFBST and BSTTK, four of the six non-vEB trees (see Figure 8c).

7.3 Performance evaluation

On the Intel HPC platform, GreenBST outperformed all the other trees in all the cases from 9 cores to 36 cores with both the 90%-search and 50%-search benchmarks (see Figure 8a bottom line-charts). GreenBST throughput was 40% higher than that of LFBST, the best competitor, in the case of the 50%-search benchmark using 36 cores (see the right line-chart).

On the ARM embedded platform, GreenBST significantly outperformed all the other trees in all the experiments (see Figure 8b bottom line-charts). GreenBST throughput was 60% higher than that of ABTREE, the best competitor, in the experiment running 50%-search benchmark with 4 cores (see the right chart).

On the Intel MIC accelerator platform, GreenBST outperformed all the other trees in all the experiments (see Figure 8c bottom line-chart). GreenBST throughput was 40% higher than that of LFBST, the best competitor, in the 50%-search benchmark using 36 cores (see Figure 8c bottom right line-chart).

8 DISCUSSIONS

8.1 Locality-awareness and overhead minimization

The usage of concurrency-aware vEB layouts (or CvEB) is able to reduce GreenBST energy consumption and at the same time increase GreenBST performance across the HPC, embedded and accelerator platforms. Section 7.2 has highlighted how CvEB-based search trees such as GreenBST manage to outperform their counterparts in terms of energy efficiency and performance for synthetic benchmarks on the different platforms.

One of the interesting findings is that minimizing hidden overhead (e.g., pointers in search trees) in locality-aware data structures can significantly reduce the energy consumption and increase the runtime performance. Our new optimization techniques such as removing space-overhead of pointers (see Section 5.1) significantly reduce GreenBST space overhead, thereby reducing the data transfer between memory levels (cf. Figures 8d). Moreover, the smaller embedded trees reduce the maintenance overhead for leaf GNodes because less data needs to be arranged in rebalance.
(a) **HPC platform.** GreenBST is up to 40% more energy efficient than ABTREE, the best competitor, and has up to 40% higher throughput than LYBTree, the best competitor, in the 50% search benchmark using 36 hyper-threaded cores.

(b) **ARM platform.** GreenBST is up to 35% more energy efficient than ABTREE, the best competitor, and has up to 60% higher throughput than ABTREE in the 50% search benchmark using 4 cores.

(c) **MIC platform.** GreenBST is up to 50% more energy efficient than LYBTree, the best competitor, in the 90% search benchmark using 57 cores and has up to 40% higher throughput than LFBST, the best competitor, in the 50% search benchmark using 57 cores.

(d) **Data movement** in the 50% search benchmark between CPU last level cache (LLC) and main memory (DRAM) on the HPC platform and the MIC platform.

(e) Energy efficiency and data movement between LLC and DRAM on the HPC platform for LYBTree and GreenBST with varying node size.

(f) The tree memory footprint (in GB) on the HPC platform. Tested using random keys in the range \(1, r\), with \(r = 2^{25}\).

Figure 8: 
(a,b,c) Energy efficiency and throughput comparison of the trees on the HPC, ARM and MIC platforms. 
(d) LLC-DRAM data movement on the HPC platform, collected from the CPU counters using Intel PCM, and on the MIC platform, collected using PAPI library. 
(e) Energy efficiency and data movement on the HPC platform for varying node size. 
(f) The tree memory footprint, collected using Linux’s `PMAP` command.
when the tree grows. Thanks to the fixed-size GNodes, GreenBST with fixed-size GNodes gradually expands on-demand, but also exponentially increases the tree footprint. Indeed, nodes in the concurrent exponential CO tree grow exponentially in size, which not only complicates the maintenance of inter-node pointers but also exponentially increases the tree memory footprint in practice. In contrast, the memory footprint of GreenBST with fixed-size GNodes gradually expands on-demand when the tree grows. Thanks to the fixed-size GNodes, GreenBST exploits further spatial locality by utilizing a cached map to eliminate the basic pointers overhead (see Section 3.1 for details).

8.2 Concurrency control

Some of the benchmark results show that besides data movements, efficient concurrency control is also necessary in order to devise energy-efficient data structures on multicore platforms. For example, in sequential executions (i.e., 1 core), the conventional vEB tree (SVEB) had the smallest amount of data transferred between memory and the last level cache (cf. Figure 8a and 8d) and thereby achieved the best energy efficiency (cf. Figure 8a). However, when using 2 or more cores, its energy efficiency failed to scale (cf. Figures 8a, 8b and 8c). SVEB is not designed for concurrent operations and therefore an inefficient concurrency control (i.e., a global mutex) had to be incorporated in order to include SVEB in this study. Note that we were unable to use a more fine-grained concurrency control without significantly changing SVEB data structure because SVEB uses a recursive layout fitted in a contiguous memory block (see Section 3.1). Therefore, although SVEB had the smallest amount of data transfer in sequential executions, in parallel executions the concurrent cores had to spend a lot of time waiting and competing for a lock. This is inefficient as waiting cores still consume power (e.g., static power).

8.3 Comparison with previous concurrent cache-oblivious trees

Based on experimental insights, GreenBST, a concurrent CvEB-based tree, is more efficient than previous theoretical concurrent cache-oblivious (CO) trees such as the concurrent packed-memory CO tree and concurrent exponential CO tree [9]. The concurrent packed-memory CO tree has a good amortized memory transfer cost of $\Theta(\log_B N + (\log^2 N/B))$ for tree updates, assuming that operations occur sequentially. However, the proposed data structure requires each node to have the parent-child pointers. Besides the complication in re-arranging those pointers, we have found that removing pointers from nodes to minimize memory footprint is significantly beneficial for cache-oblivious trees in practice (see Section 3.1).

In the concurrent exponential CO tree [9], expected memory transfer cost for search and update operations is $\Theta(\log_B N + (\log^2 N/B))$, assuming that all processors are synchronous. Cormen et al. [16, pp. 212], however, wrote that although exponential search tree algorithms [4] are an important theoretical breakthrough, they are fairly complicated in practice. Indeed, nodes in the concurrent exponential CO tree grow exponentially in size, which not only complicates the maintenance of inter-node pointers but also exponentially increases the tree memory footprint in practice. In contrast, the memory footprint of GreenBST with fixed-size GNodes gradually expands on-demand when the tree grows. Thanks to the fixed-size GNodes, GreenBST exploits further spatial locality by utilizing a cached map to eliminate the basic pointers overhead (see Section 5.1 for details).

9 CONCLUSIONS

The results presented in this paper provide a starting point to investigate further energy-efficient data structures and algorithms that exploit fine-grained data locality provided by ideal cache models. The results not only show that GreenBST is an energy-efficient concurrent search tree, but also provide important insights into how to develop energy efficient data structures in general. On single core systems, locality-aware data structures that can lower data movement, have been shown to be able to increase energy-efficiency. However, on multicore systems, locality-awareness alone is not enough, and good concurrency control and cache strategy are needed. Otherwise, the energy overhead of either waiting cores or interconnect-based cache coherency mechanisms can exceed the energy saving obtained by less data movement.

The CvEB-based search trees such as GreenBST are composed of tree-containers (i.e., GNodes), thereby being highly decomposable. Therefore, it is possible to extend the CvEB search trees to work with heterogeneous cores and memory systems, for example by utilizing Cosh OS abstractions [6]. Also, devising in-memory key-value stores on top of GreenBST is among our future works.

ACKNOWLEDGMENTS

This work was supported by the Research Council of Norway under projects PREAPP (grant no 231746/F20) and eX3 (grant no 270053). The evaluation was partly performed on resources provided by UNINETT Sigma2 - the National Infrastructure for High Performance Computing and Data Storage in Norway (grant no N9342K).

REFERENCES
