Cache-sensitive Memory Layout for Binary Trees*

Riku Saikkonen and Eljas Soisalon-Soininen

Helsinki University of Technology, Finland, {rjs,ess}@cs.hut.fi

Abstract We improve the performance of main-memory binary search trees (including AVL and red-black trees) by applying cache-sensitive and cache-oblivious memory layouts. We relocate tree nodes in memory according to a multi-level cache hierarchy, also considering the conflict misses produced by set-associative caches. Moreover, we present a method to improve one-level cache-sensitivity without increasing the time complexity of rebalancing. The empirical performance of our cache-sensitive binary trees is comparable to cache-sensitive B-trees. We also use the multi-level layout to improve the performance of cache-sensitive B-trees.

1 Introduction

Most of today's processor architectures use a hierarchical memory system: a number of caches are placed between the processor and the main memory. Caching has become an increasingly important factor in the practical performance of main-memory data structures. The relative importance of caching will likely increase in the future [1, 2]: processor speeds have increased faster than memory speeds, and many applications that previously needed to read data from disk can now fit all of the necessary data in main memory. In data-intensive main memory applications, reading from main memory is often a bottleneck similar to disk I/O for external-memory algorithms.

There are two types of cache-conscious algorithms. We will focus on the *cache-sensitive* (or cache-aware) model, where the parameters of the caches are assumed to be known to the implementation. In contrast, *cache-oblivious* algorithms attempt to optimize themselves to an unknown memory hierarchy.

The simplest cache-sensitive variant of the B-tree is an ordinary B^+-tree where the node size is chosen to match the size of a cache block (e.g., 64 or 128 bytes) [3]. A more advanced version called the Cache-Sensitive B^+-tree or CSB^+-tree [1] additionally removes pointers from internal nodes by storing the children of a node consecutively in memory. The CSB^+-tree has been further optimized using a variety of techniques, such as prefetching [4], storing only partial keys in nodes [5], and choosing the node size more carefully [2]. The above structures used a one-level cache model; B-trees in two-level cache models (one level of cache plus the TLB) are examined in [6, 7].

* This research was partially supported by the Academy of Finland.

A weight-balanced B-tree based on the cache-oblivious model has been proposed in [8]. Its simpler variants [9, 10] use an implicit binary tree (a complete binary tree stored in a large array without explicit pointers) whose structure and rebalancing operations are dictated by the cache-oblivious memory layout. In all three, update operations may rebuild parts of the tree, so most of the complexity bounds are amortized.

When using binary search trees, the node size cannot be chosen as freely as in B-trees. Instead, we will place the nodes in memory so that each cache block contains nodes that are close to each other in the tree. Binary search tree nodes are relatively small; for example, AVL and red-black tree nodes can fit in about 16 or 20 bytes using 4-byte keys and 4-byte pointers, so 3–8 nodes fit in one 64-byte or 128-byte cache block. (We assume that the nodes contain only small keys. Larger keys could be stored externally with the node storing a pointer to the key.)

Caching and explicit-pointer binary search trees have been previously considered in [11], which presents a cache-oblivious splay tree based on periodically rearranging all nodes in memory. In addition, [12] presents a one-level cache-sensitive periodic rearrangement algorithm for explicit-pointer binary trees. A similar one-level layout (extended to unbalanced trees) is analyzed in [13], which also discusses the multi-level cache-oblivious layout known as the van Emde Boas layout. The latter is analyzed in detail in [14].

We give an algorithm that preserves cache-sensitivity in binary trees in the dynamic case, i.e., during insertions and deletions. Our algorithm retains single-level cache-sensitivity using small worst-case constant-time operations executed when the tree changes. In addition, we give an explicit algorithm for multi-level cache-sensitive global rearrangement, including a variation that obtains a cache-oblivious layout. We also investigate a form of conflict miss caused by cache-sensitive memory layouts that interact poorly with set-associative caches.

Our approach does not change the internal structure of the nodes nor the rebalancing strategy of the binary search tree. The approach is easy to implement on top of an existing implementation of any tree that uses rotations for balancing, e.g., red-black trees and AVL trees. Our global rearrangement algorithm can also be applied to cache-sensitive B-trees, and our empirical results indicate that the multi-level memory layout improves the performance of both B^+-trees with cache-block-sized nodes and CSB^+-trees.

2 Cache model

We define a multi-level cache model as follows. We have a k-level cache hierarchy with block sizes $B_1, \ldots, B_k$ at each level. We also define $B_0 =$ node size in bytes, $B_{k+1} = \infty$. We assume that our algorithms know these cache parameters. (In practice, they can be easily inferred from the CPU model or from metadata

stored in the CPU.) To keep the model simple, we do not model any other features of the cache, such as the capacity.

Our algorithms shorten the *B_i-block search path length*, denoted P_i and defined as the length of a root-to-leaf path measured in the number of separate cache blocks of size B_i encountered on the path. Using this terminology, P_0 is the traditional search path length in nodes (assuming that the search does not end before the leaf level), P_1 is the length counted in separate B_1-sized cache blocks encountered on the path, and so on.

We assume that for $i > 1$, each block size B_i is an integer multiple of B_{i-1}. Additionally, if B_1 is not an integer multiple of the node size B_0, a node should not cross a B_1-block boundary (so that it is never necessary to fetch two cache blocks from memory in order to access a single node). In practice, this is achieved by not using the last B_1 **mod** B_0 bytes of each B_1-block. (In practice, B_i, $i > 0$, is almost always a power of 2.)

A typical modern computer employs two levels of caches: a relatively small and fast level 1 ("L1") cache, and a larger and slower level 2 ("L2") cache. In addition, the mapping of virtual addresses to physical addresses used by multitasking operating systems employs a third hardware cache: the Translation Lookaside Buffer or TLB cache.

Currently the cache block size is often the same in the L1 and L2 caches. They then use only one level of our hierarchy. For example, the cache model used in the experiments in Section 5 is $k = 2$, $B_0 = 16$ (16-byte nodes), $B_1 = 64$ (the block size of the L1 and L2 caches in an AMD Athlon XP processor), $B_2 = 4096$ (the page size of the TLB cache), $B_3 = \infty$. However, our algorithms can be applied to an arbitrary hierarchy of cache block sizes.

3 Global relocation

Figure 1 gives an algorithm that rearranges the nodes of a tree in memory into a multi-level cache-sensitive memory layout. The algorithm can be used for any kind of balanced tree with fixed-size nodes.

The produced layout can be considered to be a generalization of the one-level cache-sensitive layouts of [12, 13] and the two-level layouts of [6, 7] to an arbitrary hierarchy of block sizes. It is different from the multi-level "van Emde Boas" layout (see [13]) in that the recursive placement of smaller blocks in larger ones is more complex, because, in the cache-sensitive model, we cannot choose the block sizes according to the structure of the tree, as is done in the cache-oblivious van Emde Boas layout.

In the produced layout, the first lowest-level ($l = 1$) block is filled by a breadth-first traversal of the tree starting from the root r. When this "root block" is full, each of its children (i.e., the "grey" or border nodes in the breadth-first search) will become the root node of its own level 1 block, and so on. On levels $l > 1$, level $l-1$ blocks are allocated to level l blocks in the same manner.

```
RELOC-BLOCK(l, r):
 1: if l = 0 then
 2:    Copy node r to address A, and update the link in its parent.
 3:    A ← A + B_0
 4:    return children of r
 5: else
 6:    S ← A
 7:    E ← A + F(A, l) − B_{l−1}
 8:    Q ← empty queue
 9:    put(Q, r)
10:    while Q is not empty and A ≤ E do
11:       n ← get(Q)
12:       c ← RELOC-BLOCK(l − 1, n)
13:       put(Q, all nodes in c)
14:    end while
15:    if Q is not empty then
16:       A ← start of next level l block (= E + B_{l−1})
17:       if F(S, l) < B_l/2 then       {less than half of the block was free}
18:          Free the copies made above, i.e., all nodes at addresses S to A − 1.
19:          return r       {our caller will try to relocate r again later}
20:       end if
21:    end if
22:    return remaining nodes in Q
23: end if

RELOCATE(r):
1: A ← beginning of a new memory area, aligned at a level k block boundary
2: RELOC-BLOCK(k + 1, r)       {B_{k+1} = ∞, so this relocates everything}
```

Fig. 1 The global relocation algorithm. The address A of the next available position for a node is a global variable. $F(A, l) = B_l - A$ **mod** B_l is the number of bytes between A and the end of the level l block containing A. (To be able to update the link in a parent when a node is copied, the algorithm actually needs to store (node, parent) pairs in the queue Q, unless the tree structure contains parent links. This was left out of the pseudocode for clarity.)

The algorithm of Figure 1 produces this layout using a single traversal of the tree using auxiliary queues that store border nodes for each level of the breadth-first search. Lines 17–20 are an optional space optimization: at the leaf level, there may not be enough nodes to fill a block. Lines 17–20 ensure that each level l block will be at least half full by trying to allocate the next available subtree in the remaining space in a non-full block.

Theorem 1. *Assume that the global relocation algorithm of Figure 1 is executed on a complete binary tree of height h. Then the worst-case B_i-block path length will be $P_i = \lceil h/h_i \rceil$, where $h_i = h_{i-1} \cdot \lfloor \log_{d_{i-1}}(B_i/B_{i-1} + 1) \rfloor$, $h_0 = 1$. If B_1 is an integer multiple of B_0, then $d_i = B_i/B_0 + 1$; otherwise, $d_0 = 2$ and $d_i = (d_{i-1} - 1) \cdot \lfloor B_i/B_{i-1} \rfloor + 1$.*

Proof. Consider a cache block level $i \in \{1, \ldots, k\}$. Each level $i - 1$ block produced by the layout (except possibly for blocks that contain leaves of the tree) contains a connected part of the tree with $d_{i-1} - 1$ binary tree nodes. These

blocks can be thought of as "super-nodes" with fanout d_{i-1}. The algorithm of Figure 1 produces a level i block by allocating B_i/B_{i-1} of these super-nodes in breadth-first order (i.e., highest level $i-1$ block first). The shortest root-to-leaf path of the produced level i block has h_i binary tree nodes. □

The produced layout is optimal on the level of B_1-blocks: it is not possible to produce a larger h_1. It is not possible to be optimal on all levels [14], and we resolved this tradeoff by preferring the lowest level. Knowledge of the relative costs of cache misses at each level could in theory be used to produce a more optimal layout, but we did not want our cache-sensitive algorithms to depend on these kinds of additional parameters.

Theorem 2. *The algorithm of Figure 1 rearranges the nodes of a tree into a multi-level cache-sensitive memory layout in time $O(nk)$, where n is the number of nodes in the tree and k is the number of memory-block levels.*

Proof. Each node in the tree is normally copied to a new location only once. However, the memory-usage optimization in line 18 may "undo" (free) some of these copies. The undo only happens when filling a level l cache block that was more than half full, and the layout is then restarted from an empty level l cache block. Thus, an undo concerning the same nodes cannot happen again on the same level l. However, these nodes may already have taken part in an undo on a smaller level $l' < l$. In the worst case, a node may have taken part in an undo on all k memory-block levels. Each of the n nodes can then be copied at most k times.

Consider then the queues Q at various levels of recursion. Each node enters a queue at level $l = 1$ (line 13, using c from line 4), and travels up to a level $l' \leq k+1$, where it becomes the root of a level $l'-1$ subtree and descends to level 0 in the recursion. Thus, each node is stored in $O(k)$ queues. □

Cache-oblivious layout. Though cache-sensitive, the produced layout is similar to the "van Emde Boas" layout used as the basis of many cache-oblivious algorithms. In fact, our algorithm can produce the van Emde Boas layout: simply use the block sizes $B_i = (2^{2^i} - 1) \cdot B_0$ ($i = 1, \ldots, k$ where $k = 4$ or $k = 5$ is enough for trees that fit in main memory). The only difference between the layout thus produced and the van Emde Boas layout (as described in, e.g., [13]) is that the recursive subdivision is done top-down instead of bottom-up, and some leaf-level blocks may not be full. (These differences are unavoidable because the van Emde Boas layout is defined only for complete trees.)

Aliasing correction. While experimenting with the global relocation algorithm, we found that multi-level cache-sensitive layouts can suffer from a problem called aliasing, a kind of repeated conflict miss. Many hardware caches are d-way set associative ($d \in \{2, 4, 8\}$ are common), i.e., there are only d possible places in the cache for a block with a given address A. The problem is that, for instance, the ith cache block in each TLB page is often mapped to the same set of d places. Therefore, if the ith cache blocks of several TLB pages are accessed, the cache can only hold d of these blocks.

A straightforward multi-level cache-sensitive layout (including the one produced by the above algorithm) fills a TLB page (of size B_l for some l) with a subtree so that the root of the subtree is placed at the beginning of the TLB page (i.e., in the first B_{l-1}-sized cache block). Then, for example, when a particular root-to-leaf path is traversed in a search, only d root nodes of these TLB-sized subtrees can be kept in the (set associative) B_{l-1}-block cache. (The root of the TLB-sized subtree is not of course the only problematic node, but the problem is most pronounced at the root.)

The problem can be fixed by noting that we can freely reorder the cache blocks inside a TLB page. The B_l-sized TLB page consists of B_l/B_{l-1} cache blocks, and the subtree located in the TLB page can use these cache blocks in any order. We simply use a different ordering for separate TLB pages, so the root node of the subtree will not always be located in the first cache block.

We implement the reordering by a simple cache-sensitive translation of the addresses of each node allocated by the global relocation algorithm, as follows.[2] Every address A can be partitioned into components according to the cache block hierarchy: $A = A_k \dots A_2 A_1 A_0$, where each A_i, $i \in \{1, \dots, k-1\}$, has $\log_2 B_i/B_{i-1}$ bits of A, and A_0 and A_k have the rest. For each level $i = \{1, \dots, k\}$, we simply add the upper portion $A_k \dots A_{i+1}$ to A_i, modulo B_i/B_{i-1} (so that only the A_i part is changed).

For example, if B_l is the size of the TLB page, the root of the first allocated TLB page ($A_k \dots A_{l+1} = 0$) will be on the first cache block (the translated portion $A'_l = 0$), but the root of the second TLB page (which is a child of the first page) will be on the second cache block ($A_k \dots A_{l+1} = 1$, so $A'_l = 1$) of its page.

It would be enough to apply this correction to those memory-block levels with set associative caches on the previous level (i.e., only level l in the above example, since level $l-1$ has the set associative cache). However, we do it on all levels, because then our cache-sensitive algorithms only need knowledge of the block sizes and not any other parameters of the cache hierarchy. Applying the translation on every level increases the time complexity of the relocation algorithm to $O(nk^2)$, but this is not a problem in practice, since k is very small (e.g., $k = 2$ was discussed above).

4 Local relocation

When updates (insertions and deletions) are performed on a tree which has been relocated using the global algorithm of the previous section, each update may disrupt the cache-sensitive memory layout at the nodes that are modified in the update. In this section, we present modifications to the insert and delete algo-

[2] The translation is applied to every address used in lines 2 and 18 of the algorithm of Figure 1. The other addresses S, A and E in the algorithm do not need to be translated, because they are only used to detect block boundaries.

rithms that try to preserve a good memory layout without increasing the time complexity of insertion and deletion in a binary search tree that uses rotations for balancing. These algorithms can be used either together with the global relocation algorithm of the previous section (which could be run periodically) or completely independently.

Our approach preserves the following memory-layout property:

Invariant 1 *For all non-leaf nodes x, either the parent or one of the children of x is located on the same B_1-sized cache block as x.*

This property reduces the average B_1-block path length even in a worst-case memory layout. For simplicity, the proof only considers a complete binary tree of height h. (To see that Invariant 1 improves the memory layout of, e.g., a red-black tree, note that the top part of a red-black tree of height h is a complete tree of height at least $h/2$.)

Theorem 3. *Assume that Invariant 1 holds in a complete binary tree of height h. Then the average B_1-block path length $\overline{P_1} \leq 2h/3 + 1/3$.*

Proof. In the worst-case memory layout, each B_1-sized cache block contains only nodes prescribed by Invariant 1, i.e., a single leaf or a parent and child.

By Invariant 1, the root r of the tree (with height h) is on the same cache block as one of its children. Considering all possible paths down from r leads to the following recurrence for the expected value of the B_1-block path length: $P(h) = 1/2 \cdot (1 + P(h-2)) + 1/2 \cdot (1 + P(h-1))$ (with $P(1) = 1$ and $P(0) = 0$). Solving gives $E[P_1|\text{worst-case memory layout}] = P(h) = 2h/3 + 2/9 - 2(-1)^h/(9 \cdot 2^h) \leq 2h/3 + 1/3$. In any memory layout, the average $\overline{P_1} \leq E[P_i|\text{worst-case memory layout}]$. □

We say that a node x is *broken* if Invariant 1 does not hold for x. To analyze how this can happen, denote $N(x)$ = the set of "neighbors" of node x, i.e., the parent of x and both of its children (if they exist). Furthermore, say that x *depends* on y if y is the only neighbor of x that keeps x non-broken (i.e., the only neighbor on the same cache block).

Our local relocation approach works as follows. We do the standard binary search tree structure modification operations (insertion, deletion, rotations) as usual, but after each such operation, we collect a list of nodes that can potentially be broken (Figure 2), and use the algorithm given below to re-establish Invariant 1 before executing the next operation.

The nodes that can break are exactly those whose parent or either child changes in the structure modification, since a node will break if it depended on a node that was moved away or deleted. As seen from Figure 2, 1 to 6 nodes can be broken by one structure modification. We explain the various cases in Figure 2 below.

In internal trees, actual insertion is performed by adding a new leaf node to an empty location in the tree. If the parent of the new node was previously a leaf, it may now be broken; thus, the parent is marked as potentially broken in Figure 2(c).

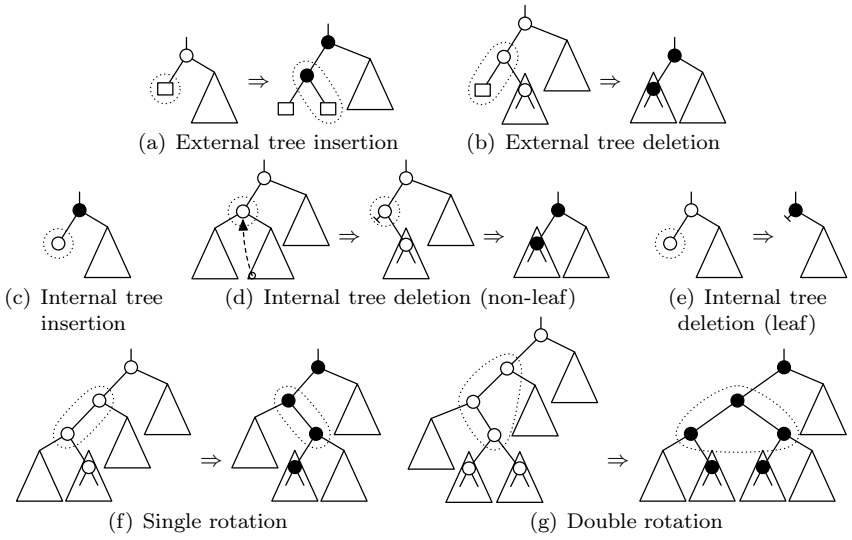


Fig. 2 Broken nodes in actual insertion, actual deletion and rotations. Potentially broken nodes are filled black; the dotted lines indicate the nodes that the operation works on.

In external (leaf-oriented) trees, actual insertion replaces a leaf node by a new internal node with two children: the old leaf and a new one (see Figure 2(a)). The new internal node is potentially broken (if it was not allocated on the same cache block as one of the other nodes), and its parent may become broken, if the parent depended on the old leaf node.

Actual deletion in external trees deletes a leaf and its parent and replaces the parent with its other child (Figure 2(b)). The parent of the deleted internal node and the other child can become broken, since they could have depended on the deleted internal node.

Actual deletion in internal trees is slightly more complicated, with two cases. In the simple case (Figure 2(e)), a leaf is deleted, and its parent becomes broken, if it depended on the deleted leaf. The more complicated case arises when a non-leaf node x needs to be deleted (Figure 2(d)). The standard way of doing the deletion is to locate the node y with the next-larger key from the right child of x, copy the key and possible associated data fields to x, and then delete y by replacing it with its right child (if any). In this process, the parent of y and the right child can become broken (if they depended on y). The node x cannot become broken, since it or its neighbors were not moved in memory. (The equivalent implementation that looks for the next-smaller key of x in its left child is completely symmetric with regard to broken nodes.)

When a single or double rotation is performed, the nodes that can break are those whose parent or either child changes in the rotation, since a node will break if it depended on a node that was moved away by the rotation.

FIX-BROKEN(B):

1: **while** $B \neq \emptyset$ **do**
2: Remove any non-broken nodes from B (and exit if B is emptied).
3: **if** a node in $N(B)$ has free space in its cache block **then**
4: Select such a node x and a broken neighbor $b \in B$. (Prefer the x with the most free space and a b with no broken neighbors.)
5: Move b to the cache block containing x.
6: **else if** a node $b \in B$ has enough free space in its cache block **then**
7: Select the neighbor $x \in N(b)$ with the smallest $|D(x)|$.
8: Move x and all nodes in $D(x)$ to the cache block containing b.
9: **else**
10: Select a node $x \in N(B)$ and its broken neighbor $b \in B$. (Prefer a broken x, and after that an x with small $|D(x)|$. If there are multiple choices for b, prefer a b with $N(b) \setminus x$ non-broken.)
11: Move b, x and all nodes in $D(x)$ to a newly-allocated cache block.
12: **end if**
13: **end while**

Fig. 3 The local relocation algorithm. B is a set of potentially broken nodes which the algorithm will make non-broken; $N(B) = \bigcup_{b \in B} N(b)$. An implementation detail is that the algorithm needs access to the parent, grandparent and great grandparent of each node in B, since the grandparent may have to be moved in lines 8 and 11.

We can optimize the memory layout somewhat further with a simple heuristic (not required for Invariant 1): In insertion, a new node should be allocated in the cache block of its parent, if it happens to have enough free space.

We need an additional definition for the algorithm of Figure 3: $D(x)$ is the set of neighbors of node x that depend on node x (i.e., will be broken if x is moved to another cache block). Thus, $D(x) \subset N(x)$ and $0 \leq |D(x)| \leq |N(x)| \leq 3$. A crucial property is that nothing depends on a broken node (because no neighbor is on the same cache block), and thus broken nodes can be moved freely.

The algorithm of Figure 3 repeats three steps until the set of broken nodes B is empty. First, all neighbors of the broken nodes are examined to find a neighbor x with free space in its cache block. If such a neighbor is found, a broken node $b \in N(x)$ is fixed by moving it to this cache block. If no such neighbor was found, then the cache blocks of the nodes in B are examined; if one of them has enough space for a neighbor x and its dependants $D(x)$, they are moved to this cache block. Otherwise, if nothing was moved in the previous steps, then a broken node b is forcibly fixed by moving it and some neighboring nodes to a newly allocated cache block. At least one neighbor x of b needs to be moved along with b to make b non-broken; but if x was not broken, some of its other neighbors may depend on x staying where it is – these are exactly the nodes in $D(x)$, and we move all of them to the new cache block. (It is safe to move the nodes in $D(x)$ together with x, because their other neighbors are not on the same cache block.)

Theorem 4. *Assume that Invariant 1 holds in all nodes in a tree, except for a set B of broken nodes. Then giving B to the algorithm of Figure 3 will establish Invariant 1 everywhere.*

Theorem 5. *The algorithm of Figure 3 moves at most $4|B| = O(|B|)$ nodes in memory. The total time complexity of the algorithm is $O(|B|^2)$.*

Proof. Each iteration of the loop in the algorithm of Figure 3 fixes at least one broken node. Line 5 does this by moving one node; line 11 moves at most 4 nodes (b, x, and the two other neighbors of x), and line 8 moves at most 3 nodes (x and two neighbors). Thus, at most $4|B|$ nodes are moved in the at most $|B|$ iterations that the algorithm executes.

Each iteration looks at $O(|B|)$ nodes; thus, the total time complexity is $O(|B|^2)$. Additionally, looking for free nodes in a B_1 cache block can require more time. A naïve implementation looks at every node in the B_1-block to locate the free nodes, thus increasing the time complexity to $O(|B|^2 \cdot B_1/B_0)$. This may actually be preferable with the small B_1 of current processors. (The implementation we describe in Section 5 did this, with $B_1/B_0 = 4$.)

With larger B_1/B_0, the bound of the theorem is reached simply by keeping track of the number of free nodes in an integer stored somewhere in the B_1-sized block. To find a free node in constant time, a doubly-linked list of free nodes can be stored in the (otherwise unused) free nodes themselves, and a pointer to the head of this list is stored in a fixed location of the B_1-block. □

Remember that $|B| \leq 6$ always when we execute the algorithm.

A space-time tradeoff is involved in the algorithm of Figure 3: we sometimes allocate a new cache block to get two nodes on the same cache block (thus improving cache locality), even though two existing cache blocks have space for the nodes. Since our relocation algorithm always prefers an unused location in a previously allocated cache block, it is to be hoped that the cache blocks do not become very empty on average. (Moving unrelated nodes on the existing cache blocks "out of the way" is not practical: to move a node x in memory, we need access to the parent of x to update the link that points to the node, and our small-node trees do not store parent links.)

We get a lower limit for the cache block fill ratio from the property that our algorithm preserves: each non-leaf node has at least the parent or one child accompanying it on the same cache block. (Empty cache blocks should of course be reused by new allocations.)

5 Experiments

We implemented the algorithms of Sections 3 and 4 on internal AVL and red-black trees, and compared them to the cache-sensitive B^+-tree (the "full CSB^+-tree" of [1]) and to a standard B^+-tree with cache block-sized nodes (called a

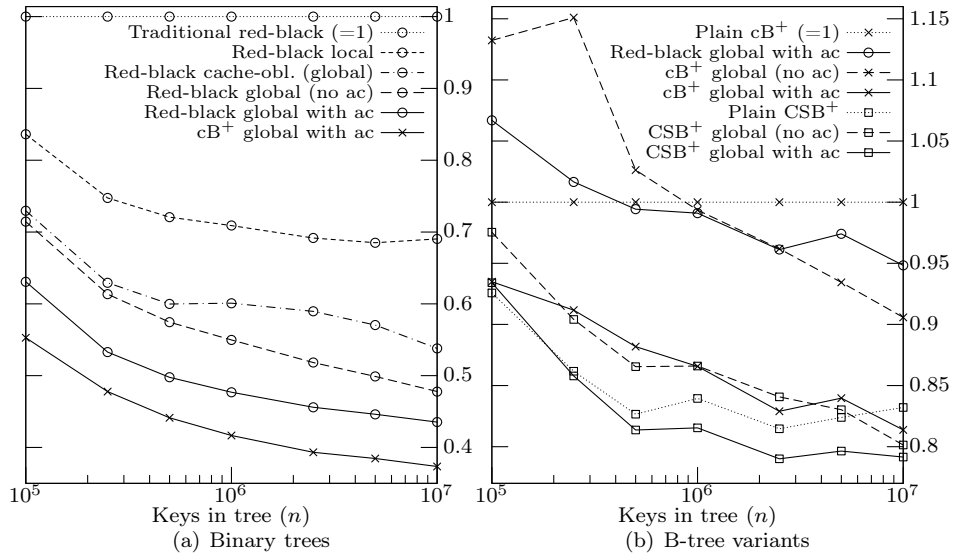


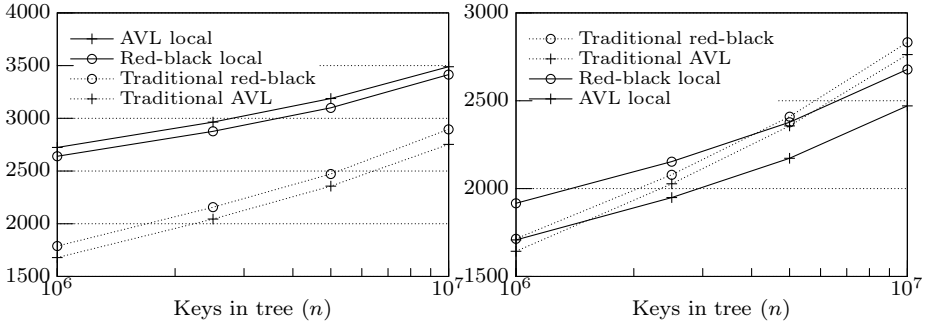
Fig. 4 Effect of global and local relocation and aliasing correction (="ac"). The figures give the search time relative to (a) the traditional red-black tree, (b) the cB$^+$-tree. The trees marked "global" have been relocated using the global algorithm. "Red-black local" uses local relocation; the others use neither global nor local relocation. AVL trees (not shown) performed almost identically to red-black trees.

"cB$^+$-tree" below for brevity).[3] As noted in Section 2, we used the following cache parameters: $k = 2$, $B_0 = 16$, $B_1 = 64$, $B_2 = 4096$, $B_3 = \infty$. The tree implementations did not have parent links: rebalancing was done using an auxiliary stack.[4]

Figure 4 examines the time taken to search for 10^5 uniformly distributed random keys in a tree initialized by n insertions of random keys. (Before the 10^5 searches whose time was measured, the cache was "warmed up" with 10^4 random searches.) The search performance of red-black and AVL trees relocated using the global algorithm was close to the cB$^+$-tree. The local algorithm was not quite as good, but still a large (about 30%) improvement over a traditional non-cache-optimized binary tree. The cache-oblivious layout produced by the

[3] We executed our experiments on an AMD Athlon XP processor running at 2167 MHz, with 64 Kb L1 data cache (2-way associative) and 512 Kb L2 cache (8-way associative). Our implementation was written in C, compiled using the GNU C compiler version 4.1.1, and ran under the Linux kernel version 2.6.18. Each experiment was repeated 15 times; we report averages.

[4] The binary tree node size $B_0 = 16$ bytes was reached by using 4-byte integer keys, 4-byte data fields and 4-byte left and right children. The balance and color information for the AVL and red-black tree was encoded in the otherwise unused low-order bits of the child pointers. The nodes of the B-trees were structured as simple sorted arrays of keys and pointers. The branching factor of a non-leaf node was 7 in the cB$^+$-tree and 14 in the CSB$^+$-tree.



(a) Nanoseconds per insertion (b) Nanoseconds per deletion

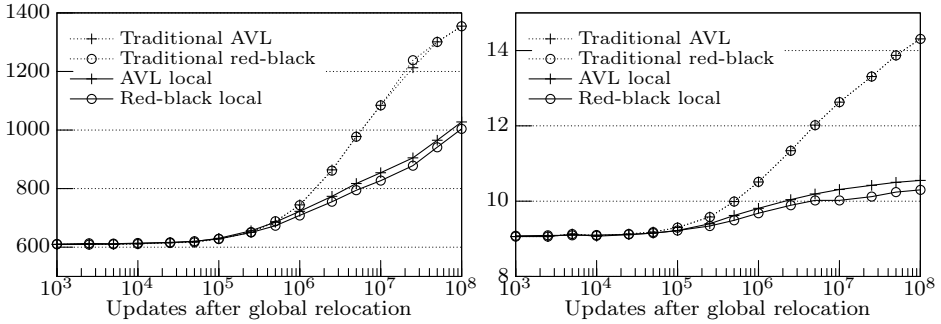
Fig. 5 Effect of the local relocation algorithm on the time taken by (a) insertions and (b) deletions.

global algorithm was somewhat worse than a cache-sensitive layout, but about 40–45% better than a non-cache-optimized tree. Aliasing correction had about 10–15% impact on binary trees and cB$^+$-trees, and about 5% on CSB$^+$-trees (which don't always access the first B_1-sized node of a TLB page). Especially in the B-trees, global relocation was not very useful without aliasing correction. In summary, the multi-level cache-sensitive layout improved binary search trees by 50–55%, cB$^+$-trees by 10–20% and CSB$^+$-trees by 3–5% in these experiments.

Figure 5 examines the running time of updates when using the local algorithm. Here the tree was initialized with n random insertions, and then 10^4+10^5 uniformly distributed random insertions or deletions were performed. The times given are averaged from the 10^5 updates (the 10^4 were used to "warm up" the cache). The local algorithm increased the insertion time by about 20–70% (more with smaller n). The deletion time was affected less: random deletions in binary search trees produce less rotations than random insertions, and the better memory layout produced by the local algorithm decreases the time needed to search for the key to be inserted or deleted.

In addition, we combined the global and local algorithms and investigated how quickly updates degrade the cache-sensitive memory layout created by the global algorithm. In Figure 6, we initialized the tree using $n = 10^6$ random insertions, executed the global algorithm once, and performed a number of random updates (half insertions and half deletions). Finally, we measured the average search time from 10^5 random searches (after a warmup period of 10^4 random searches), and the average B_1-block path length. The results indicate that the cache-sensitivity of the tree decreases significantly only after about n updates have been performed. The local algorithm keeps a clearly better memory layout, though it does not quite match the efficiency of the global algorithm.

Our experiments, as well as those in [6, 9], support the intuition that multi-level cache-sensitive structures are more efficient than cache-oblivious ones. It has been shown in [14] that a cache-oblivious layout is never more than 44%



(a) Search time (nanoseconds)

(b) Average P_1 (B_1-sized blocks)

Fig. 6 Degradation of locality when random insertions and deletions are performed after global relocation of a tree with $n = 10^6$ initial keys: (a) average search time from 10^5 searches, (b) B_1-block path length. For 0 to 10^4 updates after relocation, there was practically no change in the values; here, the x axis begins from 10^3 for clarity.

worse in the number of block transfers than an optimal cache-sensitive layout, and that the two converge when the number of levels of caches increases. However, the cache-sensitive model is still important, because the number of levels of caches with different block sizes is relatively small in current computers (e.g., only two in the one we used for our experiments).

6 Conclusions

We have examined binary search trees in a k-level cache memory hierarchy with block sizes $B_1, \ldots, B_k$. We presented an algorithm that relocates tree nodes into a multi-level cache-sensitive memory layout in time $O(nk)$, where n is the number of nodes in the tree. Moreover, our one-level local algorithm preserves an improved memory layout for binary search trees by executing a constant-time operation after each structure modification (i.e., actual insertion, actual deletion or individual rotation).

Although cache-sensitive binary trees did not quite match the speed of the B$^+$-tree variants in our experiments, in practice there may be other reasons than average-case efficiency to use binary search trees. For instance, the worst-case (as opposed to amortized or average) time complexity of updates in red-black trees is smaller than in B-trees: $O(\log_2 n)$ vs. $O(d \log_d n)$ time for a full sequence of page splits or merges in a d-way B-tree, $d \geq 5$. Red-black tree rotations are constant-time operations, unlike B-tree node splits or merges (which take $O(B_1)$ time in B-trees with B_1-sized nodes, or $O(B_1^2)$ in the full CSB$^+$-tree). This may improve concurrency: nodes are locked for a shorter duration. In addition, it has been argued in [15] that, in main-memory databases, binary trees are optimal

for a form of shadow paging that allows efficient crash recovery and transaction rollback, as well as the group commit operation [16].

The simple invariant of our local algorithm could be extended, for instance, to handle multi-level caches in some way. However, we wanted to keep the property that individual structure modifications use only $O(1)$ time (as opposed to $O(B_1)$ or $O(B_1^2)$ for the cache-sensitive B-trees). Then we cannot, e.g., move a cache-block-sized area of nodes to establish the invariant after a change in the tree structure. A multi-level approach does not seem feasible in such a model.

Other multi-level cache-sensitive search tree algorithms are presumably also affected by the aliasing phenomenon, and it would be interesting to see the effect of a similar aliasing correction on, for example, the two-level cache-sensitive B-trees of [7].

References

1. Rao, J., Ross, K.A.: Making B+-trees cache conscious in main memory. In: 2000 ACM SIGMOD International Conference on Management of Data, ACM Press (2000) 475–486
2. Hankins, R.A., Patel, J.M.: Effect of node size on the performance of cache-conscious B+-trees. In: 2003 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems, ACM Press (2003) 283–294
3. Rao, J., Ross, K.A.: Cache conscious indexing for decision-support in main memory. In: 25th International Conference on Very Large Data Bases (VLDB 1999), Morgan Kaufmann (1999) 78–89
4. Chen, S., Gibbons, P.B., Mowry, T.C.: Improving index performance through prefetching. In: 2001 ACM SIGMOD International Conference on Management of Data, ACM Press (2001) 235–246
5. Bohannon, P., McIlroy, P., Rastogi, R.: Main-memory index structures with fixed-size partial keys. In: 2001 ACM SIGMOD International Conference on Management of Data, ACM Press (2001) 163–174
6. Rahman, N., Cole, R., Raman, R.: Optimised predecessor data structures for internal memory. In: 5th Workshop on Algorithm Engineering (WAE 2001). Volume 2141 of Lecture Notes in Computer Science., Springer-Verlag (2001) 67–78
7. Chen, S., Gibbons, P.B., Mowry, T.C., Valentin, G.: Fractal prefetching B+-trees: Optimizing both cache and disk performance. In: 2002 ACM SIGMOD International Conference on Management of Data, ACM Press (2002) 157–168
8. Bender, M.A., Demaine, E.D., Farach-Colton, M.: Cache-oblivious B-trees. SIAM Journal on Computing **35**(2) (2005) 341–358
9. Brodal, G.S., Fagerberg, R., Jacob, R.: Cache oblivious search trees via binary trees of small height. In: 13th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2002), Society for Industrial and Applied Mathematics (2002) 39–48
10. Bender, M.A., Duan, Z., Iacono, J., Wu, J.: A locality-preserving cache-oblivious dynamic dictionary. In: 13th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2002), Society for Industrial and Applied Mathematics (2002) 29–38
11. Jiang, W., Ding, C., Cheng, R.: Memory access analysis and optimization approaches on splay trees. In: 7th Workshop on Languages, Compilers and Run-time Support for Scalable Systems, ACM Press (2004) 1–6
12. Oksanen, K., Malmi, L.: Memory reference locality and periodic relocation in main memory search trees. In: 5th Hellenic Conference of Informatics, Greek Computer Society (1995)

13. Bender, M.A., Demaine, E.D., Farach-Colton, M.: Efficient tree layout in a multilevel memory hierarchy. In: 10th Annual European Symposium on Algorithms (ESA 2002). Volume 2461 of Lecture Notes in Computer Science., Springer-Verlag (2002) 165–173
14. Bender, M.A., Brodal, G.S., Fagerberg, R., Ge, D., He, S., Hu, H., Iacono, J., López-Ortiz, A.: The cost of cache-oblivious searching. In: 44th Annual IEEE Symposium on Foundations of Computer Science (FOCS 2003), IEEE Computer Society (2003) 271–282
15. Soisalon-Soininen, E., Widmayer, P.: Concurrency and recovery in full-text indexing. In: String Processing and Information Retrieval Symposium (SPIRE 1999), IEEE Computer Society (1999) 192–198
16. Gray, J., Reuter, A.: Transaction Processing: Concepts and Techniques. Morgan Kaufmann (1993)