GreenBST: Energy-Efficient Concurrent Search Tree

Ibrahim Umar(✉), Otto Anshus, and Phuong Ha

Department of Computer Science
UiT The Arctic University of Norway
{ibrahim.umar, phuong.hoai.ha, otto.anshus}@uit.no

Abstract. Like other fundamental abstractions for energy-efficient computing, search trees need to support both high concurrency and fine-grained data locality. However, existing 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 the non-blocking binary search trees do not consider data locality.

We present GreenBST, a practical energy-efficient concurrent search tree that supports fine-grained data locality as vEB-based trees do, but unlike vEB-based trees, GreenBST supports high concurrency. GreenBST is a k-ary leaf-oriented tree of GNodes where each GNode is a fixed size tree-container with the van Emde Boas layout. As a result, GreenBST minimizes data transfer between memory levels while supporting highly concurrent (update) operations. Our experimental evaluation using the recent implementation of non-blocking binary search trees, highly concurrent B-trees, conventional vEB trees, as well as the portably scalable concurrent trees shows that GreenBST is efficient: its energy efficiency (in operations/Joule) and throughput (in operations/second) are up to 65% and 69% higher, respectively, than the other trees on a high performance computing (HPC) platform (Intel Xeon), an embedded platform (ARM), and an accelerator platform (Intel Xeon Phi). The results also provide insights into how to develop energy-efficient data structures in general.

1 Introduction

Recent researches have suggested that the energy consumption of future computing systems will be dominated by the cost of data movement [12, 34, 35]. It is predicted that for 10nm technology chips, the energy required between accessing data in nearby on-chip memory and accessing data across the chip, will differ as much as 75× (2pJ versus 150pJ), whereas the energy required between accessing on-chip data and accessing off-chip data will only differ 2× (150pJ versus 300pJ) [12]. Therefore, in order to construct energy-efficient software systems, data structures and algorithms must not only be concerned with whether the data is on-chip (e.g., in cache) or not (e.g., in DRAM), but must consider also data locality in *finer-granularity*: where the data is located on the chip.

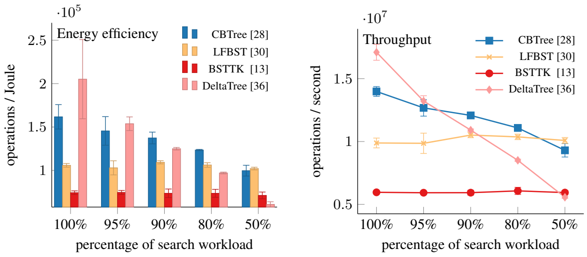


Fig. 1: Result of 5 millions tree operations of decreasing search percentage workloads using 12 cores (1 CPU). DeltaTree's energy efficiency and throughput are lower than the other concurrent search trees after 95% search workload on a dual Intel Xeon E5-2650Lv3 CPU system with 64GB RAM.

Concurrent search trees are crucial data structures that are widely used as a backend in many important systems such as databases (e.g., SQLite [24]), filesystems (e.g., Btrfs [32]), and schedulers (e.g., Linux's Completely Fair Scheduler (CFS)), among others. These important systems can access and organize data in a more energy efficient manner by adopting the energy-efficient concurrent search trees as their backend structures.

Devising fine-grained data locality layout for concurrent search trees is challenging, mainly because of the trade-offs needed: (i) a platform-specific locality optimization might not be *portable* (i.e., not work on different platforms while there are big interests of concurrent data structures for unconventional platforms [18, 21]), (ii) the usage of transactional memory [20, 23] and multi-word synchronization [19, 22, 27] complicates locality because each core in a CPU needs to consistently track read and write operations that are performed by other cores, and (iii) fine-grained locality-aware layouts (e.g., van Emde Boas layout) poorly support concurrent update operations. Some of the fine-grained locality-aware search trees such as Intel Fast [25] and Palm [33] are optimized for a specific platform. Concurrent B-trees (e.g., B-link tree [28]) only perform well if their B size is optimal. Highly concurrent search trees such as non-blocking concurrent search trees [14, 30] and Software Transactional Memory (STM)-based search trees [1, 11], however, do not take into account fine-grained data locality.

Fine-grained data locality for *sequential* search trees can be theoretically achieved using the van Emde Boas (vEB) layout [15, 31], which is analyzed using cache-oblivious (CO) models [16]. 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 cache size and cache-line length, in order to optimize its data transfer complexity, assuming that the optimal off-line cache replacement strategy is used. If a cache-oblivious algorithm is optimal for an arbitrary two-level memory, the algorithm is also asymptotically optimal for any adjacent pair of available levels of the memory hierarchy [9]. Therefore, cache-oblivious algorithms are expected to be locality-optimized irrespective of variations in memory hierarchies, enabling less data transfer between memory levels and thereby saving energy.

However, the throughput of a vEB-based tree when doing *concurrent* updates is lower compared to when it is doing *sequential* updates. Inserting or deleting a node may result in relocating a large part of the tree in order to maintain the vEB layout. Solutions to this problem have been proposed [7]. The first proposed solution's structure requires each node to have parent-child pointers. Update operations may result in updating the pointers. Pointers will also increase the tree memory footprint. The second proposed solution uses the exponential tree algorithm [3]. Although the exponential tree is an important theoretical breakthrough, it is complex [10]. The exponential tree grows exponentially in size, which not only complicates maintaining its inter-node pointers, but also exponentially increases the tree's memory footprint. Recently, we have proposed a *concurrency-aware vEB layout* [36], which has a higher throughput when doing *concurrent* updates compared to when it is doing *sequential* updates. In the same study, we have proposed DeltaTree, a B+tree that uses the concurrency-aware vEB layout. We have documented that the concurrency-aware vEB layout can improve DeltaTree's *concurrent* search and update throughput over a concurrent B+tree [36].

Nevertheless, we find DeltaTree's throughput and energy efficiency are lower than the state-of-the-art concurrent search trees (e.g., the portably scalable search tree [13]) for the update-intensive workloads (cf. Figure 1). Our investigation reveals that the cost of DeltaTree's runtime maintenance (i.e., rebalancing the nodes) dominates the execution time. However, reducing the frequency of the runtime maintenance lowers DeltaTree's energy efficiency and throughput for the search-intensive workloads, because DeltaTree nodes will then be sparsely populated and frequently imbalanced. Note that DeltaTree energy efficiency and throughput are already optimized for the search intensive workloads [36, 37].

In this paper, we present *GreenBST*, an energy-efficient concurrent search tree that is more energy efficient and has higher throughput for both the concurrent search- and update-intensive workloads than the other concurrent search trees (cf. Table 1). GreenBST applies two significant improvements on DeltaTree in order to lower the cost of the tree runtime maintenance and reduce the tree memory footprint. First, unlike DeltaTree, GreenBST rebalances incrementally (i.e., fine-grained node rebalancing). In DeltaTree, the rebalance procedure has to rebalance *all* the keys within a node and the frequency of rebalancing cannot be lowered as they are necessary to keep DeltaTree in good shape (i.e., keeping DeltaTree's height low and its nodes are densely populated). Incremental rebalance makes the overall cost of each rebalance in GreenBST lower than DeltaTree. Second, we reduce the tree memory footprint by using a different layout for GreenBST's leaf nodes (*heterogeneous* layout). Reduction in the memory footprint also reduces GreenBST's data transfer, which consequently increases the tree's energy efficiency and throughput in both update- and search- intensive workloads. We will show that with these improvements, GreenBST can become up to 195% more energy efficient than DeltaTree (cf. Section 3).

We evaluate GreenBST's energy efficiency (in operations/Joule) and throughput (in operations/second) against six prominent concurrent search trees (cf. Table 1) using parallel micro-benchmarks *Synchrobench* [17] and STAMP database benchmark *Vacation* [29] (cf. Section 3). We present memory and cache profile data to provide insights into what make GreenBST energy efficient (cf. Section

#	Algorithm	Ref	Description	Synchronization	Code authors	Data structure
1	SVEB	[8]	*Conventional* vEB layout search tree	global mutex	U. Aarhus	binary-tree
2	CBTree	[28]	Concurrent B-tree (B-link tree)	lock-based	U. Tromsø	b+tree
3	Citrus	[4]	RCU-based search tree	lock-based	Technion	binary tree
4	LFBST	[30]	Non-blocking binary search tree	lock free	UT Dallas	binary tree
5	BSTTK	[13]	Portably scalable concurrent search tree	lock-based	EPFL	binary tree
6	DeltaTree	[36]	Locality aware concurrent search tree	lock-based	U. Tromsø	b+tree
7	**GreenBST**	-	Improved locality aware concurrent search tree	lock-based	this paper	b+tree

Table 1: List of the evaluated concurrent search tree algorithms.

3). We also provide insights into what are the key ingredients for developing energy-efficient data structures in general (cf. Section 4).

Our contributions. Our contributions are threefold:

1. We have devised a new *portable fine-grained locality-aware* concurrent search trees, *GreenBST* (cf. Section 2.1). GreenBST are based on our proposed concurrency-aware vEB layout [36] with the two improvements, namely the incremental node rebalance and the heterogeneous node layouts.
2. We have evaluated GreenBST throughput (in operations/second) and energy efficiency (in operations/Joule) with six prominent concurrent search trees (cf. Table 1) on three different platforms (cf. Section 3). We show that compared to the state of the art concurrent search trees, GreenBST has the best energy efficiency and throughput across different platforms for most of the concurrent search- and update- intensive workloads.
   GreenBST code and evaluation benchmarks are available at: https://github.com/uit-agc/GreenBST.
3. We have provided insights into how to develop energy-efficient data structures in general (cf. Section 4).

2 Design overview

We devise GreenBST based on the concurrency-aware vEB layout [36], based on the idea that the layout has the same data transfer efficiency between two memory levels as the *conventional* sequential vEB layout [15, 31]. Therefore, theoretically, we can use the concurrency-aware layout within a *concurrent* search tree to minimize data movements between memory levels, which can eventually be a basis of an energy-efficient concurrent search tree. This section starts with brief descriptions about the original vEB layout and the concurrency-aware vEB layout for concurrent search tree, followed by detailed description of GreenBST structure and algorithms.

The van Emde Boas (vEB) layout. The vEB layout has inspired several cache-oblivious (CO) search trees such as the concurrent CO B-trees [5, 6] and the CO binary trees [8]. The vEB layout based trees *recursively* arrange related

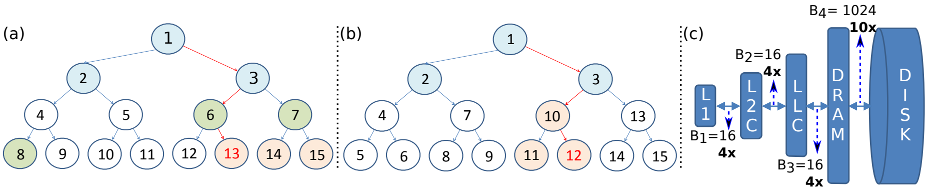


Fig. 2: Illustration of the required data block transfer in searching for (a) key 13 in BFS tree and (b) key 12 in vEB tree, where a node's value is *its address in the physical memory*. Note that in (b), adjacent nodes are grouped together (e.g., (1,2,3) and (10,11,12)) because of the *recursive* tree building. The similarly colored nodes indicates a single block transfer B. An example of multi-level memory is shown in (c), where B_x is the *block transfer* size B between levels of memory.

data in contiguous memory locations, minimizing data transfer between any two adjacent levels of the memory hierarchy.

Figure 2 illustrates the vEB layout, where B size is 3. B is the data block transfer between two memory levels (e.g., RAM and disk) in the I/O model [2]. Traversing a complete binary tree with the Breadth First Search layout (or BFS tree for short) with height 4 will need three data block transfers to locate the key at leaf-node 13 (cf. Figure 2a). The first two levels with three nodes (1, 2, 3) fit within a single block transfer while the next two levels need to be loaded in two separate block transfers that contain nodes (6, 7, 8)[1] and nodes (13, 14, 15), respectively. Generally, the number of data block transfers for a BFS tree of size N is $(\log_2 N - \log_2 B) = \log_2 N/B \sim \log_2 N$ for $N \gg B$.

For a vEB tree with the same height, the required block transfers is only two. As shown in Figure 2b, locating the key in leaf-node 12 requires only a transfer of nodes (1, 2, 3), followed by a transfer of nodes (10, 11, 12). Generally, the data transfer (or I/O) complexity of searching for a key in a tree of size N is now reduced to $\frac{\log_2 N}{\log_2 B} = \log_B N$, simply by using an efficient tree layout so that nearby nodes are located in adjacent memory locations. If $B = 1024$, searching a BFS tree for a key at a leaf requires 10× (or $\log_2 B$) more I/Os than searching a vEB tree with the same size N, where $N \gg B$.

On commodity machines with multi-level memory, the vEB layout is even more efficient. So far the vEB layout is shown to have $\log_2 B$ less I/Os for two-level memory. In a typical machine having three levels of inclusive caches (with cache line size of 64B), a RAM (with page size of 4KB) and a disk, a vEB tree search can intuitively give 640× less I/Os than a BFS tree search, assuming the node size is 4 bytes (cf. Figure 2c). However, the drawback of the vEB layout is in its recursive structure. For example if the tree is full, a new bigger tree needs to be built, recursively in one contiguous block of memory, which also means that the old tree needs to be invalidated and its members copied to the new tree. This drawback prevents an effective way to implement concurrency.

The concurrency-aware vEB layout. Our proposed concurrency-aware vEB layout has been proved to have the same data transfer efficiency between two

[1] For simplicity, we assume that the memory controller transfers a block of 3 nodes starting at the address of the requested node in memory.

```
Struct Map:
    member fields:
        $left \in \mathbb{N}$, $left$ child pointer address interval
        $right \in \mathbb{N}$, $right$ child pointer address intvl.

Map $map[UB]$

function RIGHT(p, base)
    $nodesize \leftarrow$ SIZEOF(node)
    $idx \leftarrow (p - base)/nodesize$
    if ($map[idx].right$ != 0) then
        return $base + map[idx].right$
    else
        return 0

function LEFT(p, base)
    $nodesize \leftarrow$ SIZEOF(node)
    $idx \leftarrow (p - base)/nodesize$
    if ($map[idx].left$ != 0) then
        return $base + map[idx].left$
    else
        return 0
```

Fig. 3: Map structure and the *mapping* functions.

memory levels as the *conventional* sequential vEB layout [36]. Because of the limited space, we spared the full details of our layout design in this paper, but in brief, a concurrency-aware vEB layout tree (U) is a tree consisting of $|U|$ GNodes $T_i, i = 1, \ldots, |U|$. Nodes of tree T_i are called *internal* nodes in order to distinguish them from GNodes. Each GNode contains a pre-allocated vEB-layout binary search tree (BST) structure that can hold a maximum of *UB internal* nodes. Each GNode's internal leaf nodes may link to another GNode's internal root node, which eventually form a *k-ary* tree of GNodes at the higher level. Note that this *k-ary* tree does not required to have a cache-oblivious layout [36].

2.1 GreenBST

GreenBST and DeltaTree is designed by devising three major strategies, namely it uses a common GNode map instead of pointers or arithmetic-based implicit BST (i.e., a node's successor memory address is calculated *on the fly*) for node traversals, crafting an efficient inter-node connection, and using balanced layouts. In addition to the shared common traits with DeltaTree, GreenBST also employs two new major strategies: (i) GreenBST uses incremental GNode rebalance and (ii) GreenBST uses heterogeneous GNode layouts.

Data structures. GreenBST is a collection of GNodes where each GNode consists of an *UB* internal **nodes** that hold the tree keys and a $^1/_2\,UB$ **link** array that links the GNode internal leaf nodes to another GNode's root node. Chain of GNodes formed a B+tree (to avoid confusion, from this point onward, we refer the "fat" nodes of GreenBST as GNode and the GNode's internal tree nodes as *internal nodes* or *nodes*). Each GNode also contains a lock (**locked**); a **rev** counter that is used for optimistic concurrency [26]; **nextRight** variable, which is a pointer that points to the GNode's right sibling; and **highKey** variable, which contains the lowest key member of the right sibling GNode. These last four variables are used for GreenBST concurrency control.

Cache-resident map instead of pointers or arithmetic implicit array. GreenBST does not use pointers to link between its internal nodes, instead it uses a single map-based implicit BST array. This approach is unique to the concurrency-aware vEB layout as it benefits from the usage of the fixed-size GNodes. The usage of pointers and arithmetic-based implicit array in cache-oblivious (CO) trees has been previously studied [8] and both are found to have weaknesses.

```
1:  function SEARCH($key$, $GNode$, $maxDepth$)
2:      while GNode is not leaf do
3:          $rev \leftarrow GNode.rev$ ▷ Get revision
4:          $bits \leftarrow 0$
5:          $depth \leftarrow 0$
6:          $p \leftarrow GNode.nodes[0]$
7:          $base \leftarrow p$
8:          $link \leftarrow GNode.link$
          ▷ continue until leaf node:
9:          while ($p$ & $p.key != $ EMPTY ) do
              ▷ increment depth:
10:             $depth \leftarrow depth + 1$
              ▷ shift one bit to the left in each level
11:             $bits \leftarrow bits << 1$
12:             if ($key < p.key$) then
13:                 $p \leftarrow$ LEFT($p$, $base$)
14:             else
15:                 $p \leftarrow$ RIGHT($p$, $base$)
                  ▷ right child color is 1:
16:                 $bits \leftarrow bits + 1$
          ▷ pad the $bits$:
17:         $bits \leftarrow bits << (maxDepth - depth) - 1$
18:         if ($GNode.rev$ != $rev$ or not even) then
19:             Goto 3          ▷ Re-try GNode search
          ▷ follow $nextRight$ if $key \leq highKey$:
20:         if ($GNode.highKey \leq key$) then
21:             $GNode \leftarrow GNode.nextRight$
22:         else
23:             $GNode \leftarrow link[bits]$   ▷ child GNode
24:     return $GNode$
```

Fig. 4: Search within pointer-less GNode. This function will return the *leaf* GNode containing the searched key. From there, an implicit array search using LEFT and RIGHT functions is adequate to pinpoint the key location. The search operations are utilizing both the **nextRight** pointers and **highKey** variables to handle concurrent search even during GNode split.

Pointer based CO tree search operation is slow, mainly because overheads in every data transfer between memory (although CO tree can minimize data transfers, the inclusion of pointers can lower the amount of meaningful data (e.g., keys) in each block transfer). The implicit array that uses arithmetic calculation for every node traversal may increase the cost of computation, especially if the tree is big.

The cache-resident-maps technique emulates BST's (left and right) child traversals inside a GNode using a combination of a cache-resident GNode *map* structure and LEFT and RIGHT functions (cf. Figure 3). The LEFT and RIGHT functions, given an arbitrary node v and its GNode's root memory addresses, return the addresses of the left and right child nodes of v, or 0 if v has no children (i.e., v is an internal leaf node of a GNode). The LEFT and RIGHT operations throughout GreenBST share a common cache-resident *map* instance (cf. Figure 3, line 5). All GNodes use the same fixed-size vEB layout, so only one *map* instance with size UB is needed for all traversing operations. This makes GreenBST's memory footprint small and keeps the frequently used *map* instance in cache.

Note that the mapping approach does not induce memory fragmentation. This is because mapping approach applies only for each GNode, and map is only used to point to internal nodes within a GNode. GNode layout uses a contiguous memory block of fixed size UB and *update* operations can only change the values of GNode internal nodes (e.g., from EMPTY to a key value in the case of insertion), but cannot change GNode's memory layout.

Inter-GNode connection. To enable traversing from a GNode to its child GNodes, we develop a new inter-GNode connection mechanism. We logically assign binary values to GNode's internal edges so that each path from GNode root to an internal leaf node is represented by a unique bit-sequence. The bit-sequence is then used as an index in a **link** array containing pointers to child GNodes. As GNode's internal node has only left and right edges, we assign 0 and 1 to the left and right edges, respectively. The maximum size of the bit representation is GNode's height or $\log(UB)$ bits. We allocate a link pointer array whose size is half UB length. The algorithm in Figure 4 explains how the inter-GNode connection works in a pointer-less search function.

Balanced and concurrent tree. GreenBST adopts the concurrent algorithms of B-link tree that provides lock-free search operations and adopts the B+tree structure for its high-level structure [28]. However, unlike B-link tree, GreenBST is an in-memory tree and uses optimistic concurrency to handle lock-free concurrent search operations even in the occurrences of the unique "in-place" GNodes maintenance operations.

Similar to B-link tree, GreenBST *insert* operations built the tree from the bottom up, but unlike B-link tree, GreenBST insert operation can trigger *rebalance* operation, a unique GreenBST feature to maintain GNode's small height.

Function $\text{REBALANCE}(T_i)$ is responsible for rebalancing a GNode T_i after an insertion. If a new node v is inserted at the *last level* node of a GNode, that GNode is rebalanced to a complete BST. Rebalance sets all GNode leaves node height to $\lfloor \log N \rfloor + 1$, where N is the count of the GNode's internal nodes and $N \leq UB$. Note that this is the default rebalance strategy used by DeltaTree, the incremental rebalance used by GreenBST is explained further in this section.

The *delete* operation in GreenBST simply marks the requested key (v) as deleted. This function fails if v does not exist in the tree or v is already marked. GreenBST does not employ merge operation between GNodes as node reclamation is done by the rebalance and split operations. The offline memory reclamation techniques used in the B-link tree [28] can be deployed to merge nearly empty GNodes in the case where delete operations are the majority. Our new search trees aim at workloads dominated by search operations.

GreenBST concurrency control uses locks and **nextRight** and **highKey** variables to coordinate between search and update operations [28] in addition to **rev** variable that is used for the search's optimistic concurrency. When a GNode needs to be maintained by either rebalance or split operations, the GNode's **rev** counter is incremented by one before the operation starts. The GNode counter is incremented by one again after the maintenance operation finishes. Note that all maintenance procedures happen when the lock is still held by the insert operation and therefore, only one operation may update **rev** counter and maintain a GNode at a time. The usage of **rev** counter is to prevent search from returning wrong key because of the "in-place" GNode maintenance operation.

The *search* operation in GreenBST uses a combination of function SEARCH (cf. Figure 4) and an implicit tree traversal using map. Function SEARCH traverses the tree from the internal root node of the root GNode down to a leaf GNode, at which the search is handed over to the implicit tree traversal to find the searched key within the leaf GNode. GreenBST *search* operation does not wait or use lock, even in the occurrence of the concurrent updates.

GreenBST *search* uses optimistic concurrency [26] to ensure the operation always returns the correct answer even if it arrives at a GNode that is undergoing the in-place maintenance operation (i.e., *rebalance* and *split*). First, before starting to traverse a GNode, a search operation records the GNode **rev** counter. Before following a link to a child GNode or returning a key, the search operation re-checks again the counter. 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 search as this indicates that GNodes are being or have been maintained.

Incremental Rebalance. As explained earlier, the rebalance in DeltaTree always involves *UB* keys, which eventually makes insertions require amortized $\mathcal{O}(UB)$ time. GreenBST borrows the incremental rebalance idea similar to the conventional vEB layout [8] that has the amortized $\mathcal{O}((\log^2 UB)/(1 - \mathit{\Gamma}_1))$ time if used in GreenBST. However, unlike the conventional vEB layout that might have to rebalance the whole tree, we only apply the incremental rebalance to GNodes. To briefly explain the idea, we denote *density(w)* as the ratio of number of keys inside a subtree rooted at w divided by the number of maximum keys that a subtree rooted at w can hold. For example, a subtree with root w that is located three levels away from an internal leaf of a GNode can hold at most $2^3 - 1$ keys. If the subtree only contains 3 keys, then $\mathit{density(w)} =^3 /_7 = 0.42$. We also denote a *density threshold* $0 < \mathit{\Gamma}_1 < \mathit{\Gamma}_2 < ... < \mathit{\Gamma}_H = 1$, where H is the GNode's height. The main idea is after a new key is inserted at an internal leaf position v, we find the nearest ancestor w of v where $\mathit{density(w)} \leq \mathit{\Gamma}_{depth(w)}$ and *depth(w)* is the level where w resides, counted from the root of the GNode. If that w is found, we rebalance the subtree rooted at w.

Heterogeneous GNodes. We aim to reduce the overhead of rebalancing and lower the GreenBST height with the usage of different layout for the leaf GNodes (or *heterogeneous*). All DeltaTree's GNodes use the leaf-oriented BST layout, or DeltaTree uses *homogeneous* GNodes. Unlike DeltaTree, leaf GNodes in GreenBST use the internal tree layout instead of the external (or leaf-oriented) tree layout. In the internal tree layout, keys are located in all nodes of a tree, while in the external tree layout, keys are only located in the leaf nodes. The reasoning behind this choice is although leaf-oriented GNodes layout is required for inter-GNode connection (i.e., between parent- and child- GNodes), leaf GNodes do not have any children and therefore, need not to adopt same structure as the other GNodes.

3 Experiments

We run several different benchmarks to evaluate GreenBST throughput and energy efficiency. We combine the benchmark results with the last level cache (LLC) and memory profiles of the trees to draw a conclusion of whether GreenBST improved fine-grained data locality layout (i.e., heterogeneous layout) and concurrency (i.e., lower overall cost of runtime maintenance) over DeltaTree are able to make GreenBST the most energy-efficient tree across different platforms, even when processing the update-intensive workloads. Note that we are not collecting the computation profiles (e.g., Mflops/second) because all the tree operations are data-intensive instead of compute-intensive.

We conduct an experiment on GreenBST and several prominent concurrent search trees (cf. Table 1) using parallel micro-benchmark that is based on Synchrobench [17] (cf. Figure 5). The trees' LLC and memory profiles during the micro-benchmarks are collected and presented in Figure 5d and 5e, respectively.

To investigate GreenBST behavior in real-world applications, we implement GreenBST and CBTree as the backend structures in the STAMP database benchmark `Vacation` [29], alongside the `Vacation`'s original backend structure red-black tree (rbtree) (cf. Figure 6).

All the experimental benchmarks are conducted on an Intel high performance computing (**HPC**) platform with 24 core 2× Intel Xeon E5-2650Lv3 CPU and 64GB of RAM, an **ARM** embedded platform with an 8 core Samsung Exynos 5410 CPU and 2GB of RAM (Odroid XU+E), and an accelerator platform based on the Intel Xeon Phi 31S1P with 57 cores and 6GB of RAM (**MIC** platform). For the parallel micro-benchmark, the trees are pre-initialized with several initial keys before running 5 million operations of 100% (search-intensive) and 50% searches (update-intensive), respectively. The initial keys given to both the ARM and MIC platforms are 2^{22} keys and to the HPC platform are 2^{23} keys. All experiments are repeated at least 5 times to guarantee consistent results.

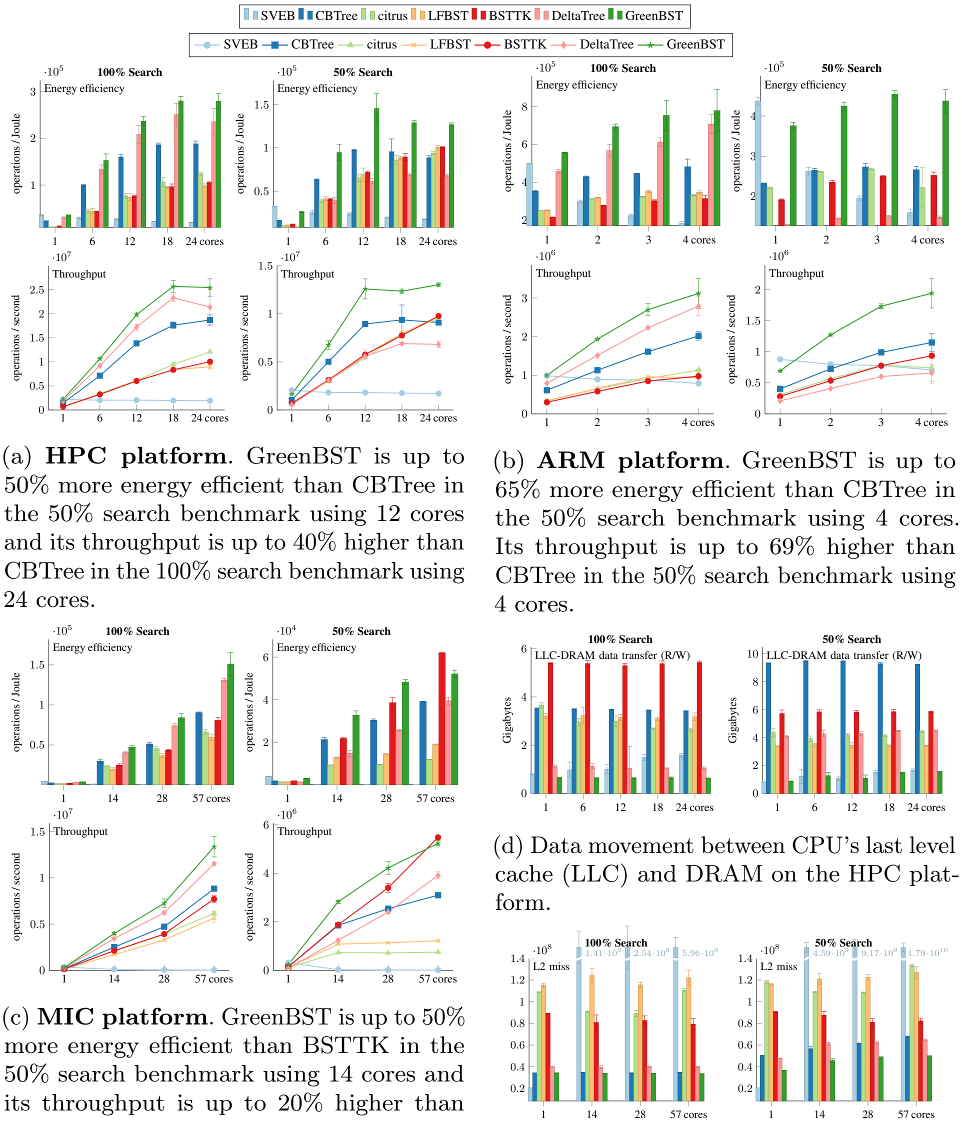
Energy efficiency metrics (in operations/Joule) are the energy consumption divided by the number of operations and *throughput metrics* (in operations/second) are the number of operations divided by the maximum time for the threads to finish the whole operations. Energy metrics are collected from the on-board power measurement on the ARM platform, Intel RAPL interface on the HPC platform, and micras sysfs interface (i.e., `/sys/class/micras/power`) on the MIC platform.

Experimental results. Based on the results in Figure 5 and 6, GreenBST's energy efficiency and throughput are the highest compared to DeltaTree and the other trees. Because of its *incremental rebalance*, GreenBST outperformed DeltaTree (and the other trees) in the update-intensive workloads. With its *heterogeneous layout*, GreenBST is able to outperform DeltaTree in the search-intensive workloads. GreenBST energy efficiency and throughput are up to 195% higher than DeltaTree for the update intensive benchmark and up to 20% higher for the search intensive benchmark (cf. Figure 5b). Compared to the other trees, GreenBST energy efficiency and throughput are up to 65% and 69% higher, respectively. Note that CBTree (B-link tree) is a highly-concurrent B-tree variant that it's still used as a backend in popular database systems such as PostgreSQL.

The reason behind GreenBST good results is GreenBST's data transfer (cf. Figure 5e) and LLC misses (cf. Figure 5d) are among the lowest of all the trees. These facts prove that combination of locality-aware layout and the optimizations that GreenBST has over DeltaTree are beneficial to both fine-grained locality and concurrency, of which are the key ingredients of an energy-efficient concurrent search tree.

4 Discussions

Some of the benchmark results showed that besides data movements, efficient concurrency control is also necessary in order to produce energy-efficient data structures. For example, the conventional vEB tree (SVEB) always transferred the smallest amount of data between memory to CPU, but unfortunately, its



(a) **HPC platform**. GreenBST is up to 50% more energy efficient than CBTree in the 50% search benchmark using 12 cores and its throughput is up to 40% higher than CBTree in the 100% search benchmark using 24 cores.

(b) **ARM platform**. GreenBST is up to 65% more energy efficient than CBTree in the 50% search benchmark using 4 cores. Its throughput is up to 69% higher than CBTree in the 50% search benchmark using 4 cores.

(c) **MIC platform**. GreenBST is up to 50% more energy efficient than BSTTK in the 50% search benchmark using 14 cores and its throughput is up to 20% higher than BSTTK in the 100% search benchmark using 14 cores.

(d) Data movement between CPU's last level cache (LLC) and DRAM on the HPC platform.

(e) L2 cache misses on the MIC platform.

Tree name	SVEB	CBTree	citrus	LFBST	BSTTK	DeltaTree	**GreenBST**
Memory used (in **GB**)	0.1	0.4	0.8	0.7	1.0	0.6	**0.4**

(f) The tree memory footprint after 2^{23} integer keys insertion on the HPC platform.

Fig. 5: (a,b,c) Energy efficiency and throughput comparison of the trees. On the HPC platform, DeltaTree and GreenBST energy efficiency and throughput decreases in the 50% search benchmark using 18 and 24 cores (i.e., with 2 chips) because of the coherence overheads between two CPUs (cf. Section 4). In the 50% search benchmark using 57 cores (MIC platform), BSTTK energy efficiency and throughput beats GreenBST by 20% because of the coherence overheads in the MIC platform (cf. Section 4). (d) LLC-DRAM data movements on the HPC platform, collected from the CPU counters using Intel PCM. (e) L2 cache miss counter on the MIC platform, collected using PAPI library. (f) The tree memory footprint.

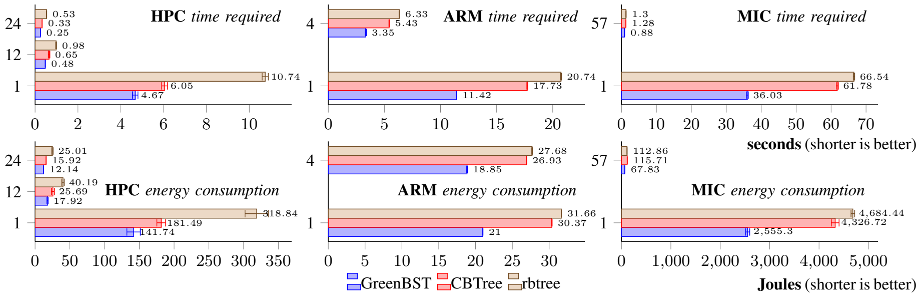


Fig. 6: GreenBST energy efficiency and throughput against CBTree and STAMP's built-in red-black tree (rbtree) for the vacation benchmark. At best, GreenBST consumes 41% less energy and requires 42% less time than CBTree (in the 57 clients benchmark on the MIC platform).

energy efficiency and throughput failed to scale when using 2 or more cores. SVEB is not designed for concurrent operations and an inefficient concurrency control (a global mutex) had to be implemented in order to include the tree in this study (note that we are unable to use a more fine-grained concurrency because SVEB uses recursive layout in a contiguous memory block). Therefore, even if SVEB has the smallest amount of data transfer during the micro-benchmarks, the concurrent cores have to spend a lot of time waiting and competing for a lock. This is inefficient as a CPU core still consumes power (e.g., static power) even when it is waiting (idle).

Finally, an important lesson that we have learned is that minimizing overheads in locality-aware data structures can reduce the structure's energy consumption. One of the main differences between DeltaTree and GreenBST is that DeltaTree uses the homogeneous (leaf-oriented) layout, while GreenBST does not. Leaf-oriented leaf GNodes increased DeltaTree's memory footprint by 50% compared to GreenBST (cf. Figure 5f) and has caused higher data transfer between LLC and DRAM (cf. Figure 5d). Bigger leaf size also increases maintenance cost for each leaf GNode, because there more data that need to be arranged in every rebalance or split operation, which leads to lower update concurrency. Therefore, DeltaTree energy efficiency and throughput are lower than GreenBST.

Inter-CPU and many-core coherence issue Our experimental analysis has revealed that multi-CPU and many-core cache coherence, if triggered, can degrade concurrent update throughput and energy efficiency of the locality-aware trees. Figure 5a shows the "dips" in GreenBST's 50% update energy efficiency and throughput on the HPC platform (i.e., in the *50% update/18 cores* and *50% update/24 cores* cases). Figure 5c also shows that BSTTK beats GreenBST in the *50% update/57 cores* case on the MIC platform.

Using the CPU performance counters, we have found that the GreenBST concurrent updates frequently triggered the inter-CPU coherency mechanism. In the HPC platform, coherency mechanism causes heavy bandwidth saturation in the CPU interconnect. In the MIC platform, it causes most of the L2 data cache misses to be serviced from other cores and saturates the platform's bidirectional ring interconnect. These facts highlight the challenge faced by the locality-aware concurrent search tree: because of its locality awareness (i.e., related data are kept nearby and often re-used), the tree concurrent update operations might trigger heavy interconnect traffic on the multi-CPU platforms. The coherency mechanisms increase the total number of data transfer and the platform's energy consumption.

5 Conclusions

The results presented in this paper not only show that GreenBST is an energy-efficient concurrent search tree, but also provide an important insight into how to develop energy efficient data structures in general. On single core systems, having locality-aware data structures that can lower data movement has been demonstrated to be good enough to increase energy-efficiency. However, on multi-CPU and many cores systems, data-structures' locality-awareness alone is not enough and good concurrency and multi-CPU cache strategy are needed. Otherwise, the energy overhead of "waiting/idling" CPUs or multi-CPU coherency mechanism can exceed the energy saving obtained by fewer data movements.

Acknowledgments

This work has received funding from the European Union Seventh Framework Programme (EXCESS project, grant n°611183) and from the Research Council of Norway (PREAPP project, grant n°231746/F20).

References

1. 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. pp. 1–15. DISC'12 (2012)
2. Aggarwal, A., Vitter, Jeffrey, S.: The input/output complexity of sorting and related problems. Commun. ACM 31(9), 1116–1127 (1988)
3. Andersson, A.: Faster deterministic sorting and searching in linear space. In: Proc. 37th Annual Symp. on Foundations of Computer Science. pp. 135–141. FOCS '96 (Oct 1996)
4. Arbel, M., Attiya, H.: Concurrent updates with rcu: Search tree as an example. In: Proc. 2014 ACM Symposium on Principles of Distributed Computing. pp. 196–205. PODC '14 (2014)
5. Bender, M., Demaine, E.D., Farach-Colton, M.: Cache-oblivious b-trees. SIAM Journal on Computing 35, 341 (2005)
6. 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. pp. 81–92. SPAA '07 (2007)
7. 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. pp. 228–237. SPAA '05 (2005)
8. Brodal, G.S., Fagerberg, R., Jacob, R.: Cache oblivious search trees via binary trees of small height. In: Proc. 13th ACM-SIAM Symp. Discrete algorithms. pp. 39–48. SODA '02 (2002)
9. Brodal, G.: Cache-oblivious algorithms and data structures. In: Hagerup, T., Katajainen, J. (eds.) Algorithm Theory - SWAT 2004, Lecture Notes in Computer Science, vol. 3111, pp. 3–13 (2004)
10. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms, Third Edition. The MIT Press (2009)
11. Crain, T., Gramoli, V., Raynal, M.: A speculation-friendly binary search tree. In: Proc. 17th ACM SIGPLAN Symp. Principles and Practice of Parallel Programming. pp. 161–170. PPoPP '12 (2012)
12. Dally, B.: Power and programmability: The challenges of exascale computing. In: DoE Arch-I presentation (2011)

13. David, T., Guerraoui, R., Trigonakis, V.: Asynchronized concurrency: The secret to scaling concurrent search data structures. In: Proc. 12th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems. pp. 631–644. ASPLOS '15 (2015)
14. 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. pp. 131–140. PODC '10 (2010)
15. van Emde Boas, P.: Preserving order in a forest in less than logarithmic time. In: Proc. 16th Annual Symp. Foundations of Computer Science. pp. 75–84. SFCS '75 (1975)
16. Frigo, M., Leiserson, C.E., Prokop, H., Ramachandran, S.: Cache-oblivious algorithms. In: Proc. 40th Annual Symp. Foundations of Computer Science. p. 285. FOCS '99 (1999)
17. 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. pp. 1–10. PPoPP 2015 (2015)
18. Ha, P.H., Tsigas, P., Anshus, O.J.: Wait-free programming for general purpose computations on graphics processors. In: Proc. 2008 IEEE International Symposium on Parallel and Distributed Processing. pp. 1–12. IPDPS'08 (2008)
19. Ha, P.H., Tsigas, P.: Reactive multi-word synchronization for multiprocessors. In: Proc. 12th Intl. Conf. on Parallel Architectures and Compilation Techniques. pp. 184–193. PACT '03 (2003)
20. Ha, P.H., Tsigas, P., Anshus, O.J.: Nb-feb: A universal scalable easy-to-use synchronization primitive for manycore architectures. In: Proc. 13th Intl. Conf. on Principles of Distributed Systems. pp. 189–203. OPODIS '09 (2009)
21. Ha, P.H., Tsigas, P., Anshus, O.J.: The synchronization power of coalesced memory accesses. IEEE Transactions on Parallel and Distributed Systems 21(7), 939–953 (2010)
22. Ha, P.H., Tsigas, P., Wattenhofer, M., Wattenhofer, R.: Efficient multi-word locking using randomization. In: Proc. 24th Annual ACM Symp. on Principles of Distributed Computing. pp. 249–257. PODC '05 (2005)
23. Herlihy, M., Moss, J.E.B.: Transactional memory: Architectural support for lock-free data structures. In: Proc. 20th Annual Intl. Symp. on Computer Architecture. pp. 289–300. ISCA '93 (1993)
24. Hipp, D.R.: Sqlite (2015), http://www.sqlite.org
25. 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. pp. 339–350. SIGMOD '10 (2010)
26. Kung, H.T., Robinson, J.T.: On optimistic methods for concurrency control. ACM Trans. Database Syst. 6(2), 213–226 (Jun 1981)
27. Larsson, A., Gidenstam, A., Ha, P.H., Papatriantafilou, M., Tsigas, P.: Multi-word atomic read/write registers on multiprocessor systems. In: Proc. 12th Annual European Symposium on Algorithms (ESA '04). pp. 736–748. LNCS 3221 (2004)
28. Lehman, P.L., Yao, s.B.: Efficient locking for concurrent operations on b-trees. ACM Trans. Database Syst. 6(4), 650–670 (Dec 1981)
29. Minh, C.C., Chung, J., Kozyrakis, C., Olukotun, K.: Stamp: Stanford transactional applications for multi-processing. In: Workload Characterization, 2008. IISWC 2008. IEEE International Symposium on. pp. 35–46 (Sept 2008)
30. Natarajan, A., Mittal, N.: Fast concurrent lock-free binary search trees. In: Proc. 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. pp. 317–328. PPoPP '14 (2014)
31. Prokop, H.: Cache-oblivious algorithms. Master's thesis, MIT (1999)
32. Rodeh, O.: B-trees, shadowing, and clones. Trans. Storage 3(4), 2:1–2:27 (Feb 2008)
33. 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)
34. Tran, V., Barry, B., Ha, P.H.: RTHpower: Accurate fine-grained power models for predicting race-to-halt effect on ultra-low power embedded systems. In: Proc. 17th IEEE International Symposium on Performance Analysis of Systems and Software. ISPASS '16 (2016), pages to appear
35. Tran, V., Barry, B., Ha, P.H.: Supporting energy-efficient co-design on ultra-low power embedded systems. In: Proc. 2016 Intl. Conf. on Embedded Computer Systems: Architectures, Modeling, and Simulation. SAMOS XVI (2016), pages to appear
36. Umar, I., Anshus, O.J., Ha, P.H.: Deltatree: A locality-aware concurrent search tree. In: Proc. 2015 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems. pp. 457–458. SIGMETRICS '15 (2015)
37. Umar, I., Anshus, O.J., Ha, P.H.: Effect of portable fine-grained locality on energy efficiency and performance in concurrent search trees. In: Proc. 21th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. pp. 36:1–36:2. PPoPP '16 (2016)