Getting to the Root of Concurrent Binary Search Tree Performance

Maya Arbel-Raviv, Technion; Trevor Brown, IST Austria; Adam Morrison, Tel Aviv University

https://www.usenix.org/conference/atc18/presentation/arbel-raviv

This paper is included in the Proceedings of the 2018 USENIX Annual Technical Conference (USENIX ATC ’18).

July 11–13, 2018 • Boston, MA, USA

ISBN 978-1-939133-02-1

Open access to the Proceedings of the 2018 USENIX Annual Technical Conference is sponsored by USENIX.
Getting to the Root of Concurrent Binary Search Tree Performance

Maya Arbel-Raviv
Technion

Trevor Brown
IST Austria

Adam Morrison
Tel Aviv University

Abstract

Many systems rely on optimistic concurrent search trees for multi-core scalability. In principle, optimistic trees have a simple performance story: searches are read-only and so run in parallel, with writes to shared memory occurring only when modifying the data structure. However, this paper shows that in practice, obtaining the full performance benefits of optimistic search trees is not so simple.

We focus on optimistic binary search trees (BSTs) and perform a detailed performance analysis of 10 state-of-the-art BSTs on large scale x86-64 hardware, using both microbenchmarks and an in-memory database system. We find and explain significant unexpected performance differences between BSTs with similar tree structure and search implementations, which we trace to subtle performance-degrading interactions of BSTs with systems software and hardware subsystems. We further derive a prescriptive approach to avoid this performance degradation, as well as algorithmic insights on optimistic BST design. Our work underlines the gap between the theory and practice of multi-core performance, and calls for further research to help bridge this gap.

1 Introduction

Many systems rely on optimistic concurrent search trees for multi-core scalability. (For example, in-memory databases [35], key/value stores [29], and OS virtual memory subsystems [10].) Optimistic search trees seem to have a simple performance story, based on the observation that to scale well a workload must contain sufficient high-level parallelism (e.g., operations should not all modify the same key [21]). Optimistic search trees therefore strive to avoid synchronization contention between operations that do not conflict semantically, such as updates to different keys. In particular, optimistic trees use read-only searches, which do not lock or otherwise write to traversed nodes, with writes to shared memory occurring only to modify the data structure [7, 29]. This design is considered key to search tree performance [12, 18].

We show, however, that realizing the full performance benefits of optimistic tree designs is far from simple, because their performance is affected by subtle interactions with systems software and hardware subsystems that are hard to identify and solve. To demonstrate this issue, consider the problem faced by systems designers who need to reason about data structure performance. Given that real-life search tree workloads operate on trees with millions of items and do not suffer from high contention [3, 26, 35], it is natural to assume that search performance will be a dominating factor. (After all, most of the time will be spent searching the tree, with synchronization—if any—happening only at the end of a search.) In particular, we would expect two trees with similar structure (and thus similar-length search paths), such as balanced trees with logarithmic height, to perform similarly.

In practice, however, this expectation turns out to be false. We test the above reasoning on optimistic binary search trees (BSTs), since there are BST designs with various tree structures [2, 7, 14, 15, 22, 23, 32, 33]. We find significant performance differences between BSTs with similar structure and traversal techniques. Figure 1a depicts examples of such anomalies. (We show a read-only workload, consisting only of lookups, to rule out synchronization as a cause. We detail the studied BSTs and experimental setup in §2.) For instance, one unbalanced internal BST (edge-int-lf) outperforms other BSTs with the same tree structure (log-int and citrus). There is even a significant difference between two implementations of the same BST algorithm (occ-avl and occ-avl-2).

The goal of this work is to explain and solve such unexpected performance results. We perform a detailed performance analysis of 10 state-of-the-art optimistic BST implementations on large scale x86-64 hardware, in which we uncover the root causes of the observed anomalies. Using microbenchmarks, we find that performance anomalies are caused by multiple performance-degrading interactions of BSTs with systems software and hardware subsystems, mostly related to cache effects. These cache effects are due either to cache-unfriendly implementation oversights or, more interestingly, to memory layout pathologies that are caused by interactions between the BST and the memory allocator. To determine whether our observations are only artifacts of micro benchmarking, or whether similar issues appear in more complex software, we deploy the BSTs as the index structure in DBx1000, an in-memory database [4, 27, 39, 40]. We find that similar anomalies exist in DBx1000 as well. Most importantly, we find that a simple approach of segregating BST-related allocations, so that BST data is not mixed with application data, improves performance of the BSTs by up to 20%
and of the overall application by up to 10%. Figure 1b

demonstrates part of our results.

Our work underlines the gap between the theory and practice of multi-core performance. As we show, it is non-trivial to understand a search tree’s performance, and specifically, whether performance is due to fundamental algorithmic factors or to implementation issues. While we focus on BSTs, the effects we uncover are relevant to other optimistic concurrent data structures, as they stem from general principles of memory allocator and systems design. (But we leave such analysis for future work.) Our results therefore call for further research to help bridge the gap between the principles and practice of multi-core performance, to simplify the task of deploying a concurrent data structure and reasoning about its performance.

2 Scope

2.1 BSTs

We analyze C implementations of 8 BST algorithms, two of which have independent implementations, for a total of 10 implementations. The algorithms implement the standard key/value-map operations, lookup, insert and remove. Table 1 lists the implementations studied. These BSTs span the known points in the design space, covering combinations of synchronization techniques, tree types (internal vs. external), and balancing choices (unbalanced vs. self-balancing algorithms).

All BSTs but int-lf feature read-only traversals; in int-lf, a traversal might synchronize with a concurrent update. In the lock-free BSTs, updates manipulate the data structure using atomic instructions, such as compare-and-swap (CAS), instead of synchronizing with locks. Both int-lf and ext-lf use operation descriptor objects to implement helping between their operations. A descriptor details the memory modifications that an update operation needs to perform. Before performing its modifications, the update operation CASes a pointer to its descriptor in each of the nodes it needs to update. Other operations that encounter the descriptor use the information therein to help the update complete. edge-int-lf

Table 1: BST implementations studied (ordered by the expected performance of searches).

<table>
<thead>
<tr>
<th>name</th>
<th>synchronization technique</th>
<th>tree type</th>
<th>self-balance</th>
<th>impl. source</th>
</tr>
</thead>
<tbody>
<tr>
<td>occ-avl†</td>
<td>fine-grained locks</td>
<td>part.</td>
<td>✓</td>
<td>ASCYLIB</td>
</tr>
<tr>
<td>occ-avl-2</td>
<td></td>
<td>ext</td>
<td></td>
<td></td>
</tr>
<tr>
<td>edge-int-lf [33]</td>
<td>lock-free</td>
<td>int</td>
<td></td>
<td>authors</td>
</tr>
<tr>
<td>log-int [14]</td>
<td>fine-grained locks</td>
<td>int</td>
<td></td>
<td>ASCYLIB</td>
</tr>
<tr>
<td>citrus [2]</td>
<td>fine-grained locks</td>
<td>int</td>
<td></td>
<td>authors</td>
</tr>
<tr>
<td>int-lf [23]</td>
<td>lock-free</td>
<td>int</td>
<td></td>
<td>ASCYLIB</td>
</tr>
<tr>
<td>edge-ext-lf [32]</td>
<td>lock-free</td>
<td>ext</td>
<td></td>
<td>authors</td>
</tr>
<tr>
<td>edge-ext-lf-2</td>
<td></td>
<td>ext</td>
<td></td>
<td>ASCYLIB</td>
</tr>
<tr>
<td>ticket [12]</td>
<td></td>
<td>ext</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ext-lf [15]</td>
<td>lock-free</td>
<td>ext</td>
<td></td>
<td>ASCYLIB</td>
</tr>
</tbody>
</table>

† authors refers to original authors’ implementation, and ASCYLIB to the implementation in the ASCYLIB library [12].

and edge-ext-lf avoid descriptors by “stealing” some bits from node left/right pointers to encode helping-related information.

An internal BST stores an item in every node, whereas an external BST stores items only in leaves. Internal BSTs have different solutions for removing a node with two children while maintaining consistency of concurrent searches. edge-int-lf and int-lf searches use validation to detect such a concurrent removal and restart the search. log-int avoids restarts by having an unsuccessful search (i.e., that fails to find its target key) traverse an ordered list which links all nodes, to verify that the key is indeed not present. Finally, occ-avl marks a node with two children as logically removed instead of physically removing it from the data structure, resulting in a partially external tree. occ-avl uses validation to restart a search that could take the wrong path due to a concurrent tree rotation.

The BSTs appear in Table 1 according to their expected relative performance in workloads where search time dominates performance: all else being equal, one expects self-balancing BSTs, which maintain logarithmic height, to outperform unbalanced BSTs; and internal BSTs to outperform external BSTs.

The original implementation of two of the BSTs [7, 14] is in Java. We choose, however, to evaluate 3rd-party C implementations of these BSTs, to obtain an apples-to-apples comparison and to simplify the analysis.

We fixed incorrect use of the C volatile keyword in some of the evaluated implementations. In general, to
We begin by analyzing BST performance on the standard

We perform experiments on two multi-socket x86 plat-

These remaining anomalies occur due to cache behaviour

differences due to BST memory layout (§ 3.1).

Most of the BST implementations contain one or more
of three implementation oversights that negatively impact
the performance of BST searches. Table 3 summarizes
our findings, which we discuss next:

<table>
<thead>
<tr>
<th>Bloated nodes</th>
<th>Scattered fields</th>
<th>Heavy traversals</th>
</tr>
</thead>
<tbody>
<tr>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>yes</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
</tbody>
</table>

Table 3: BST implementation issues.

avoid such problems, one should either use C
atomics, or place the volatile keyword correctly: a volatile
pointer to a node is written node * volatile ptr, not
volatile node * ptr.

2.2 Experimental setup

We perform experiments on two multi-socket x86 plat-
forms, by AMD and Intel. Table 2 details the hardware
characteristics of these platforms. Both machines are
NUMA platforms, configured so that DRAM is equally
divided between the NUMA nodes. When running exper-
iments, we use the standard practice of interchanging
the benchmark’s memory pages across the system’s NUMA
nodes (using the numact1 command) to prevent any
NUMA node from becoming a bottleneck. We compile
the benchmarks with gcc v4.8. As in prior work, we use
a scalable memory allocator (jemalloc [16]) to prevent
memory allocation from becoming a bottleneck.

3 BST performance in isolation

We begin by analyzing BST performance on the standard
microbenchmark used in the concurrency literature [2, 7, 11, 14, 22, 23, 32, 33], which models an application
using a BST. The benchmark consists of a loop in which
each thread repeatedly performs a random BST operation
on a random integer key, and its performance metric is
the obtained aggregate throughput of BST operations. We
find that several implementations make simple oversights
that lead to inefficient BST searches, but that fixing these
problems still leaves many unexpected results (§ 3.1).
These remaining anomalies occur due to cache behaviour

Scattered fields Fields commonly read by traversals
(key/left/right, as well as fields related to detecting concur-
rent tree modifications) should be located first in the node
structure, to minimize the chance that a search accesses
two cache lines when traversing a node.

Heavy traversals edge-int-lf and ext-lf base all opera-
tions on one shared traversal method, and so end up
burdening lookup operations with the book-keeping re-
quired only for updates, such as maintaining pointers to
the parent/grandparent of the current node.

3.1 Evaluating impact of implementation issues

We fix the above implementation issues by replacing
pthread mutex locks with spin locks, removing padding
and reordering node fields in the affected implementa-
tions, and evaluate the impact of these fixes.

Methodology Our benchmark is parameterized by the
distributions that the operation types and keys are chosen
from, the size of the key space, and the number of items
(key/value pairs) initially present in the tree. Following
the practice in the concurrency research literature, we (1)
choose operation keys uniformly at random; (2) perform
insert and remove with equal probability throughout
the benchmark; and (3) initialize the BSTs (using concur-
rent insert()s) with U/2 random items, where U is
the size of the key space. We report averages of five 3-second
runs on an otherwise idle system.

Results Figure 2 shows the performance impact of our
changes on trees that initially contain 1 M and 10 M items,
to model realistic working sets. We show results from
read-only (100% lookup) workloads so that we can rea-
son about search performance and remove synchroniza-
tion effects as a confounding factor. We have, however,
verified that read-only workloads are a good proxy for
read-dominated workloads on this benchmark: e.g., in
workloads with 90% lookups, the relative performance
order of the BSTs matches that of the read-only case al-
most perfectly and most comparison points remain similar
even at a 70% lookup rate.

We show results from executions with the maximum number of threads on each platform, as all BSTs scale with the amount of concurrency. On the 1 M-item tree, our fixes improve the throughput of the BSTs by up to 86% on abu-dhabi and by up to 43% on haswell, with a geo mean improvement of 11% on abu-dhabi and 23% on haswell. Moreover, reducing occ-avl’s node size brings its performance to the level of occ-avl-2.

**Unexpected results** Several unexpected results remain even after fixing the BST implementation issues, and we uncover their cause in the remainder of this section:

- Why does decreasing node size hurt throughput for 10 M-item int-lf, ticket and ext-lf on abu-dhabi? (§ 3.2.2)
- Why does int-lf benefit from a reduction of 64-byte nodes to 48-byte nodes much more than ticket on haswell? Why does log-int perform worse than the other unbalanced internal BSTs? (§ 3.2.3)
- Why does edge-ext-lf outperform other external BSTs, when they all have the same tree structure? (§ 3.2.4)
- Why do occ-avl and occ-avl-2, self-balancing BSTs, behave differently on abu-dhabi and haswell? On abu-dhabi they significantly outperform unbalanced trees (as expected), whereas on haswell they do not. (§ 3.3)

### 3.2 Memory layout issues

We trace most of the anomalies to memory layout issues that lead to different cache behaviours between the BSTs. These memory layout issues result from subtle interactions between the BST’s allocation pattern and the policies of the memory allocator, particularly the use of segregated free lists [24] for satisfying allocations.

#### 3.2.1 Segregated free list allocation

At a high level, scalable memory allocators [5, 16, 17] avoid contention by providing each thread with its own heap. These heaps are implemented as a set of free lists [24], one for each possible size class. Free lists are generally implemented as superblocks, which contain an array of blocks. To satisfy an n-byte allocation request, the allocator rounds n up to the nearest size class, s, and returns an s-byte block obtained from the relevant free list. In the jemalloc memory allocator we used for our experiments, the size classes used for allocations of up to 1 KiB are 8, 16, 32, 48, 64, 80, 96, 112, 128, 192, 256, 320, 384, 448, 512, and 1024. In addition to size classes, allocators differ in the structure and size of superblocks, the algorithm for mapping a block to its superblock, policies for allocating and releasing superblocks, and synchronization schemes. The important point in our context is that we can model the behaviour of the memory allocator as satisfying allocations of size s from an array of blocks of size s.

#### 3.2.2 Crossing cache lines

In the BSTs we study, visiting a node should in principle incur at most one cache miss: the size of the searched fields (key, child pointer, and any fields used to synchronize with concurrent updates) fit in one cache line. We find, however, that the memory allocator might place a node in memory so that these search fields straddle a cache line boundary, causing a visit to the node to incur 2 cache misses.

Consider, for example, a BST whose searches access the first 24 bytes of a node (8-byte key and 8-byte left or right child pointer). If its node size is 48 bytes and the memory allocator’s block array is cache line-aligned, then nodes will start at offsets 0, 16, 32, and 48 within cache lines. For the nodes at offset 48, the last 8 bytes of these searched fields extend into the next line, possibly leading to a cache miss. Such a miss occurs with probability 1/8, as one in 4 nodes straddles a cache line boundary, and a...
search reads each next pointer with probability 1/2.

Originally, int-lf, ticket and ext-lf do not experience such cache line crosses, as they have padded 64-byte nodes that the memory allocator allocates from a size class of 64-byte blocks. Decreasing their node size introduces this issue, whose performance impact is a trade-off that depends on the workload. At one end of the spectrum, if a smaller node size allows the entire tree to fit into the LLC, then one can eliminate cache misses altogether. At the other end, if the workload is such that almost every node traversed incurs a cache miss, then it is better to increase the node size to avoid crossing cache lines, as otherwise the expected number of cache misses per search increases by the expected number of nodes whose search incurs an extra miss (e.g., by $1 \frac{1}{5} \times$ for 48-byte nodes).

In our workloads, we observe a 17% (geo mean) throughput degradation on the 10-M item tree on abudhabi, but negligible overhead on the 1 M-item tree. We do not observe this anomaly on haswell because it has an adjacent-line prefetcher [36] that effectively doubles the cache line size and hides the effect of misses caused by cache line crossings.

### 3.2.2 Underutilized caches due to allocation pattern

We find that BST allocation patterns can lead to cache set underutilization,\(^2\) in which the workload uses some cache sets more than others, thereby leading to increased associativity misses on the overused cache sets. We identify two causes for underutilized cache sets. First, the memory allocator might place allocated nodes in memory so that they map to just a subset of the cache sets. More insidiously, even if the nodes cover all cache sets but are allocated next to cache lines containing useless data, then prefetching this data evicts useful nodes from the cache.

We demonstrate cache set underutilization that occurs in int-lf; log-inr has a similar issue, whose description we omit due to space constraints.

**int-lf analysis** We observe an anomaly on haswell, in which int-lf benefits from a reduction of 64-byte nodes to 48-byte nodes much more than ticket. We focus on the 1 M-item tree experiment. While performance counter data shows that int-lf’s throughput improvement with smaller nodes is correlated with reduced LLC miss rates, the 1 M-item tree should almost fit into haswell’s 30 MiB LLC even with bloated nodes. This points to a cache set underutilization problem, in which int-lf effectively runs as if with a smaller cache. We verify this hypothesis by computing the cache set indexes of each node,\(^3\) finding that the original int-lf implementation uses only 50% of the L2 and LLC sets. Next, we analyze int-lf’s allocation pattern to find the cause for this problem.

Like many lock-free algorithms, int-lf uses operation descriptors so that threads can help each other to complete their operations (see § 2.1). Each thread’s allocation pattern during the BST initialization is thus $NDNDND\ldots$, as each insert operation allocates a new node of size $N$ and a descriptor of size $D$. The size of both descriptors and int-lf’s original padded nodes is 64 bytes, the cache line size. Both allocation types are thus satisfied from the same allocator size class, and consequently, nodes occupy only even (or only odd) cache set indexes, utilizing only 50% of the available cache sets. (We note that int-lf intentionally does not free descriptors, to avoid an ABA problem\(^4\) on the descriptor-pointer field in the nodes. The idea is that if the content of this field only changes from one descriptor to another, an ABA problem cannot occur.)

**Fixing cache set underutilization** Shrinking int-lf’s node size as part of fixing its implementation oversights has the serendipitous effect of segregating node and descriptor allocations. As nodes and descriptor allocations become satisfied from different size classes, nodes occupy all cache sets. Moreover, only nodes are allocated from their size class, and so no prefetching of useless data occurs. To prevent cache set underutilization in a principled way, we explicitly segregate BST nodes by allocating them from a dedicated memory pool; see § 4 for details.

It remains to show that cache set underutilization is not only caused by mapping nodes to a strict subset of the cache. To this end, we modify the microbenchmark to add allocation calls of random sizes between BST operations. These random allocations break the benchmark’s regular allocation pattern, causing int-lf nodes to map to all cache sets. Nevertheless, unless we additionally disable prefetching,\(^5\) int-lf performs poorly. Table 4 shows the result of our experiments. Fixing cache set underutilization improves throughput by 40% on the 1 M-item tree.

### 3.2.4 Collocated children

We find that the high throughput obtained by edge-ext-lf compared to the other external BSTs is due to a fortunate allocation pattern, which causes many leaves to be

<table>
<thead>
<tr>
<th>int-lf variant</th>
<th>ops/sec</th>
<th>unused L1 sets</th>
<th>unused L2 sets</th>
<th>unused L3 sets</th>
</tr>
</thead>
<tbody>
<tr>
<td>64 b node</td>
<td>42.5M</td>
<td>1.6%</td>
<td>50.8%</td>
<td>50.8%</td>
</tr>
<tr>
<td>64 b node, w/ allocs</td>
<td>42.5M</td>
<td>1.6%</td>
<td>1.6%</td>
<td>1.6%</td>
</tr>
<tr>
<td>64 b node, w/ allocs, no prefetching</td>
<td>60.0M</td>
<td>1.6%</td>
<td>1.6%</td>
<td>1.6%</td>
</tr>
<tr>
<td>40 b node</td>
<td>60.0M</td>
<td>1.6%</td>
<td>1.6%</td>
<td>1.6%</td>
</tr>
</tbody>
</table>

Table 4: int-lf on haswell cache set usage (1 M-item BST).

\(^2\)An 2\(^{e}\)-way associative cache of size $2^e$ bytes with $2^i$ cache lines groups its slots into sets of size $2^{C-i-n}$. Bits $l + 1, \ldots, C - n + 1$ of an address determine its set index.

\(^3\)We compute LLC set indexes using the physical addresses of the nodes. Specifically, we use the techniques of [30, 38] to reverse engineer the mapping from physical address to haswell LLC cache slices.

\(^4\)An ABA problem occurs when a thread reads the same value (A) from a location twice, interpreting this to mean that the location has contained (A) at all times between the two reads, whereas between the two reads, the location was actually changed to (B) and back to (A).

\(^5\)Specifically, the L1 data cache prefetcher.
collocated on the same cache line with their parent.

down. "edge-ext-lf" is an external BST with immutable 32-byte
time. We observe throughput slowdowns of 14% nodes: an insert whose search completes at leaf u allocates
an internal (routing) node v and a new leaf node (with the inserted item) w, which is a child of v. It then
replaces u with v. The memory allocator satisfies node allocations from a superblock of 32-byte blocks. Therefore, v
and w might be collocated on the same cache line.

We analyze the node addresses in the evaluated trees and find that 75% of the internal nodes which have a leaf
child are also collocated in the same cache line with one of their children. (This collocation can only occur for leaves. Whenever "edge-ext-lf" extends a path, it breaks the previous parent/child collocation.)

To evaluate the performance impact of the collocation property, we implement shifted versions of "edge-ext-lf", where we add one 32-byte allocation before the initialization of the tree. This shifts the cache line offsets of all later allocations, moving the child nodes to a different cache line (Figure 3). As expected, we find that in the shifted implementations, 75% of internal nodes which have a leaf child have a child located on the adjacent cache line. On a 1 M-item BST, we observe throughput slowdowns of 14% and 11% on "abu-dhabi" and haswell, respectively.

The reason that prefetching does not hide this problem is again due to the allocation pattern. We examine the node addresses and find that 100% of the nodes which have a leaf child in the next cache line are themselves located on an odd cache line (Figure 3). The adjacent-line prefetcher on haswell "fetches the cache line that comprises a cache line pair" [36]. This appears to imply that it is only triggered on accesses to an even cache line, and thus is ineffective in this case.

3.3 Prefetching issues

Bronson et al.'s relaxed balance AVL BST [7] ("occ-avl
and "occ-avl-2") is considered as one the fastest BSTs. While on "abu-dhabi" the AVL tree indeed outperforms
the other BSTs by a geo mean of 40% in both tree sizes, on haswell it is not the best performer on the 1 M-item tree experiment. We trace this anomaly to a novel interaction of the BST's optimistic concurrency control (OCC) and the L2 prefetcher, which is exposed after removing a different bottleneck in the OCC implementation.

The algorithm uses versioning—an OCC implemen-
tation technique—to detect concurrent tree modifications during searches. Glossing over some details, each child pointer has an associated version number that increases when the pointer is updated. Observing that this version has not changed between time t0 and t1 allows a search to verify that the associated pointer has not changed as well. Searches use this property to verify that they traverse through a node only if both the inbound pointer to u and the outbound pointer to the next node on the path were valid together at the same point in time.

When the validation at some node u fails, the search starts ascending along the traversed path, revalidating at each node, until it returns to a consistent state from which it resumes the search. Both "occ-avl" and "occ-avl-2" use recursive calls to visit nodes, thereby recording this bookkeeping data on the stack. This information, however, is used only if a search encounters a concurrent update, which is expected to be a rare event. We therefore change "occ-avl-2" to restart the search from the root when validation fails, yielding the "occ-avl-2-ret" implementation.

In the 1 M-item tree experiment, "occ-avl-2-ret" outperforms "occ-avl" and "occ-avl-2" by 17% on haswell. We observe, however, that it generates many L2 prefetch misses. Our workload does not benefit from hardware prefetching, since the next cache line a BST search visits is random. Prefetching thus hurts BST throughput, as it evicts potentially useful tree nodes (e.g., nodes at the top of the tree) from the cache.

We find that reading the version stored in the nodes triggers an L2 prefetch. While reading twice from the cache line (key and next pointer) does not trigger prefetching, any additional read from the node does. To evaluate the impact of prefetching, we implement a variant of the algorithm without the version reads, "occ-avl-2-unsafe". This variant is safe to run only in a read-only workload; we use it just to estimate the performance lost due to prefetching. Figure 4 shows the results: On a 1 M-item tree, "occ-avl-2-unsafe" improves a further 12% over "occ-avl-2-ret", for an overall 31% improvement over "occ-avl-2"; its total improvement over "occ-avl-2" on a 10 M-item BST is 21%.

4 BST performance in an application

The next logical step is to ask whether these performance issues are simply artifacts of micro benchmarking, or whether similar issues appear in more complex software. To this end, we study an in-memory database management system called DBx1000 [39] (henceforth, simply DBx), which is used in multi-core database research [4, 27, 39,
In this section, we focus on the haswell machine.

DBx DBx implements a single relational database, which contains one or more tables, each of which consists of a sequence of rows. It offers a variety of different concurrency control mechanisms for allowing processes to access tables and rows. We use its 2-phased locking option, which locks individual rows of tables, and has been shown to scale on simulated systems containing up to one thousand cores [39].

Each table can have one or more key fields and associated indexes. Each index allows processes to query a specific key field, quickly locating any rows in which the key field contains a desired value. Any thread-safe data structure can serve as an index in DBx, as long as it implements a multimap. A multimap represents a set of keys, each of which maps to one or more values (pointers to rows of a table), and offers three operations: search(key), insert(key, value) and remove(key, value). search(key) returns all of the values to which key maps in the multimap. insert(key, value) adds a mapping from key to value. If key maps to value, then remove(key, value) removes the mapping from key to value, and returns true. Otherwise, it simply returns false.

Methodology We replace the default index implementation in DBx with each of the BSTs that we study. To do so, we had to overcome a minor complication: each of these BSTs implements a map, not a multimap. That is, each key maps only to a single value. We transformed these maps into multmaps as follows. Instead of storing keys and pointers to rows in the map, each key maps to the head of a linked list that is protected by a lock. (The locks are stored in a separate lock table.) Then, to perform insert(key, value) on the multimap, where value is a pointer to a row, we simply insert row into the appropriate linked list in the underlying map.

Workloads To analyze the performance of the various BST implementations in DBx, we use the well known Yahoo! Cloud Serving Benchmark (YCSB), and the Transaction Processing Performance Council’s TPC-C benchmark. The relatively simple transactions in DBx with each of the BSTs that we study. To do so, we had to overcome a minor complication: each of these BSTs implements a map, not a multimap. That is, each key maps only to a single value. We transformed these maps into multmaps as follows. Instead of storing keys and pointers to rows in the map, each key maps to the head of a linked list that is protected by a lock. (The locks are stored in a separate lock table.) Then, to perform insert(key, value) on the multimap, where value is a pointer to a row, we simply insert row into the appropriate linked list in the underlying map.

In all of our experiments, we measure the number of committed transactions, the number of index operations performed, the time needed to perform all transactions (total time), and the time spent accessing the index(es) (index time). Timing measurements were performed using x86-64 RDTSC instructions. The overall performance of a benchmark is measured in terms of transaction throughput, the total number of committed transactions divided by total time. We define dbx time as the time in an execution that is not spent accessing the index(es) (i.e., total time — index time).

4.1 YCSB

Following the approach in [39], we run a subset of the YCSB core with a single table containing ten million rows. Each thread performs a fixed number of transactions (100,000 in our runs), and the execution terminates when the first thread finishes performing its transactions. Each transaction accesses 16 different rows in the table, which are determined by index lookups on randomly generated keys. Each row is read with probability 0.9 and updated with probability 0.1. The keys are generated according to a Zipfian distribution following the approach in [19].

Segregating tree data When we use a BST implementation as the index in YCSB, we are effectively merging the memory address space of YCSB with the address space of the BST. In doing so, we may change the memory layout of objects in YCSB (for example, by interleaving nodes with table rows in YCSB), which can have a significant impact on performance. We can isolate and study these memory layout changes, and selectively eliminate them, by using segregation to effectively separate parts of the address spaces for the BST and YCSB.

In a real application, it can be difficult to segregate simply by changing object sizes, so we implement segregation by using several separate instances of the memory allocator: one for YCSB, and one for each type of objects we would like to segregate from other object types. In our case, this means one for BST nodes, one for BST descriptors, and one for other implementation specific tree data. Consequently, when we segregate tree data, nodes are allocated consecutively in each page, descriptors are not interleaved with nodes (avoiding the performance problem with int-lf in § 3.2.3), and tree data is not interleaved with YCSB data.

4.2 Comparison with the microbenchmark

We first address the question: to what degree do the results of YCSB match our microbenchmark results? We compare with microbenchmark results for trees containing 10 million keys, since this is approximately the size of the index in YCSB. The left side of Figure 5 contains the results of running the microbenchmark for all of the BSTs we studied, after fixing all of the performance issues described. To make the results easier to understand, we sort the BSTs by performance and group them into the following equivalence classes: (occ-avl, occ-avl-2), (log-int, edge-int-lf, citrus, int-lf, edge-ext-lf-2, edge-ext-lf), (ticket), (ext-lf). Within each of these equivalence classes, the performance differences are not significant.

The results of our YCSB experiments appear in Figure 5. The BSTs are listed in the same order as they appear in the microbenchmarks. Without segregation (middle
Figure 5: Microbenchmark compared to YCSB results: (left) Microbenchmark for 10M item BSTs (middle) YCSB without segregation, (right) YCSB with segregation.

graph) there are several differences between the YCSB results and the microbenchmark results. First, log-int performs about as well as occ-avl and occ-avl-2, which were significantly faster than log-int in the microbenchmarks. Here, it appears that log-int belongs in the same equivalence class as occ-avl and occ-avl-2. Second, edge-ext-lf-2 is significantly slower than edge-ext-lf, whereas they have the same performance in the microbenchmark. Third, ticket and ext-lf have the same performance, whereas ticket is significantly faster in the microbenchmark. As the graph on the right shows, segregating the tree data brings the results closer to the original behaviour observed in the microbenchmark.

4.3 Memory layout issues

In our analysis of YCSB, we found several memory layout issues that were similar to the issues we found in our microbenchmarks. We describe a few key examples.

4.3.1 Underutilized caches due to allocation pattern

When we add all of our BST implementations to YCSB, several of them exhibit very poor cache set utilization. We find that their nodes map to only 1/3rd of the L3 cache sets, rendering 2/3rds of the L3 cache unusable for the storing nodes. These implementations include occ-avl and occ-avl-2, which have 64-byte nodes. Only implementations with 64-byte nodes were affected.

Since we did not observe this behaviour in the microbenchmarks, we hypothesize it is the result of adding these trees to YCSB (more specifically, merging each tree’s memory space with the memory space of YCSB). We analyze the allocations performed by YCSB, and find that it allocates a large number (millions) of objects in size classes: 8, 32, 48, 64, 128, 192 and 384. In the 64-byte size class, it allocates only row and row wrapper objects. In particular, it always allocates a row, followed by a row wrapper, and then inserts the row into the index (BST). In the BSTs that exhibit this memory layout problem, index insertion allocates one 64-byte node. Thus, the allocation pattern in memory is RWNRWN... where R is a row, W is a row wrapper, and N is a node. Consequently, rows have addresses satisfying \(\text{addr} = 0 \pmod{192}\), row wrappers have addresses satisfying \(\text{addr} = 64 \pmod{192}\) and nodes have addresses satisfying \(\text{addr} = 128 \pmod{192}\). That is, each object type has a 192-byte stride.

This pattern turns out to have a pathological interaction with the processor’s internal hash function that maps physical addresses to L3 cache sets, resulting in an execution where rows, row wrappers and nodes each map to only 1/3rd of the L3 cache sets. (This is similar to how we saw a memory layout anomaly with a 128-byte stride in §3.2.3.) In contrast, if a particular object type appears with a 256-byte stride, the L3 hash function will map objects approximately uniformly over all cache sets.

We break up this deleterious allocation pattern by segregating the tree data. This segregation results in a significant speedup for these data structures, since it allows nodes to occupy the entire cache. For example, in occ-avl, it reduces index time from 121 to 108 seconds (a 13 second difference), and total time from 188 to 178 seconds (a 10 second difference). Note, however, that it increases dbx time by 3 seconds. Further timing measurements demonstrate that the increase in dbx time is due to added contention on row locks. In fact, we can show that whenever segregation increased dbx time in YCSB, the increase is due to added contention on row locks.

Perhaps surprisingly, the deleterious allocation pattern we saw above did not affect ticket, which has 64-byte nodes, or a variant of ext-lf with 64-byte nodes. (These were the only other implementations with 64-byte nodes.) The explanation turns out to be fairly simple. Although their nodes are 64 bytes, these trees are external, so they allocate two nodes per insertion operation, producing the allocation pattern RWNRWNWRWN. The second node allocation breaks up the (pathological) 192-byte strides that we saw above.

4.3.2 Accidentally fixing a memory layout problem

In the previous section, we saw how merging two address spaces can cause a memory layout issue. In this
section, we see how merging two address spaces can fix a preexisting memory layout issue.

When adding BSTs with 48-byte nodes to YCSB, and experimenting to see how much segregation helps, we find that segregating tree data for these BSTs caused significant increases in \textit{dbx time}. For example, segregation increases \textit{dbx time} for \textit{edge-int-lf} by 9 seconds, from 58 to 67. Analyzing executions of YCSB, we find that approximately ten million row locks (implemented with \texttt{pthread} mutexes) and 4,000 other miscellaneous objects are allocated in the 48-byte size class (in addition to any 48-byte nodes).

If there are no 48-byte node allocations, then these 48-byte row locks experience false sharing. Since the locks are smaller than a cache line, and they are allocated consecutively, a single cache line contains parts of two different locks. Thus, write contention on one lock additionally creates write contention on another lock. This is exacerbated by the adjacent line prefetcher, which effectively causes accesses to a lock to contend with the (three to four) locks stored in two cache lines. By merging the address space of a BST with 48-byte nodes with the address space of YCSB, we accidentally mitigated this false sharing by interleaving row locks with nodes. Of course, this unfairly favours the BSTs with 48-byte nodes over the other BSTs. Thus, we fix this problem in a more principled way by padding the row locks to eliminate false sharing. (The same effect was seen, and fixed, in TPC-C.)

### 4.3.3 Unnecessary page scattering

So far, we have seen that segregation can improve performance by breaking up deleterious memory layouts, and generally improving cache behaviour. Our results have thus far suggested that we can reasonably expect to see some change in performance due to segregation whenever nodes are allocated from the same size class as some other objects. However, it turns out that segregation can improve performance, even when nodes are the only allocations performed from a given size class.

Once the row locks in YCSB are padded, YCSB only performs about 4,000 miscellaneous allocations from the 48-byte size class. Thus, in BSTs with 48-byte nodes, nodes are the only significant source of allocations in their size class. We were quite surprised to find that segregation significantly improved performance for these trees.

One interesting difference caused by segregation is a substantial reduction in the TLB miss rate for algorithms with 48-byte nodes. This improvement comes from an interaction between huge pages and the allocator. When huge pages are enabled in Linux, pages occupy 2MB instead of 4096 bytes. This generally improves TLB miss rates, since a program’s working set can be represented using fewer pages.

However, we found that the allocator \texttt{jemalloc} divides each huge page into 512 chunks of 4096 bytes each, and distributes these into different size classes. More specifically, during its initialization, \texttt{jemalloc} allocates a bank of chunks for each thread. Each thread distributes these chunks \textit{on-demand} to its individual size classes. Whenever a thread runs out of space in the current chunk for one of its size classes, it fetches one (or more) chunks from its bank, and assigns them to this size class. In our experiments, we observed that threads fetch one chunk at a time for the 32-, 48- and 64-byte size classes.

In YCSB, this has the following effect. Before performing insertion on a BST, a transaction allocates a 64-byte row, followed by 128 bytes of data, a 64-byte row wrapper, a 192-byte (padded) \texttt{pthread} mutex, and a 32-byte value. Thus, for each node allocated by an insertion operation, several objects are allocated in several different size classes. Consequently, each thread regularly takes chunks from its bank and assigns them to these size classes, almost in round-robin fashion, but with more chunks going to the size classes that exhaust them more quickly.

As a result, in the small size classes used for nodes, the chunks often do not have consecutive addresses. For example, in a variant of \textit{edge-int-lf} with 48-byte nodes, we found that threads would allocate full 4096-byte chunks of nodes, but would only store nodes in approximately one out of every 10 chunks that it allocated. As another example, in \textit{occ-avl-2}, which has 64-byte nodes, threads would use up to three consecutive chunks to store nodes, and then the next chunk used to store nodes would typically appear five or six chunks later in the address space.

Figure 6 visualizes the actual layout of chunks used to store nodes in an execution of YCSB with \textit{occ-avl-2}.

We now consider what happens when the tree data for \textit{occ-avl-2} is segregated. Figure 7 shows the resulting layout of chunks used to store nodes. The difference is striking. Since the nodes are allocated by a separate instance of \texttt{jemalloc}, each thread uses its entire bank of chunks to store nodes. Consequently, the chunks allocated for nodes almost always have consecutive addresses. This
significantly reduces the number of pages needed to store the tree, and results in far fewer TLB misses. In YCSB with occ-avl-2, the average number of TLB misses per YCSB transaction decreases from 412 to 161 (a 61% reduction). Segregation reduces TLB misses in all of the BSTs we studied.

4.4 TPC-C

TPC-C simulates a large scale online transaction processing application for the order-entry environment of a wholesale supplier. According to the Transaction Processing Performance Council, it represents the business activity of “any industry that must manage, sell, or distribute a product or service.” At a high level, TPC-C assumes that business operations are organized around a fixed number of warehouses, which each service a number of districts. For each warehouse and district, the database stores information about customers, orders, payments, items for sale, and warehouse stock. TPC-C features complex transactions over nine tables with widely varying row types and population sizes, and with varying degrees of non-uniformity in the data. These tables are indexed by up to three different indexes on different key fields.

Our implementation of TPC-C executes a representative subset of the TPC-C transactions. In particular, we include the new-order and payment transactions, which comprise 88% of all transactions executed in the full TPC-C benchmark. This same approach was taken in [39].

Note that payment transactions update data in the warehouse table, and thus contend with all transactions operating on the same warehouse. Consequently, concurrency in TPC-C is limited by the number of warehouses. Thus, it is common to run with at least as many warehouses as there are concurrent threads in the experimental system. We run with 48 warehouses.

Segregating tree data As in YCSB, we segregate the tree data by using several allocator instances: one for TPC-C, one for BST nodes, one for descriptors, and one for other implementation specific tree data. All indexes share the same allocators. So, for example, all indexes use the same allocator for nodes. Thus, nodes for all indexes are interleaved with one another, but not with TPC-C data.

4.5 Impact of improved BSTs on TPC-C

We now present an experiment that demonstrates the impact of our improvements to the BSTs on the performance of TPC-C. The results appear in Figure 8. We obtain each data point by dividing the throughput of TPC-C when the final BST implementation is used for indexes (with segregation) by the throughput when the baseline BST implementation (without segregation) is used for indexes.

By improving the BST implementations, we obtain an overall improvement of up to 7.6%. Initially, this improvement might seem somewhat small, but TPC-C is a large, complex workload that takes over 200 seconds to run, and allocates over 30 GiB of memory. Accesses to the indexes comprise a relatively small part of the work, and Amdahl’s law limits the improvement we can see, so a 7.6% overall improvement is actually fairly substantial.

Source of the improvement Let us drill down into the details of where this improvement comes from. As an example, we consider occ-avl (which obtains the full 7.6% improvement). With the baseline implementation of occ-avl, the total time to run TPC-C is 249 seconds. This breaks down into 108 seconds of index time and 141 seconds of dbx time. If we follow the recommendations in §3, then total time decreases by 9 seconds to 240. This breaks down into 105 seconds of index time and 135 seconds of dbx time. If we additionally segregate tree data, then total time further decreases by 8 seconds to 232. This breaks down into 95 seconds of index time and 137 seconds of dbx time.

Interestingly, segregation causes a slight increase in dbx time. It turns out that, when the indexes in DBx speed up significantly, a new bottleneck appears. This manifests as increased contention on row locks. However, this is not the only component of the increase in dbx time. DBx and TPC-C are quite complex, and there is an additional component that we are unable to identify. We leave it as future work to perform additional profiling of DBx.

4.6 Impact of segregation on TPC-C

We now study the effect of segregation on the other BSTs. Figure 9 shows the breakdown of TPC-C total time into index time and dbx time both with and without segregation.
whether they would address the anomalies we consider.

25 TLB utilization. Lattner and Adve [72x254]

Some of the phenomena we find

5 Related work

structures into separate pools. Their approach does not

be required to avoid underutilizing cache sets.

Understanding performance Several studies compare

the performance of concurrent data structures [12, 18], but
do not analyze the root causes of performance differences.
Our work is complementary to research on the difficulties
of understanding experimental evaluation results [6, 13,
31], which does not consider concurrent data structures.

6 Discussion

We believe that the lessons learned in this work can be

applied to other concurrent data structures, as they stem
from general performance principles. Here, we attempt to

distill these lessons into concrete recommendations.

Data structure designers and implementers: Study the

memory layout of the data structure. If cache lines ad-

jacent to nodes often contain other objects, then the

cache may be underutilized by nodes. Pad objects to se-

parate them into different allocator size classes. Padding

should also be used to avoid false sharing, particularly be-

tween frequently-accessed nodes and other program data.

Such padding should take prefetching (e.g., the adjacent

line prefetcher) into account. However, indiscriminately

padding all nodes may reduce performance, since this

reduces the number of nodes that fit in the LLC. Finally,

watch for and avoid the implementation problems in § 3.1.

Programmers using a data structure: Importing a data

structure into a program merges two memory spaces, and

may create or eliminate false sharing or cache underuti-

lization problems. Thus, one should either (a) inspect the

combined memory layout of the data structure and the

program, and fix such problems, or (b) segregate the data

structure’s memory by using a separate allocator.

Memory allocator designers and implementers: The

above recommendations would be substantially easier to

put into practice with additional support from memory

allocators. First, providing an interface for allocation seg-

regation. Second, providing interfaces or tools for memory

layout inspection, to allow determining (1) the mapping

of objects to size classes; (2) which object types are fre-

quently located close to one another in memory (where

close could mean in the same cache line, or in adjacent

cache lines, or in the same page); and (3) the distribution

of objects into cache sets in the LLC (for each object type).

Such queries could also lead to high quality automated
tools for identifying memory layout problems.

Acknowledgments

This work was funded in part by the ISF (grants 2005/17 &
1749/14). Maya Arbel-Raviv was supported in part by the
Technion Hasso Plattner Institute Research School. Trevor
Brown was funded by PhD and post-doctoral fellowships
from NSERC, and grant no. RGPIN-2015-05080. Adam
Morrison was supported by Len Blavatnik and the Blavat-
nik Family Foundation.

Figure 9: Impact of segregation on TPC-C.
References


