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 (Synchrobench). Our experimental results show that GreenBST achieves 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.

Index Terms—Concurrent data abstractions, trees, energy efficiency, performance optimization, data locality, portability.

1 Introduction

As energy efficiency is emerging as one of the most important metrics in designing computing systems [26], [41], [46], [47], 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 [1]. 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 [38]), 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], [36]) 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 [39] 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], [34], [35], [37], [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-

- *Preliminary versions of this paper appeared in the Proceedings of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS'15) [43] and the Proceedings of the 22nd International European Conference on Parallel and Distributed Computing (Euro-Par'16) [42].*

- *P. H. Ha is with the Department of Computer Science, UiT The Arctic University of Norway. This work was conducted in part while he was visiting the Computer Science Department, in the Computational Research Division at Lawrence Berkeley National Laboratory, USA. Email: phuong.hoai.ha@uit.no*
- *O. J. Anshus is with the Department of Computer Science, UiT The Arctic University of Norway. Email: otto.anshus@uit.no.*
- *I. Umar is with the Department of Computer Science, UiT The Arctic University of Norway. This work was conducted in part while he was working at the Institute of Marine Research, Norway. Email: ibrahim.umar@uit.no.*

1. In the rest of this paper, the term *fine-grained locality* is used to refer to data locality on-chip (e.g., data movement between registers, L1-/ L2-caches and last level cache (LLC)) while the term *coarse-grained locality* is used to refer to data locality off-chip (e.g., data movement between DRAM and LLC)

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], [20]. 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 3).
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 4 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[3].
4) We have provided insights into how portable fine-grained data locality actually influences energy efficiency and performance in concurrent search trees (see Section 8). 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 devise 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 3) 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)

#	Algorithm	Ref	Description	Synchronization	Affiliation	Data structure
1	SVEB	[11]	*Conventional* vEB layout search tree	global mutex	U. Aarhus	binary-tree
2	CITRUS	[5]	RCU-based search tree	lock-based	Technion	binary tree
3	LFBST	[36]	Non-blocking binary search tree	lock free	UT Dallas	binary tree
4	BSTTK	[19]	Portably scalable concurrent search tree	lock-based	EPFL	binary tree
5	LYBTREE	[32]	Concurrent B-tree (B-link tree[2])	lock-based	CMU	b-tree
6	ABTREE	[13]	Lock-free (a,b)-tree	lock-free	Technion	(a,b)-tree
7	BWTREE	[44]	Lock-free Bw-tree	lock-free	CMU	b-tree
new	**GreenBST**	-	Locality aware concurrent search tree	lock-based	**this paper**	b-tree

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].

Note that in the ideal cache model, B-tree is no longer optimal because of the unknown B. 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 U on the unknown memory block size B 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 U. Without knowing an upper bound U, we must organize data for unknown block size B 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 are 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.

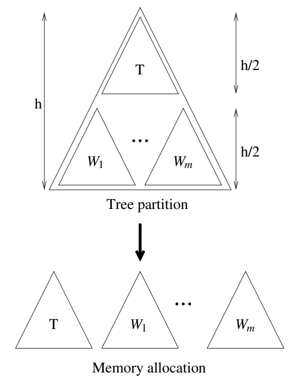


Figure 1: Static van Emde Boas (vEB) layout: a tree of height h is recursively split at height $h/2$. The top subtree T of height $h/2$ and $m=2^{h/2}$ bottom subtrees $W_1;W_2;...;W_m$ of height $h/2$ are located in contiguous memory locations where T is located before $W_1;W_2;...;W_m$.

3 Concurrency-aware vEB layout

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

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 at most U nodes (see dashed triangles of height 2^L 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.
- Let L be the level of detail of GNode. Let H be the height of GNode, we have $H = 2^L$. For simplicity, we assume $H = \log_2(U+1)$.
- N,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],

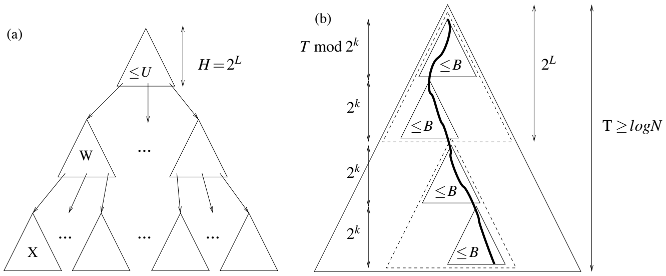


Figure 2: *(a)* The new concurrency-aware vEB layout. *(b)* A search path using the concurrency-aware vEB layout.

[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 $W_1, W_2, \cdots, W_{m_1}$ of height $h/2$. The (m_1+1) top and bottom subtrees are then located in contiguous memory locations where T is located before $W_1, W_2, \cdots, W_{m_1}$. Each of the subtrees of height $h/2$ is then laid out similarly to (m_2+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 $O(\log_B N)$ 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 B to be *known in advance*. Moreover, at any level of detail, each subtree is stored in a contiguous block of memory (e.g., subtree W_1 in Figure 1).

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 $W_2, W_3, \cdots, W_m$ because of rebalancing nodes between W_1 and $W_2, W_3, \cdots, W_m$ 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.

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 $O(\log_B N)$ 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 $S \leq U$).

Figure 2a 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 2a, 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 $k > L$ 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 $k > L$ 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]).

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 $O(\log_B N)$, 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-k} 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 2b, 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^i is stored in a contiguous block of memory, each subtree of height 2^k *within* a GNode is stored in at most 2 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 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 $2^k \geq \frac{1}{2}\log_2(B+1)$.

We have $2^{T-1} \leq N \leq 2^T$ 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 $2\lceil T/2^k \rceil \leq 4\lceil \frac{\log_2 N+1}{\log_2(B+1)} \rceil = 4\lceil \log_{B+1} N + \log_{B+1} 2 \rceil = O(\log_B N)$, where $N \geq 2$. ☐

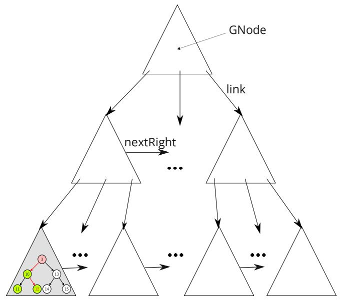


Figure 3: Illustration of the GreenBST layout.

1: **Struct** UNIVERSETPYE U_T:
2: *root*, pointer to the *root* of the topmost GNode ($T_1.root$)

3: **Struct** GNODETYPE G_T:
4: *root*, a pointer to the root NODE of the GNode
5: *nodes*$[U]$, a group of pre-allocated NODE n: $\{n_1, n_2, ..., n_U\}$
6: *link*$[U]$, an array that links the basic leaf nodes to the roots of child GNodes (see Figure 3).
7: *nextRight*, a pointer to the GNode's right sibling (see Figure 3)
8: *highKey*, the lowest key value of the right sibling GNode, default is ∞
9: *rev*, a counter used for optimistic concurrency
10: *locked*, a lock protecting the GNode for update

11: **Struct** NODETYPE N_T:
12: *value* $\in \mathbb{N}$, the node value, default is EMPTY
13: *mark* $\in$ {TRUE, FALSE}, TRUE indicates a logically deleted node, default is FALSE. ▷ fields *value* and *mark* are stored in a single word such that they can be updated atomically.

14: **Struct** MAPTYPE M_T:
15: $left \in \mathbb{N}$, *left* child's address offset
16: *right* $\in \mathbb{N}$, *right* child's address offset

Figure 4: GrenBST data structures.

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: UPDATE(v,T_G), which adds or removes value v from GreenBST T_G, and 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 1 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[4]. 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 4) contains i) field *value* holding a value for guiding the search, or a data value (or key) if it 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 UPDATE function (see Figure 6). Meanwhile, the *search* operation traverses GreenBST in a top-down, left-to-right manner using a combination of the 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 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 MOVE_RIGHT function (line 4). The MOVE_RIGHT function serializes concurrent updates by combining right-scanning (line 55) 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 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 REBALANCE(key,T_i) is responsible for rebalancing a GNode T_i after an insertion (see Figure 6, line 41). 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_y as evenly as possible (see Figure 6, line 21). The split operation also updates $T_v.nextRight$ to point to T_y (line 28) and fills $T_v.highKey$ with T_y's minimum key (line 27). Lastly, T_y's minimum key is inserted into the parent GNode of both T_v and T_y (line 33). 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_y. Note that for a GNode T_y with minimum key K_{min}, a new key less than K_{min} will be forwarded to one of T_y'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 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

4. To avoid confusion, from this point onward, we refer the "fat" nodes of GreenBST as *GNode* and the GNode's basic nodes as *basic nodes* or *nodes*.

```
1: function SEARCH(key, GreenBST, maxDepth)
2:     GNode ← GNODESEARCH(key,GreenBST,maxDepth)          ▷ Find the leaf GNode
3:     rev ← GNode.rev                                      ▷ Get revision
4:     if (GNode.highKey ≤ key) then                        ▷ Move right if necessary
5:         GNode ← GNode.nextRight                          ▷ Move to the right sibling GNode
6:         Goto 3
7:     result ← BINARYSEARCH(key,GNode)                     ▷ Search key within this GNode
8:     if (GNode.rev != rev or not even) then               ▷ If a concurrent update has occurred
9:         Goto 3                                           ▷ Re-check this GNode
10:    return result                                        ▷ Otherwise, return the search result

11: function GNODESEARCH(key, GreenBST, maxDepth)
12:    GNode ← GreenBST.root
13:    while GNode is not leaf do
14:        rev ← GNode.rev                                  ▷ Get revision
15:        if (GNode.highKey <= key) then                   ▷ Move right if necessary
16:            nextGNode ← GNode.nextRight
17:        else                                             ▷ key is located in this GNode's sub-tree
18:            bits ← 0
19:            depth ← 0
20:            p ← GNode.root
21:            base ← p
22:            link ← GNode.link
23:            repeat                                       ▷ Continue until reaching a leaf node of this GNode
24:                depth ← depth + 1                        ▷ Increment depth
25:                bits ← bits << 1                         ▷ Shift one bit to the left in each level
26:                if (key < p.key) then
27:                    p ← LEFT(p,base)                     ▷ Go left, see Figure 7
28:                else
29:                    p ← RIGHT(p,base)                    ▷ Go right, see Figure 7
30:                    bits ← bits + 1                      ▷ Right child color is 1
31:            until IsLeaf(p)
32:            bits ← bits << (maxDepth − depth)            ▷ Get link to child GNode
33:            nextGNode ← link[bits]                       ▷ Go to the child GNode
34:        if (GNode.rev != rev or not even) then           ▷ If a concurrent update has occurred
35:            Goto 14                                      ▷ Re-check this GNode
36:        GNode ← nextGNode                                ▷ Otherwise, move to the next GNode
37:    return GNode
```

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.

dominated by search and insert operations.

The *search* operation in GreenBST is a combination of function GNODESEARCH to find the associated GNode (line 2 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 6 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 4) 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 34 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

```
1: function UPDATE(type,key,GreenBST,maxDepth)
2:     GNode_key ← GNODESEARCH(key,GreenBST,maxDepth)
3:     Push to stack the last visited GNodes at each level along the search path from GreenBST.root to GNode_key.
       ▷ Arrive at a leaf GNode, now do the actual update operation
4:     GNode ← MOVE_RIGHT(key,GNode_key)                    ▷ Lock and move right, if necessary
5:     if (type = delete) then                              ▷ Delete
6:         increment(GNode.rev)
7:         MARKASDELETED(key,GNode)
8:         increment(GNode.rev)
9:         unlock(GNode)
10:    else
11:        if (type = insert) then                          ▷ Insert
12:            while (GNode is not NULL) do
13:                increment(GNode.rev)
14:                BINARYINSERT(key,GNode)                  ▷ Insert key to an appropriate basic node L and update GNode.link[] if necessary
15:                if (key is put at the last level H of GNode) then
16:                    if (GNode.isleaf and rebalance is possible) then   ▷ Rebalance, see Section 5.3 for details
17:                        REBALANCE(GNode)
18:                        increment(GNode.rev)
19:                        unlock(GNode)
20:                        return
21:                    else                                 ▷ Split
22:                        newGNode ← initialize new GNode
23:                        evenly distribute GNode's keys over GNode and newGNode where GNode keeps the lower half of the sorted keys.
24:                        y ← the lowest key of newGNode
25:                        newGNode.highKey ← GNode.highKey
26:                        newGNode.nextRight ← GNode.nextRight
27:                        GNode.highKey ← y
28:                        GNode.nextRight ← newGNode
                           ▷ No further modification on the GNode, increment its revision so that search operations can proceed
29:                        increment(GNode.rev)
30:                        oldGNode ← GNode
31:                        key ← y
32:                        link_new ← newGNode              ▷ link_new is a pointer to newGNode
33:                        GNode ← pop(stack)               ▷ Insert key and link_new to the parent GNode at line 12
34:                        GNode ← MOVE_RIGHT(key,GNode)    ▷ Lock parent GNode
35:                        unlock(oldGNode)
36:                else
37:                    increment(GNode.rev)
38:                    unlock(GNode)
39:                    return
40:    return

41: function REBALANCE(key,GNode)
42:    w ← Location of key
43:    while (depth(w) ≥ 1) do                              ▷ Depth of w from GNode.root
44:        sum ← key count in subtree rooted in w
45:        height ← height of subtree rooted in w
46:        density(w) ← sum/(2^height − 1)
47:        if (density(w) ≤ Γ_depth(w)) then
48:            balance the subtree rooted in w              ▷ See Section 5.3 for details
49:            return
50:        else
51:            w ← ancestor(w)
52:    return

53: function MOVE_RIGHT(key,GNode)
54:    lock(GNode)
55:    while (GNode.highKey ≤ key) do                       ▷ Move right if necessary
56:        rightGNode ← GNode.nextRight
57:        lock(rightGNode)
58:        unlock(GNode)
59:        GNode ← rightGNode
60:    return GNode
```

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).

operation re-checks again the counter (see line 34). If the current counter value is an odd number or if it is not equal to the recorded value, the search operation needs to retry as this indicates that the GNode is being or has been updated (see line 35).

The correctness proof of GreenBST is presented in Section 6.

5 GREENBST DESIGN IN DETAIL

To improve efficiency and reduce overhead, GreenBST is devised using several techniques, in addition to the CvEB layout (Section 3) and customized concurrency control (Section 4.3), such

```
1: MapType map[U]

2: function RIGHT(p, base)
3:    nodesize ← SIZEOF(single node)
4:    idx ← (p − base)/nodesize
5:    if (map[idx].right != 0) then
6:        return base + map[idx].right
7:    else
8:        return 0

9: function LEFT(p, base)
10:   nodesize ← SIZEOF(single node)
11:   idx ← (p − base)/nodesize
12:   if (map[idx].left != 0) then
13:       return base + map[idx].left
14:   else
15:       return 0
```

Figure 7: *Mapping* functions.

as GNode's cached map (Section 5.1), efficient inter-GNode connection (Section 5.2) and incremental rebalancing (Section 5.3).

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 4 and Figure 7). 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 [11] 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 33). The maximum size of the bit representation is GNode height or $\log_2(U)$ bits.

Particularly, each basic leaf node L of a non-leaf GNode has two child GNodes pointed by $link[L_{left}]$ and $link[L_{right}]$ where L_{left} and L_{right} are computed when the GNodeSearch function traverses GreenBST (lines 25, 30 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 rebalance 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 46 in see Figure 6). For example, a subtree with root w at level $H-3$, where H is GNode height, can hold at most $2^3 - 1$ keys. If the subtree only contains 3 keys, then *density(w)* $= {}^3/_7 = 0.42$. Let Γ_i be the *density threshold* at level i, we have $0 < \Gamma_1 < \Gamma_2 < ... < \Gamma_H = 1$, 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)* $\leq \Gamma_{depth(w)}$ 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_i has a nextRight pointer to a sibling GNode G_s, G_i contains keys less than G_i.highKey and G_s contains keys greater than or equal to G_i.highKey. If a basic leaf node BN in G_i has a left (resp. right) pointer to a child GNode G_c, G_c 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_1'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 $G_2.hightKey == G.highKey$ 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 $link_{new}$ to G_2 are assigned to G_p's basic leaf node L and L's right pointer $G_p.link[L_{right}]$, respectively (lines 31 - 34 and 14), 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 $G_p.link[L_{left}]$ 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_p (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 $G_1.nextRight$, 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 4 in Figure 6) returns (line 54 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 t_0 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 12, 16 or 33 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 21 - 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 4 in Figure 6) returns (line 54 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 3 and 8 in Figure 5). Since the UPDATE function must successfully acquire G's lock (line 4 in Figure 6) before increasing $G.rev$ (line 13), only one UPDATE function can increase $G.rev$ and therefore odd $G.rev$ indicates an on-going update. Note that if the split function interferes with GS and makes $G.rev$ even number again between lines 3 and 8 in Figure 5, GS will discover that $G.rev$ has been changed (i.e., $G.rev \neq rev$) and will then wait for the split operation to finish (line 8 in Figure 5). In this case, the linearization point of the split operation is the time point when INCREMENT($G.rev$) at line 13 in Figure 6 returns. The linearization point of the SEARCH function is the time point when the SEARCH function observes that $G.rev$ is unchanged and even at line 8 in Figure 5. ☐

Definition 6.2. *The* **subtree** *rooted at a GNode G in a well-formed GreenBST includes both the subtrees of G's child GNodes linked by G.link[] and the subtree of G's sibling GNode linked by G.nextRight (see Figure 3 for illustration)*

Lemma 6.3. *For each GNode VG visited during* GNODESEARCH(*key*,*GreenBST*,*maxDepth*) *where GreenBST is well-formed, the next GNode G made at time t (line 12, 16 or 33 in Figure 5) from the last optimistic transaction for VG (i.e., last execution of lines 14 - 34 with GNode == VG) satisfies the following claim: if key in the tree, then it is in the subtree rooted at G at time t.*

Proof. (Sketch) We will prove the lemma inductively on G. Let $G_1, G_2, \cdots, G_n$ be the path from the root GNode G_1 to a leaf GNode G_n visited by a GNODESEARCH function GS.

The lemma holds when G is the root GNode, i.e. $G = G_1$ (line 12).

Assume that lemma holds for $G = G_k$, we will prove that the lemma holds for $G = G_{k+1}$.

Indeed, as the lemma holds for $G = G_k$, GS will visit G_k in the next iteration to find *key* (line 13 - 36). We will prove that when visiting G_k, GS locates a correct child/ sibling GNode G_{k+1} whose subtree contains *key* (if *key* exists).

Case 1: no interference on G_k from UPDATE functions (i.e., no interference between line 14 and line 34 when $GNode = G_k$).

If $key < G_k.highKey$, GS traverses G_k'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 33). 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 $key \geq G_k.highKey$, *key*, if in the tree, was moved to one of subtrees rooted at G_k's sibling GNodes between the time GS got a reference to G_k (line 12 or 36) and the time GS started to access G_k (line 14). In this case, the next GNode G_{k+1} is G_k's 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 - 37 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 34 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). ☐

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 13), *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', $L.highKey \leq key < L'.highKey$ according to the split operation (lines 21 - 27 in Figure 6). 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 $L \rightarrow L_1 \rightarrow \cdots \rightarrow L_k$ (see Lemma 6.2), we have *key* located in sibling L_i where $L_{i-1}.highKey \leq key < L_i.highKey$. ☐

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

Proof. (Sketch) We will prove that the result returned by the SEARCH(*key*,*GreenBST*,*maxDepath*) function in Figure 5 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 5) 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 the UPDATE function since time t_0. In this case, S returns the the same result as does conventional sequential binary search (line 7). In this case, the linearization point of the SEARCH function is the linearization point of the GNODESEARCH function invoked at line 2 (cf. Lemma 6.3).

Case 2: interference without splitting on cG from an UPDATE function U since time t_0. In this case, similar to the atomicity proof of Lemma 6.3, S will discover the interference and wait for the update to finish before actually accessing cG, thanks to counter $cG.rev$ (lines 3 and 8). As S's eventual access to cG is not interfered with any update, S returns the the same result as does conventional sequential binary search (line 7). In this case, the linearization point of the SEARCH function is the time point when the function observes that $cG.rev$ is unchanged and even at line 8 in Figure 5. The linearization point of the UPDATE function U is the time point when INCREMENT($cG.rev$) returns (Figure 6: line 6 in the case of deletion and line 13 in the case of insertion).

Case 3: interference with splitting on cG from an UPDATE function U since time t_0. In this case, according to Lemmas 6.2

and 6.5, S will find the correct right sibling sG of cG where *key*, if existing, is located, by following *nextRight* pointers and checking if $key < sG.highKey$ (lines 4 - 6). Note that S will always find a sG satisfying $key < sG.highKey$ since cG's last (rightmost) sibling lG has $lG.highKey = \infty$ (line 8 in Figure 4). This case becomes Case 2 for the sibling GNode sG. □

Lemma 6.7. *The* UPDATE *function is correct.*

Proof. (Sketch) We will prove that the UPDATE(*key*, *GreenBST*, *maxDepth*) function in Figure 6 has the same effect as does the update operation of the conventional sequential binary search trees, even when other UPDATE functions are interfering.

Let t_0 be the time at which a reference to the leaf GNode lG returned by the GNODESEARCH function (line 2 in Figure 6), 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 U is visiting. Initially, cG is lG.

Case 1: no interference on cG from another 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 14 in Figure 6) or marks as *deleted* the basic node containing *key* in the case of deletion (line 7), 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 6.1). In this case, the linearization point of the UPDATE function is the linearization point of of the GNODESEARCH function invoked at line 2 (cf. Lemma 6.3).

Case 2: interference without splitting on cG from other UPDATE functions since time t_0. As U has successfully locked cG (line 4) before performing any insertion or deletion, other concurrent update functions on cG must wait for U to finish its modification and unlock cG (line 9, 19, 35 or 38). 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 UPDATE function is the time point when LOCK(G) in the MOVE_RIGHT function invoked at line 4 returns (line 54 or 57).

Case 3: interference with splitting on cG from other UPDATE functions since time t_0. In this case, according to Lemmas 6.2 and 6.5, U will find the correct right sibling sG of cG where *key* should be located, by following *nextRight* pointers and checking if $key < sG.highKey$ in the MOVE_RIGHT function invoked at line 4 (lines 55 - 59). This case becomes Case 2 for the sibling GNode sG. □

Lemma 6.8. *GreenBST is deadlock-free.*

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

Indeed, for GNodes located at different tree levels, the 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 33 - 35 in Figure 6). For GNodes located at the same tree level, the UPDATE function locks them from left to right in the MOVE_RIGHT function by following the *nextRight* pointers (lines 54 - 57). □

Name	**HPC**	**ARM**	**MIC**
System	Intel Haswell-EP	Samsung Exynos5 Octa	Intel Knights Corner
Processors	Intel Xeon E5-2699 v3	1x Samsung Exynos 5410	1x Xeon Phi 31S1P
# cores	18 (36 with hyperthreading)	– 4x Cortex A15 – 4x Cortex A7	57 (without hyperthreading)
Core clock	2.30 GHz	– 1.6 GHz (A15) – 1.2 GHz (A7)	1.1 GHz
L1 cache (per core)	32/32 KB I/D	32/32 KB I/D	32/32 KB I/D
L2 cache (per processor)	256 KB × 18 (approx. 4.6 MB)	2 MB (shared A15) 512 KB (shared A7)	512 KB × 57 (approx. 29.1 MB)
L3 cache	45 MB (shared)	-	-
Interconnect	2x 9.6 GT/s Quick Path Interconnect (QPI)	CoreLink Cache Coherent Interconnect (CCI) 400	5 GT/s Ring Bus Interconnect
Memory	512 GB DDR4	2 GB LPDDR3	6 GB GDDR5
OS	Ubuntu Linux 18.04 LTS	Ubuntu 14.04 (3.4.103 kernel)	Xeon Phi uOS (2.6.38.8+mpss3.4.2)
Compiler	GNU GCC 7.2.0 (using -O3)	GNU GCC 4.8.2 (using -O3)	Intel C Compiler (ver. 15.0.2) (-O3)

Table 2: Testing platforms specifications.

7 EXPERIMENTAL EVALUATION

We ran several different benchmarks to evaluate 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 CvEB-based trees such as GreenBST provide portable energy efficiency and performance across different platforms.

We evaluated GreenBST against the prominent non-blocking and lock-based search trees in the literature (see Table 1), using parallel micro-benchmark suite Synchrobench [27]. Their LLC and memory profiles (see Figures 8d and 8f) were collected to evaluate the impact of the locality-aware approaches on energy efficiency and performance (see Section 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 2). 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.

GreenBST is open source and available at: *https://github.com/uit-agc/GreenBST*.

7.1 Benchmark setup

We compared GreenBST with seven concurrent search trees (see Table 1) using parallel micro-benchmark suite Synchrobench [27]. GNode's U is set to 4095 and LYBTREE's order (or the maximum number of keys within a node [32]) was set to 64 (or $\log_2(order) = 6$), the best configuration (cf. Figure 8e). All running threads were pinned to the available logical cores using `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 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 vEB-based tree [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 8e, 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 3.2. 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 8f).

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 LYBTREE, 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).

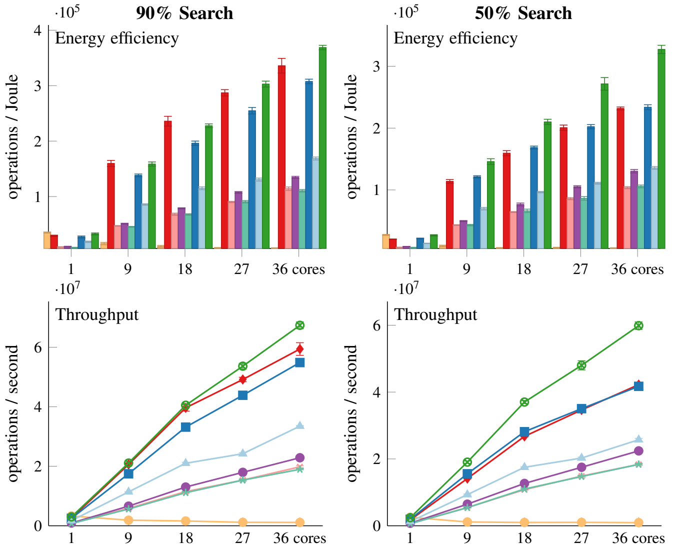
On the **Intel MIC accelerator platform**, GreenBST outperformed all the other trees in all the experiments (see Figure 8c, bottom line-chart). GreenBST throughput was 40% higher than that of LFBST, the best competitor, in the 50%-search benchmark using 57 cores (see Figure 8c, bottom right line-chart).

8 Discussions

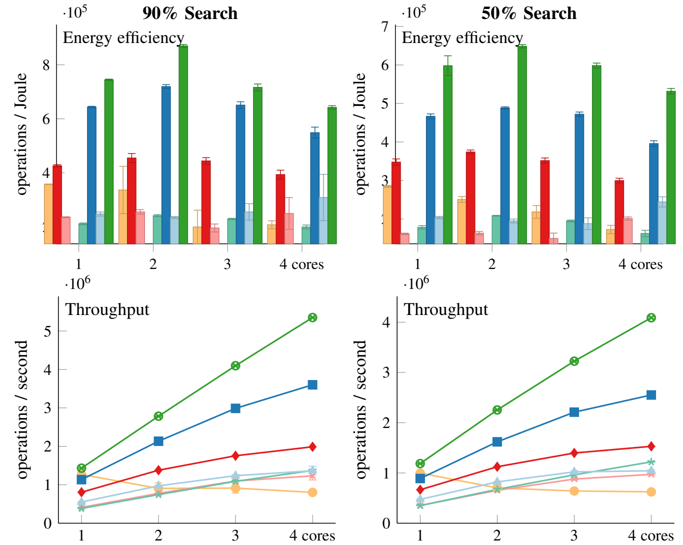
8.1 Locality-awareness and overhead minimization

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 7.1 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.

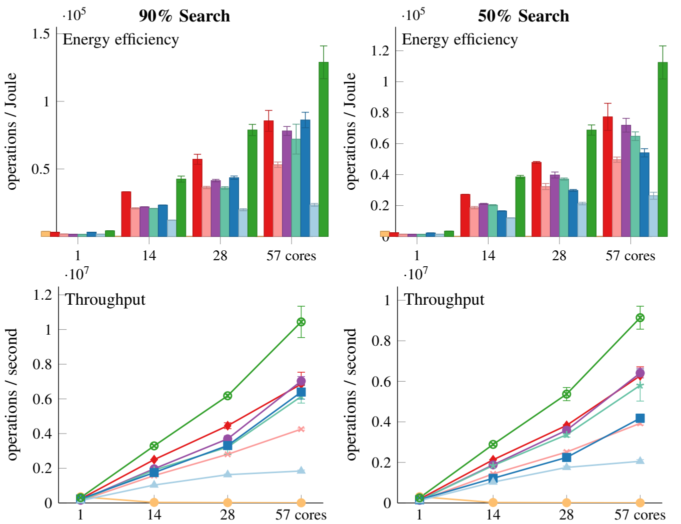
One of the interesting findings is that minimizing *hidden* overhead (e.g., pointers in search trees) in locality-aware data structures can significantly reduce the energy consumption and increase the runtime performance. Our new optimization techniques such as removing space-overhead of pointers (see 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



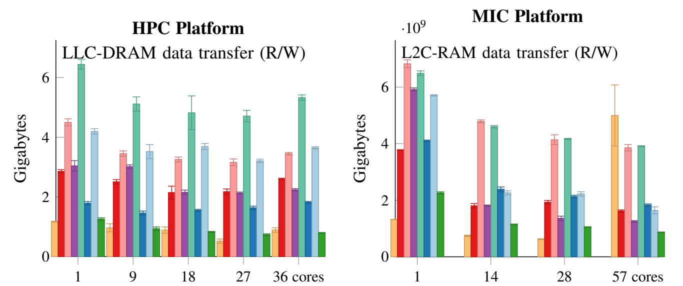
(a) **HPC platform**. GreenBST is up to 40% more energy efficient than ABTREE, the best competitor, and has up to 40% higher throughput than LYBTREE, the best competitor, in the 50% search benchmark using 36 hyper-threaded cores.



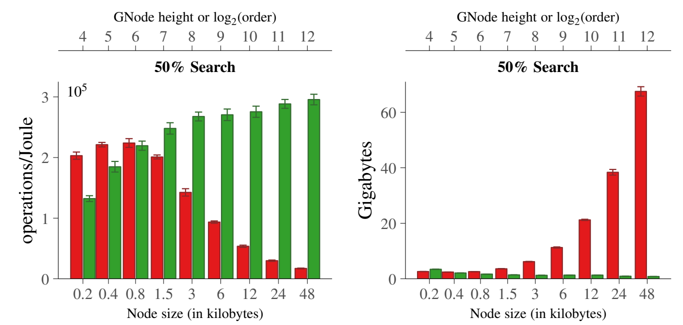
(b) **ARM platform**. GreenBST is up to 35% more energy efficient than ABTREE, the best competitor, and has up to 60% higher throughput than ABTREE in the 50% search benchmark using 4 cores.



(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**.

Tree name / Scenario	SVEB	LYBTree	citrus	LFBST	BSTTK	abtree	bwtree	**GreenBST**
Insert 50% $\times r$ keys, then	0.38	0.83	1.51	1.30	3.98	0.42	0.32	**0.57**
Delete 90% of the keys	0.38	0.83	1.65	1.30	4.00	0.42	0.33	**0.57**

(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.

or split operations. The other optimization techniques such as incremental rebalancing (see Section 5.3) significantly reduce GreenBST maintenance overhead (e.g., rebalancing overhead) (see Section 3 in [42] for performance comparison between GreenBST and its variation without the optimization called DeltaTree).

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 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 8.1).

In the concurrent exponential CO tree [9], expected memory transfer cost for search and update operations is $\mathcal{O}(\log_B N + (\log_\alpha \lg N))$, 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 n° 231746/F20) and eX3 (grant n° 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 n° NN9342K).

References

[1] Intel pcm. http://www.intel.com/software/pcm. URL http://www.intel.com/software/pcm

[2] Afek, Y., Kaplan, H., Korenfeld, B., Morrison, A., Tarjan, R.E.: Cbtree: a practical concurrent self-adjusting search tree. In: Proc. 26th international Conf. Distributed Computing, DISC'12, pp. 1–15 (2012). DOI 10.1007/978-3-642-33651-5

[3] Aggarwal, A., Vitter Jeffrey, S.: The input/output complexity of sorting and related problems. Commun. ACM **31**(9), 1116–1127 (1988)

[4] Andersson, A.: Faster deterministic sorting and searching in linear space. In: Foundations of Computer Science, 1996. Proceedings., 37th Annual Symposium on, pp. 135–141 (1996). DOI 10.1109/SFCS.1996.548472

[5] Arbel, M., Attiya, H.: Concurrent updates with rcu: Search tree as an example. In: Procs. of the ACM Symp. on Principles of Distributed Computing, pp. 196–205 (2014)

[6] Baumann, A., Hawblitzel, C., Kourtis, K., Harris, T., Roscoe, T.: Cosh: Clear os data sharing in an incoherent world. In: 2014 Conference on Timely Results in Operating Systems (TRIOS 14). USENIX Association (2014). URL https://www.usenix.org/conference/trios14/technical-sessions/presentation/baumann

[7] Bender, M., Demaine, E.D., Farach-Colton, M.: Cache-oblivious b-trees. SIAM Journal on Computing **35**, 341 (2005)

[8] Bender, M.A., Farach-Colton, M., Fineman, J.T., Fogel, Y.R., Kuszmaul, B.C., Nelson, J.: Cache-oblivious streaming b-trees. In: Proc. 19th annual ACM Symp. Parallel algorithms and architectures, SPAA '07, pp. 81–92 (2007)

[9] Bender, M.A., Fineman, J.T., Gilbert, S., Kuszmaul, B.C.: Concurrent cache-oblivious b-trees. In: Proc. 17th annual ACM Symp. Parallelism in algorithms and architectures, SPAA '05, pp. 228–237 (2005)

[10] Brodal, G.: Cache-oblivious algorithms and data structures. In: T. Hagerup, J. Katajainen (eds.) Algorithm Theory - SWAT 2004, *Lecture Notes in Computer Science*, vol. 3111, pp. 3–13. Springer Berlin Heidelberg (2004). DOI 10.1007/978-3-540-27810-8_2. URL http://dx.doi.org/10.1007/978-3-540-27810-8_2

[11] Brodal, G.S., Fagerberg, R., Jacob, R.: Cache oblivious search trees via binary trees of small height. In: Proc. 13th annual ACM-SIAM Symp. Discrete algorithms, SODA '02, pp. 39–48 (2002)

[12] Bronson, N.G., Casper, J., Chafi, H., Olukotun, K.: A practical concurrent binary search tree. In: Proc. 15th ACM SIGPLAN Symp. Principles and Practice of Parallel Programming, PPoPP '10, pp. 257–268 (2010)

[13] Brown, T.: A template for implementing fast lock-free trees using htm. In: Proceedings of the ACM Symposium on Principles of Distributed Computing, PODC, pp. 293–302 (2017)

[14] Brown, T., Helga, J.: Non-blocking k-ary search trees. In: Proc. 15th international Conf. Principles of Distributed Systems, OPODIS'11, pp. 207–221 (2011)

[15] Cha, S.K., Hwang, S., Kim, K., Kwon, K.: Cache-conscious concurrency control of main-memory indexes on shared-memory multiprocessor systems. In: Proceedings of the 27th International Conference on Very Large Data Bases, VLDB '01, pp. 181–190. Morgan Kaufmann Publishers Inc. (2001). URL http://dl.acm.org/citation.cfm?id=645927.672375

[16] Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms, Third Edition, 3rd edn. The MIT Press (2009)

[17] Crain, T., Gramoli, V., Raynal, M.: A speculation-friendly binary search tree. In: Proc. 17th ACM SIGPLAN Symp. Principles and Practice of Parallel Programming, PPoPP '12, pp. 161–170 (2012). DOI 10.1145/2145816.2145837

[18] Dally, B.: Power and programmability: The challenges of exascale computing. In: DoE Arch-I presentation (2011)

[19] David, T., Guerraoui, R., Trigonakis, V.: Asynchronized concurrency: The secret to scaling concurrent search data structures. In: Procs. of the Intl. Conf. on Architectural Support for Programming Languages and Operating Systems, pp. 631–644 (2015)

[20] Demaine, E.D.: Cache-oblivious algorithms and data structures (2002). DOI 10.1.1.154.3942

[21] Dice, D., Shalev, O., Shavit, N.: Transactional locking ii. In: Proc. 20th international Conf. Distributed Computing, DISC'06, pp. 194–208 (2006)

[22] Drachsler, D., Vechev, M., Yahav, E.: Practical concurrent binary search trees via logical ordering. In: Proc. 19th ACM SIGPLAN Symp. Principles and Practice of Parallel Programming, PPoPP '14, pp. 343–356 (2014). DOI 10.1145/2555243.2555269. URL http://doi.acm.org/10.1145/2555243.2555269

[23] Ellen, F., Fatourou, P., Ruppert, E., van Breugel, F.: Non-blocking binary search trees. In: Proc. 29th ACM SIGACT-SIGOPS Symp. Principles of distributed computing, PODC '10, pp. 131–140 (2010)

[24] van Emde Boas, P.: Preserving order in a forest in less than logarithmic time. In: Proc. 16th Annual Symp. Foundations of Computer Science, SFCS '75, pp. 75–84 (1975). DOI 10.1109/SFCS.1975.26

[25] Frigo, M., Leiserson, C.E., Prokop, H., Ramachandran, S.: Cache-oblivious algorithms. In: Proc. 40th Annual Symp. Foundations of Computer Science, FOCS '99, p. 285 (1999)

[26] Gandhi, A., Gupta, V., Harchol-Balter, M., Kozuch, M.: Optimality analysis of energy-performance trade-off for server farm management. Performance Evaluation **67**(11), 1155–1171 (2010)

[27] Gramoli, V.: More than you ever wanted to know about synchronization: Synchrobench, measuring the impact of the synchronization on concurrent algorithms. In: Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2015, pp. 1–10 (2015). DOI 10.1145/2688500.2688501. URL http://doi.acm.org/10.1145/2688500.2688501

[28] Herlihy, M.: A methodology for implementing highly concurrent data objects. ACM Transactions on Programming Languages and Systems **15**(5), 745–770 (1993)

[29] Hipp, D.R.: Sqlite. Computer Software (2015). URL http://www.sqlite.org

[30] Kim, C., Chhugani, J., Satish, N., Sedlar, E., Nguyen, A.D., Kaldewey, T., Lee, V.W., Brandt, S.A., Dubey, P.: Fast: fast architecture sensitive tree search on modern cpus and gpus. In: Proc. 2010 ACM SIGMOD Intl. Conf. Management of data, SIGMOD '10, pp. 339–350 (2010)

[31] Kung, H.T., Robinson, J.T.: On optimistic methods for concurrency control. ACM Trans. Database Syst. **6**(2), 213–226 (1981)

[32] Lehman, P.L., Yao, s.B.: Efficient locking for concurrent operations on b-trees. ACM Trans. Database Syst. **6**(4), 650–670 (1981). DOI 10.1145/319628.319663. URL http://doi.acm.org/10.1145/319628.319663

[33] Leis, V., Kemper, A., Neumann, T.: The adaptive radix tree: Artful indexing for main-memory databases. In: Proceedings of the 2013 IEEE International Conference on Data Engineering (ICDE 2013), ICDE '13, pp. 38–49. IEEE Computer Society (2013). DOI 10.1109/ICDE.2013.6544812. URL http://dx.doi.org/10.1109/ICDE.2013.6544812

[34] Lim, H., Han, D., Andersen, D.G., Kaminsky, M.: Mica: A holistic approach to fast in-memory key-value storage. In: Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation, NSDI'14, pp. 429–444. USENIX Association (2014). URL http://dl.acm.org/citation.cfm?id=2616448.2616488

[35] Mao, Y., Kohler, E., Morris, R.T.: Cache craftiness for fast multicore key-value storage. In: Proc. 7th ACM European Conference on Computer Systems, EuroSys '12, pp. 183–196 (2012). DOI 10.1145/2168836.2168855. URL http://doi.acm.org/10.1145/2168836.2168855

[36] Natarajan, A., Mittal, N.: Fast concurrent lock-free binary search trees. In: Proc. 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '14, pp. 317–328 (2014). DOI 10.1145/2555243.2555256. URL http://doi.acm.org/10.1145/2555243.2555256

[37] Rao, J., Ross, K.A.: Making b+- trees cache conscious in main memory. SIGMOD Rec. **29**(2), 475–486 (2000). DOI 10.1145/335191.335449. URL http://doi.acm.org/10.1145/335191.335449

[38] Rodeh, O.: B-trees, shadowing, and clones. Trans. Storage **3**(4), 2:1–2:27 (2008). DOI 10.1145/1326542.1326544. URL http://doi.acm.org/10.1145/1326542.1326544

[39] Sewall, J., Chhugani, J., Kim, C., Satish, N.R., Dubey, P.: Palm: Parallel architecture-friendly latch-free modifications to b+ trees on many-core processors. Proc. VLDB Endowment **4**(11), 795–806 (2011)

[40] Sleator, D.D., Tarjan, R.E.: Amortized efficiency of list update and paging rules. Commun. ACM **28**(2), 202–208 (1985). DOI 10.1145/2786.2793. URL http://doi.acm.org/10.1145/2786.2793

[41] Tiwari, D., Vazhkudai, S.S., Kim, Y., Ma, X., Boboila, S., Desnoyers, P.J.: Reducing data movement costs using energy efficient, active computation on ssd. In: Proceedings of the 2012 USENIX Conference on Power-Aware Computing and Systems, HotPower'12 (2012). URL http://dl.acm.org/citation.cfm?id=2387869.2387873

[42] Umar, I., Anshus, O., Ha, P.: Greenbst: Energy-efficient concurrent search tree. In: Procs. of the Intl. Conf. on Parallel and Distributed Computing, pp. 502–517 (2016)

[43] Umar, I., Anshus, O.J., Ha, P.H.: Deltatree: A locality-aware concurrent search tree. In: Procs of the ACM SIGMETRICS Intl. Conf. on Measurement and Modeling of Computer Systems, pp. 457–458 (2015)

[44] Wang, Z., Pavlo, A., Lim, H., Leis, V., Zhang, H., Kaminsky, M., Andersen, D.G.: Building a bw-tree takes more than just buzz words. In: Proc. of the 2018 International Conference on Management of Data, SIGMOD, pp. 473–488 (2018)

[45] Wang, Z., Qian, H., Li, J., Chen, H.: Using restricted transactional memory to build a scalable in-memory database. In: Proceedings of the Ninth European Conference on Computer Systems, EuroSys '14, pp. 26:1–26:15 (2014). DOI 10.1145/2592798.2592815. URL http://doi.acm.org/10.1145/2592798.2592815

[46] Wierman, A., Andrew, L.L.H., Tang, A.: Power-aware speed scaling in processor sharing systems: Optimality and robustness. Perform. Eval. **69**(12), 601–622 (2012). DOI 10.1016/j.peva.2012.07.002. URL http://dx.doi.org/10.1016/j.peva.2012.07.002

[47] Yao, Y., Huang, L., Sharma, A., Golubchik, L., Neely, M.: Power cost reduction in distributed data centers: A two-time-scale approach for delay tolerant workloads. IEEE Trans. Parallel Distrib. Syst. **25**(1), 200–211 (2014). DOI 10.1109/TPDS.2012.341. URL http://dx.doi.org/10.1109/TPDS.2012.341

Phuong H. Ha received the Ph.D. degree from Chalmers University of Technology, Sweden. His current research interests include energy-efficient computing, parallel programming and distributed computing systems. Currently, he is an associate professor at the Department of Computer Science, UiT The Arctic University of Norway. (www.cs.uit.no/$\sim$phuong).

Otto J. Anshus is a professor of computer science at UiT The Arctic University of Norway. His research interests include operating systems, parallel and distributed architectures and systems, and human-computer interfaces. He is a member of the IEEE Computer Society, the ACM, and the Norwegian Computer Society. Contact him at otto.anshus@uit.no.

Ibrahim Umar is a PhD student at the Department of Computer Science, UiT The Arctic University of Norway. His current research interests include energy-efficient computing, concurrent data abstractions and parallel programming.