An Evaluation of Search Tree Techniques In The Presence of Caches

Costin Iancu, Anurag Acharya
Department of Computer Science, University of California, Santa Barbara
{ciancu,acha}@cs.ucsb.edu

Abstract

Two techniques underlie the design of commonly used search-tree algorithms: move commonly accessed items closer to the root (*move-to-front*) and keep the tree balanced (*keep-it-balanced*). The move-to-front technique tries to improve the performance for the common case by reducing the number of operations required to retrieve commonly accessed items. The keep-it-balanced technique tries to improve worst-case performance by reducing the maximum number of operations required to look up an item in the tree. In this paper, we evaluate these techniques in the presence of a cache hierarchy. As representatives of move-to-front algorithms we use splay trees. As representatives of keep-it-balanced algorithms we use B-Trees and B*-Trees. In addition to classic versions of these techniques, we evaluate variants that have been optimized for a cache hierarchy. To drive our evaluation, we use a suite of synthetic datasets that were constructed to elucidate the differences between these techniques. These datasets vary primarily in the degree of locality found in the request stream and the operation mix. The major result of this paper is a qualitative performance difference analysis of the two classes of algorithms based on the expected data input characteristics. Based on this analysis we can predict which algorithm performs faster for a large class of input datasets. We also present guidelines for choosing implementation parameters in the presence of a cache hierarchy for all algorithms considered.

I. Introduction

The relative distance between the processor and the memory is large in contemporary architectures and continues to grow. To bridge this growing gap, computer architects are using increasingly large and deep cache hierarchies. Making good use of the cache hierarchy is crucial for achieving good overall performance. This can be particularly important for pointer-rich algorithms, such as tries, search trees, graph-partitioning etc., which tend to have irregular memory access patterns. To be able to make educated choices between such algorithms, we need to evaluate their performance in the presence of caches.

Two techniques underlie the design of a large number of search-tree algorithms [3], [4], [5], [6], [8], [9], [10], [12], [14], [15]: *move-to-front* - move commonly accessed items closer to the root and *keep-it-balanced* - keep the tree balanced. In this paper, we evaluate these techniques in the presence of caches. In addition to classic versions of these techniques, we evaluate variants that have been optimized for a cache hierarchy. To represent the move-to-front technique, we use two algorithms: classic splay trees as described by Sleator&Tarjan [15] and a cache optimized version of splay trees we developed for this study. Our algorithm tries to take advantage of the cache hierarchy by packing related nodes into the same cache line. As representatives of the keep-it-balanced technique, we evaluate two algorithms: classic B-trees as described by Bayer&McCreight [6], and a variant of B-trees described by Knuth [10] that tries to increase the average number of keys contained in each node [10].

The move-to-front technique [4], [12], [14], [15] tries to improve the average-case performance by reducing the number of operations required to retrieve commonly accessed items. On the other hand, the keep-it-balanced technique [3], [5], [6], [9], [10] tries to improve the worst-case performance by keeping the tree balanced and thereby reducing the maximum number of operations required to look up an item in the tree. To drive our evaluation, we use a suite of synthetic datasets that were constructed to elucidate the differences between these techniques. To exercise the differences, our datasets vary primarily in the degree of locality found in the request stream. To represent applications with little or no locality in the request stream, we use a dataset with uniformly distributed requests. To represent applications with varying degrees of locality in the request stream, we use normally distributed datasets with the same mean and different standard deviations (the σ/μ for these datasets varies between 0.01 and 0.72). To represent applications with an extremely high degree of locality (each item is repeatedly requested), we use a highly skewed dataset that consists of the first 25 million file requests from a month-long trace of file-system requests on a cluster of machines at UC Berkeley. To exploit the locality intrinsic to the move-to-front algorithms we generated data sets with different lookup-insert ratios : $95\% - 5\%$ (lookup), $60\% - 40\%$ (lookup dominated), $30\% - 70\%$ (insertion dominated).

The experimental results obtained show that, for most workloads, classical B-trees outperform all other alternatives we considered. In the presence of a cache hierarchy, the *move-to-front* characteristic exhibited by splay trees and their variants is implicitly played by the memory hierarchy itself. The better memory hierarchy behavior of B-trees is able to compensate for their higher operational complexity and this makes B-tree based implementations to run two to three times faster than their splay tree based counterparts. If a splay tree implementation is already used, the addition of cache oriented optimizations as presented in Section II-A, is likely to offer speed-up improvements. We obtained for our implementation speed-ups in the range 1% to 41%.

For benchmarks with very high locality characteristics, we found splay trees to outperform B-trees. We introduce in Section III-A a dataset locality metric. Based on this metric and the performance analysis introduced in Section IV-A we are able to predict which data structure is likely to perform better for a particular dataset.

The rest of this paper is structured as follows. In Section II we present the design and the implementation details of the algorithms we considered. Throughout Section III we present the experimental setup, the datasets we considered and their characteristics and the experiments we performed to tune our implementations. In Section IV we present detailed experimental results for our study and in Section IV-A we present our qualitative analysis and its experimental validation. Section V covers the related work and we present our conclusions in Section VI.

II. Algorithms

In this section, we describe the algorithms we evaluated. For *all* algorithms used in this study, we used customized memory allocators which ensure that the data structures being allocated are suitably aligned in cache lines. These allocators were also able to achieve significantly better performance than a general purpose allocator, such as *malloc()*, as they were tuned to allocate records of a fixed size and in general minimize the overhead of memory allocation.

In the rest of this section a brief description of each data structure is given and the implementation parameters that influence the runtime performance are presented. The reader is assumed to be familiar with the implementation details of splay trees and B-trees.

A. Splay Tree and Compacted Splay Tree

Splay trees [15], [4] are binary trees that adjust themselves at run-time depending on the value of the key that is inserted or searched in the tree. The adjustment is done by the "splaying" operation. A splaying operation *splay(key,tree)* restructures the tree in such a way that the node containing the *key* (or a node which would be a neighbor of key if key is not in the tree) ends up being the root of the tree.

For any basic operation on a splay tree, a splaying step is performed which moves the key or a neighbor to the tree root. In the splaying process, once a tree node is visited, all of its direct descendants are visited, in the case where the node does not contain the key or it is not a leaf. In the presence of a cache hierarchy the runtime performance of the algorithm is heavily influenced by the memory layout of the tree. In Figure 1 two alternate layouts are presented. In the first layout Figure 1 (a), visiting node k and its descendants y, z will determine accesses to `three` cache lines. In the second layout Figure 1 (b), visiting node k and its descendants will access only `one` cache

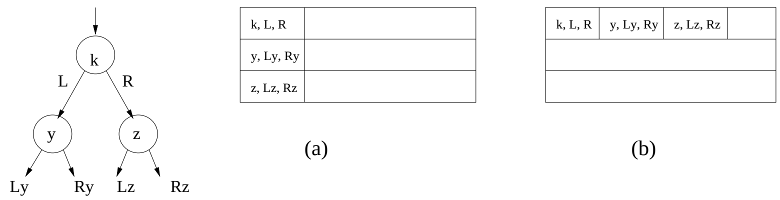


Fig. 1. Number of cache lines accessed when visiting a tree node. (a) - worst-case layout (b) - best-case layout

line. In the rest of this study we will refer to the second layout as *packed* layout. The memory layout of a splay tree is determined by the program allocation pattern and by the tree restructuring done by the splaying operation.

In order to minimize the number of cache misses during a tree traversal, our implementation tries to place nodes in memory as close to the *packed* layout as possible, i.e. a node will be contained in the same cache line with as many of its direct descendants as possible. Also, when allocated, tree nodes are aligned at the cache line boundary. We expect to obtain a better behavior by using a packed layout for two reasons. First, the reference to a node will automatically load into the cache its descendants and the subsequent accesses will hit in the cache. Second, by enforcing this layout we minimize the number of conflict misses with other unrelated tree nodes.

To position nodes in the cache conformant to a packed layout we implemented a custom memory allocator that compacts the splay tree using a technique similar to copying garbage-collection. While a copying garbage collection is usually executed when the application runs out of memory, our compactor is executed when we determine that the cache layout is likely to be poor.

When the allocator is initialized, a large memory block aligned on the L2 cache line boundary is allocated. This block is further subdivided in two subspaces - *allocation* space and *compaction* space. At any given time, only one of the two subspaces is used for allocating new nodes and is referred to as the *allocation* space. When a compaction is performed, the tree is traversed starting from the root and all nodes are moved from the *allocation* space to the *compaction* space. The compaction operation tries to pack a node and as many of its successors as possible in a single cache line. This is accomplished by a breadth first traversal of the tree that moves nodes and their direct successors into the same cache line. After a compaction the two spaces switch roles.

In order to determine when to perform a compaction we use a heuristic based on the size of the search space (number of keys) and on the amount of work performed by the algorithm since the last compaction operation. There are three aspects taken into account.

Degradation of cache layout: Consider a sequence of operations insert-lookup. For each insertion operation a new tree node is created. Since its successors are created at different times and are determined by the tree structure it is very likely that they will not occupy the same cache line with the node. For each lookup or insertion operation, rotations will occur in the tree changing the tree structure. These rotations perturb the cache layout as they change node descendance relationships.

Based on these observations we chose the following predicate to trigger a compaction :

$$(num\_nodes - old\_nodes) > old\_nodes/2 \;\; \texttt{\&\&} \;\; nrot/log(nrot) > old\_nodes/2$$

where `num_nodes` - number of nodes in the tree, `old_nodes` - number of nodes in the tree at the time of the last compaction, `nrot` - number of rotations since the last compaction.

The first term of the predicate takes into account the number of insertions since the last compaction. As the tree grows, each new insertion has a smaller impact on its layout. This term states that a new compaction is done only after the tree grew at least 50%. Also, this term makes compactions to occur less frequently as the tree size grows.

The second term is a measure of the cache layout perturbances as introduced by the rotation operations. It was chosen to be a function less than linear in the number of rotations.

Startup phase: In order to avoid unnecessary compactions in the startup phase, i.e. the problem size is still small and comparable to the L1 cache size, we have a threshold on the tree size before the first compaction is attempted. This parameter is referred to as `initCompactThreshold` and is set in terms of the number of nodes inserted in the tree. Its value was empirically chosen to be 30000, which accounts for a working set[1] size of 256KB. The algorithm sensitivity for values of the parameter in this range is minimal.

No memory available: If the size of the tree grows to fill most of the *allocation* space, the allocator doubles the amount of memory available (using `malloc()`) and doubles the size of each space. It detects this condition by checking the size of the tree right after the compaction step completes.

[1]The total amount of memory consumed by the data structure.

deg	k1 ....kdeg	C0, C1 ....Cdeg

(a) **`nodeContig`**

deg	C0, k1, C1, k2, ...kdeg, Cdeg

(b) **`nodeAlt`**

Fig. 2. Data layouts for a multiway three node.

B. B-Tree and B-Tree*

B-Trees [6] are multiway search trees with the following properties: *1) every node has* $\leq m$ *descendants; 2) every node except for the root and the leaves, has* $\geq m/2$ *sons; 3) the root has at least 2 descendants (unless it is a leaf); 4) all leaves appear on the same level; 5) a non-leaf node with* k *descendants contains* k-1 *keys.* For a detailed description of the basic operations on a B-Tree (insert, delete, search) the reader is referred to [8].

The B*-Tree data structure was first introduced by Bayer and McCreight [6] as an improvement to the classic B-Tree data structure. B*-Trees are also presented by Knuth [10]. Bayer and McCreight introduced the concept of “overflow”, and they try to improve the classic B-Tree algorithm by replacing node split operations with local rotations. The classical B-tree algorithm splits the node on overflow and inserts the median value in the parent of the node being split. The B*-tree algorithm first attempts to move some of the keys of the node to be split to its right or left neighbor. If this is unsuccessful, it reverts to splitting the node as the B-tree algorithm does. Also, In order to allow a higher node-occupancy, the rules for the classic B-Tree node structure are changed in the B*-Tree case:*1) every node except the root has at most* m *sons; 2) every node, except for the root and the leaves has at least* $(2m-1)/3$ *sons; 3) the root has at least 2 and at most* $2\lfloor(2m-2)/3\rfloor+1$ *sons.*

C. Implementation Aspects of B-Tree and Their Variations

Each multiway tree node has 3 components `(degree, keys[], children[])`. We considered 2 alternative data layouts for a node of a B-Tree and all the variants of multiway trees studied. The first implementation uses the btree data structure presented in [8] and is presented in Figure 2(a). In this alternative they keys and children are stored as contiguous arrays. In the rest of this study this alternative will be referred to as `nodeContig`. The second alternative is presented in Figure 2(b) and stores the keys and the children of a node as an array of pairs `(key, child)`. In the rest of the study, this alternative will be referred to as `nodeAlt`.

Regardless of the layout chosen, the size of a multiway tree node is `2(degree+1)` bytes.

Machine	Line Size	L1 Size	L2 Size	Clock
Ultra-1	32-byte (L1) 64-byte (L2)	16KB	512KB	167MHz
Ultra-30	32-byte (L1) 64-byte (L2)	32KB	2MB	300MHz
Pentium II	32-byte (L1) 64-byte (L2)	16KB	512KB	400MHz

Fig. 3. Platforms used in the evaluation.

Name	Description	Parameters
splay	classical splay trees	none
splay-comp	splay trees with compaction	initCompact Threshold
btree	classical B-trees	nodeDegree, nodeLayout
star	B*-trees	nodeDegree, nodeLayout
mway	self-adjusting multi-way trees	nodeDegree, nodeLayout

Fig. 4. Evaluated algorithms and their parameters.

III. Experimental Setup

We ran our experiments on three platforms with different memory hierarchies – a Sun Ultra-1 and a Sun Ultra-30 running Solaris 2.6 and an Intel Pentium II running RedHat Linux 6.2. Figure 3 provides information about these platforms. In all experiments, our primary metric was end-to-end execution time. In addition, we also measured the total number of instructions (IC) and the average number of cycles needed to execute an instruction (CPI). CPI is a good measure of the end-to-end performance of the cache hierarchy. Large values of CPI indicate that the processor is spending a large fraction of its cycles waiting for data from memory. As a measure of the memory hierarchy performance we present data for the number of L2 level cache accesses (L2RW) and the L2 level cache hit ratio (L2 Hit).

We ran the experiments in 4 batches of 4 repetitions and selected the lowest time to minimize potential interference due to daemons and other processes running during the experiments. The variance in the end-to-end runtime reported is within 8% of the value reported. We coded all our algorithms in C and compiled them using `gcc` version 2.91 and optimization level O2. For end-to-end timing, we used the `gettimeofday()` system call, available on both Solaris and Linux. The resolution of this call is microseconds, and since our programs run for much longer, the results are precise enough. To collect information about the number of cycles, instructions, cache references and cache misses, we used the Performance Counter Library - PCL 1.3 [13].

We ran experiments for all combinations of the implementation parameters considered, presented in Figure 4, and a large variety of datasets. Table I presents the characteristics of a selected subset of datasets. We used the results to determine what are the best implementation parameter values. Corresponding to these parameter values, we selected the results on which we base our comparison.

Dataset	Lookups	Unique	σ/μ	$skew_{50}$
Uniform	4M	200K (5%)	N/A	49.9
Normal (lnorm1)	2M	100K (5%)	0.10	49.8
Normal (lnorm2)	2M	750K (37%)	1.00	49.9
Normal (lnorm3)	500K	360K (72%)	1.00	49.9
UCB Trace	25M	400K (1.6%)	N/A	17.3

TABLE I

Datasets used in this study.

A. Datasets

For this study, we used a set of synthetic datasets. The synthetic datasets were designed to elucidate the differences between the two techniques. Since the *keep-it-balanced* technique tries to improve the worst-case performance, it is likely to do well for request streams with little or no locality. The *move-to-front* technique tries to adapt to bias in the dataset (by moving frequently accessed keys closer to the root) and it can be expected to do well for skewed datasets.

To represent applications with little or no locality in the request stream, we use a dataset with uniformly distributed requests. To represent applications with varying degrees of locality in the request stream, we use normally distributed datasets with the same mean and different standard deviations (the σ/μ for these datasets varies between 0.01 and 0.72). Table I presents characteristics of these datasets. In addition, we use a highly skewed dataset, 90% of the search requests are made for 8% of the values, that consists of the first 25 million file requests from a month-long trace of file-system requests on a cluster of machines at UC Berkeley. Figures 5 and 6 and Table I present the characteristics of this dataset.

In order to better quantify the locality of the datasets we present and additional metric, referred to as $skewfactor_K(S)$. Let K be an interval length. Let the original sequence S of length N be split into intervals of length K. We define $skewfactor_K(S)$ as the average number of distinct keys appearing in the intervals of the original sequence, i.e. $\mathtt{skewfactor_K(S)} = \frac{\sum_{i=0}^{N/K} diff_i n_i}{\frac{N}{K}}$, where n_i is the number of intervals containing $diff_i$ keys, $diff_i \in \{1..K\}$. Table I presents results for $K = 50$. While the data distribution is an indication of the long range locality of the dataset, we chose this metric to reflect the short range locality (within subintervals of length K) of the data set.

Another parameter that influences the behavior of the algorithms studied is the lookup-insertion ratio of the dataset. For both data distributions we generated benchmarks with various lookup-

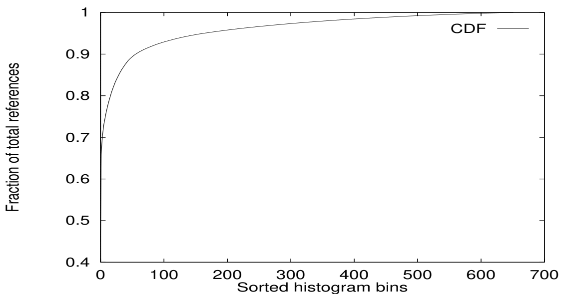


Fig. 5. Cumulative frequency distribution for UCB trace

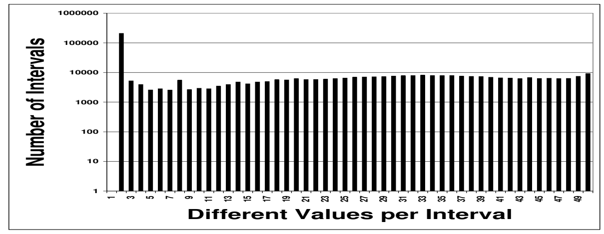


Fig. 6. Distribution of keys per interval of length 50 for UCB.

insertion ratios, `lookup dominated` (95%-5%), `balanced` (60%-40%), `insertion dominated` (30%-70%) . Additionally, for each data distribution and operation mix we generated benchmarks with the working set size comparable with the size of the L2 cache ($size_{L2}$), less than $size_{L2}$ and much bigger than $size_{L2}$.

The results presented in the rest of this study have been chosen such as to represent each class of datasets.

B. Tuning Algorithm Parameters

Figure 4 presents the set of algorithms used in this study and their parameters. We performed a initial sequence of experiments, using all the datasets presented in Table I, to determine suitable values for these parameters.

Move-To-Front: To determine a suitable value for `initCompactThreshold`, we ran experiments with values of `initCompactThreshold` between 15K and 100K insertions. On two platforms (Ultra1 and PentiumII) the best results were obtained for a value of 30K. On the Ultra-30 the best results were obtained for a value of 20K. Accordingly, we set this parameter to 30K.

Keep-It-Balanced: To determine a suitable value for `nodeDegree`, we analyzed the impact of `nodeDegree` on the number of cache lines that have to be loaded for a single lookup operation. Increasing the node degree reduces the depth of the tree which in turn reduces the maximum number of nodes that need to be traversed for a single lookup operation. On the other hand, increasing the node degree increases the number of cache lines needed to hold each node. Therefore, the maximum number of cache lines that have to be loaded for a single lookup can be expected to be:

$$numLines = \log_d(numKeys) \times (d \times keySize/lineSize)$$

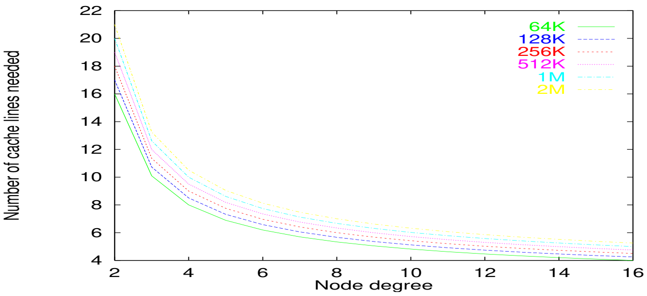


Fig. 7. Impact of node degree on the number of cache lines needed for each operation.The line labels (64K etc) indicate the number of keys in the tree.

Node Degree	IC	CPI	L2 RW	L2 Hit	Time(msec)
nodeDeg = 5	6.88B	1.11	96.9M	0.968	44750
nodeDeg = 10	6.81B	0.98	48.6M	0.958	38855
nodeDeg = 15	6.87B	0.91	27.4M	0.945	36687
nodeDeg = 20	7.49B	0.87	27.4M	0.948	37738
nodeDeg = 25	9.98B	0.814	28.2M	0.953	47272
nodeDeg = 50	12.7B	0.74	76M	0.964	54995

Fig. 8. Results for `UCB btree` for different node degrees (`nodeDeg`) on ULTRA1 for `nodeLayout` = `nodeContig`.

where d is the `nodeDegree`, $numKeys$ is the number of keys in the tree, $keySize$ is the number of bytes needed per key and $lineSize$ is the number of bytes in each cache line. Figure 7 shows how the number of cache lines needed for a single lookup operation varies as the node degree increases.

In particular, for trees with between 64K and 2 million keys and a L2 cache line size of 64 bytes, this value stabilizes around 4 for node degrees higher than 10. Because of this fact we concluded that increasing the degree above 15 is not likely improve the runtime behavior. To verify this, we ran experiments with values of `nodeDegree` of 5, 10, 15, 20, 25, 50. We repeated these experiments for all algorithms that use this parameter (i.e., `btree` and `star`).

We found that for `btree` and all datasets in the suite, a `nodeDegree` of 15 provided the best performance. This is because, with this setting, a btree node size is equal to two L2 cache lines (see Figure 3) and our custom allocator aligns nodes on L2 cache boundary. This layout minimizes the unnecessary conflict misses, at both L1 and L2 level, introduced when 2 nodes share the same cache line and has the best IC*CPI product value, as it can be seen from Figure 8. Accordingly, we set the value of `nodeDegree` to 15 for our evaluation of B-trees. For values of `nodeDegree` higher than 15, in the range $25 \ldots 50$, we found that the algorithm performs much slower.

For `star` we found that the best performance is obtained for a value of `nodeDegree` of 10 [2]. For this algorithm, internal tree nodes have a higher degree of occupancy and the split operation is more expensive than in the btree case. Because of the higher node occupancy, during a lookup, the B*-tree algorithm performs longer sequential searches than the B-tree algorithm would perform. Setting the node degree to a lower value than for btrees decreases the average length of a sequential search within internal nodes. On the other hand, a decrease in the node degree determines an increase in the number of split operations and in this case, split operations are more expensive

[2]Detailed results have been omitted for brevity.

Algorithm	IC	Cycles	L1 RW	L2 RW	L2 Hit	Time (msec)
nodeAlt	1.11B	1.66B	198M	56.2M	48.3M	9911
nodeContig	1.16B	1.54B	208M	40M	33.9M	9128

Fig. 9. Results for `lnorm2 btree` with alternate layouts on Ultra1.IC - instruction count. L1 RW - total L1 accesses. L2 RW - total L2 accesses. L2 Hit - total L2 hits.

Algorithm	L2 Hit	L1 RH	L1 WH	Time (msec)
btree	0.79	0.98	0.95	201.6
star	0.79	0.98	0.94	196.5

Fig. 10. Lookup only, 600K operations on a 1M keys tree on ULTRA 30.

than in the btree case. A value of 10 seems to provide the best balance between the length of the average sequential search and the number of split operations.

To determine a suitable value for the `nodeLayout` parameter, we ran experiments using both alternatives, `nodeContig` and `nodeAlt`. We used the value of `nodeDegree` previously determined. We found that for `btree nodeContig` outperformed `nodeAlt` in all cases by a factor within 10%. Figure 9 presents results for a run of `lnorm2` on the Ultra 1 machine. From Figure 9 it can be seen that `nodeAlt` has a lower instruction count but runs for a higher number of cycles. The lower instruction count is attributed to implementation differences. The higher number of cycles is attributed to the higher number of L2 accesses. For `star` we determined that `nodeAlt` outperforms `nodeContig` in all cases. We explain this behavior by the higher instruction count of `nodeContig` determined by implementation differences.

The results presented in this section are for implementations tuned for L2 level cache performance. We repeated all the experiments with implementations tuned for L1 cache performance. Since the working set size of out datasets intentionally exceeds the L1 cache size, the performance results for the implementations tuned specifically for L2 level cache were superior. All the results presented in the rest of this study are for the parameter values presented in this section.

IV. Results

Figure 11 presents the end-to-end execution time for all synthetic datasets and all algorithms. We make the following observations:

- The two B-tree-based algorithms (`btree`, `star`) usually outperform significantly the *move-to-front* algorithms (`splay`, `splay-comp`).
- The addition of cache optimizations to classical splay trees is likely to improve the overall performance.
- The qualitative behavior of the two classes of algorithms, i.e. *move-to-front* and *keep-it-*

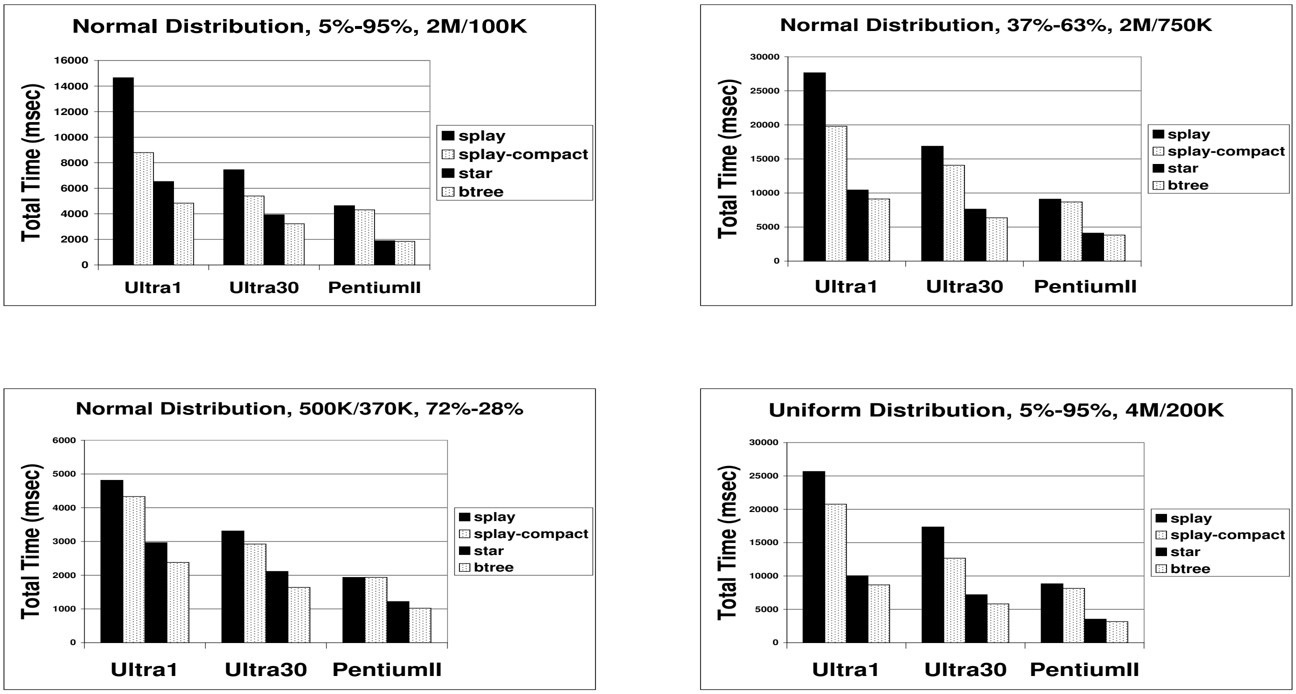
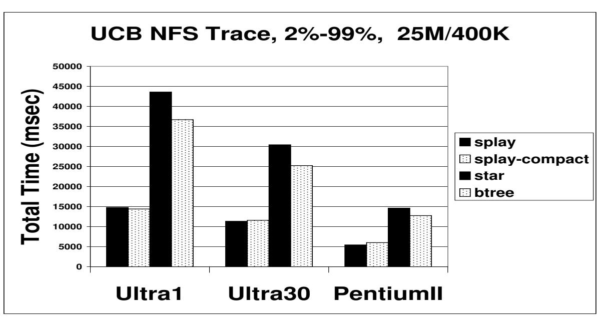


Fig. 11. End-to-end execution times for synthetic datasets. (normal distribution, uniform distribution)



Algorithm	IC	CPI	L2 RW	L2 Hit	Time (msec)	Norm Time
SUN ULTRA 1						
splay	1.95B	1.36	203.4M	0.994	14778	0.4
splay-comp	2.34B	1.15	282.8M	0.993	14403	0.39
mway	16.9B	0.74	673M	0.995	74352	2.02
star	8.4B	0.89	39.6M	0.959	43594	1.18
btree	6.87B	0.91	27M	0.959	36687	1
SUN ULTRA 30						
splay	1.94B	1.71	217M	0.993	11348	0.44
splay-comp	2.34B	1.34	285M	0.994	11583	0.45
mway	16.9B	0.76	670M	0.997	51332	2.03
star	8.04B	1.26	41.1M	0.953	30429	1.2
btree	6.87B	1.05	27.5M	0.941	25234	1
INTEL Pentium II						
splay	2.3B	1.13	9.2M	0.993	5450	0.42
splay-comp	2.56B	1.1	9.64M	0.991	6020	0.47
mway	15.3B	0.75	11.9M	0.957	25848	2.02
star	7.2B	0.88	17.7M	0.949	14630	1.14
btree	6.08B	0.82	12.8M	0.953	12752	1

Fig. 12. End-to-end execution time and cache behavior for UCB.

balanced, is not that much dependent on the overall benchmark operation mix, as it is on the short range locality of the benchmark.

Comparing `splay` and `splay-comp` we found that the latter usually outperformed the former (for all benchmarks except UCB). The speedups range, depending on the platform and the benchmark operations mix, from 1% to 41%. For all data distributions the speedup increases as the lookup-

Algorithm	IC	CPI	L2 RW	L2 Hit	Time (msec)	Norm Time
SUN ULTRA 1						
splay	789M	3.11	98.4M	0.9	14655	3.03
splay-comp	633M	2.33	100M	0.937	8792	1.81
mway	1.51B	1.04	93.4M	0.96	7837	1.62
star	730M	1.73	32M	0.883	6529	1.35
btree	739M	1.02	20M	0.939	4836	1
SUN ULTRA 30						
splay	596M	3.13	86.4M	0.984	7541	2.33
splay-comp	636M	2.12	103M	0.98	5394	1.67
mway	1.39B	0.98	71.3M	0.987	5399	1.67
star	706M	1.43	27.3M	0.93	3913	1.21
btree	738M	1.28	18.2M	0.978	3226	1
INTEL PentiumII						
splay	579M	3.28	34.3M	0.795	4639	2.51
splay-comp	626M	2.82	31.9M	0.811	4314	2.33
mway	1.3B	0.89	15.8M	0.9	2840	1.53
star	638M	1.25	19.1M	0.78	1897	1.02
btree	651M	1.17	12.3M	0.86	1848	1

Fig. 13. Normal Distribution, 5%-95%, 2M/100K

Algorithm	IC	CPI	L2 RW	L2 Hit	Time (msec)	Norm Time
SUN ULTRA 1						
splay	958M	4.9	139M	0.8	27655	3.02
splay-comp	893M	3.72	127M	0.826	19808	2.17
mway	2.37B	1.22	163.7M	0.896	17300	1.89
star	1.07B	1.65	48.5M	0.8126	10434	1.14
btree	1.1B	1.63	47.8M	0.83	9128	1
SUN ULTRA 30						
splay	834M	5.03	105M	0.829	16875	2.65
splay-comp	908M	3.85	133M	0.872	14071	2.2
mway	3.83B	1.08	182M	0.956	16208	2.54
star	1.07B	1.79	53.4M	0.872	7633	1.19
btree	1.16B	1.37	39.6M	0.883	6367	1
INTEL Pentium II						
splay	748M	4.81	60.8M	0.638	9104	2.38
splay-comp	850M	4.09	55.5M	0.621	8688	2.27
mway	3.38B	0.9	23.4M	0.632	7568	1.98
star	1B	1.63	32.3M	0.575	4114	1.07
btree	1.07B	1.44	22.8M	0.653	3818	1

Fig. 14. Normal Distribution, 37%-63%, 2M/750K

Algorithm	IC	CPI	L2 RW	L2 Hit	Time (msec)	Norm Time
SUN ULTRA 1						
splay	208M	3.99	31M	0.854	4814	2.02
splay-comp	236M	3.08	35.8M	0.871	4331	1.82
mway	568M	1.13	35.9M	0.938	3774	1.58
star	333M	1.36	16.8M	0.88	2964	1.13
btree	347M	1.1	12.4M	0.911	2377	1
SUN ULTRA 30						
splay	208M	3.96	30.9	0.9	3309	2.01
splay-comp	236M	3.11	36.2M	0.928	2924	1.78
mway	568M	1.23	34.8M	0.959	2669	1.62
star	333M	1.45	16.8M	0.9	2111	1.28
btree	348M	1.11	12.2M	0.942	1639	1
INTEL Pentium II						
splay	197M	3.92	12.7M	0.685	1934	1.89
splay-comp	222M	3.47	12.3M	0.685	1936	1.89
mway	522M	1.02	5.2M	0.692	1321	1.29
star	321M	1.47	7.78M	0.65	1219	1.19
btree	335M	1.22	5M	0.72	1021	1

Fig. 15. Normal Distribution, 72%-28%, 500K/370K

Algorithm	IC	CPI	L2 RW	L2 Hit	Time (msec)	Norm Time
SUN ULTRA 1						
splay	970M	4.5	127M	0.831	25673	2.96
splay-comp	1.02B	3.44	150M	0.873	20767	2.39
mway	2.36B	1.16	145M	0.928	15705	1.81
star	1.11B	1.52	46.6M	0.854	10037	1.15
btree	1.17B	1.37	48.7M	0.87	8666	1
SUN ULTRA 30						
splay	970M	4.43	126M	0.9	17345	2.98
splay-compac	1.02B	3.11	158M	0.936	12670	2.17
mway	2.36B	1.24	142M	0.957	11255	1.93
star	1.09B	1.66	46.3M	0.9	7190	1.23
btree	1.21B	1.21	33.5M	0.934	5812	1
INTEL Pentium II						
splay	923M	3.87	59.5M	0.689	8838	2.79
splay-compac	982M	3.39	57.2M	0.7	8147	2.57
mway	2.1B	1.01	30.2M	0.791	4909	1.55
star	992M	1.49	34M	0.697	3530	1.11
btree	1.06B	1.27	22.8M	0.771	3166	1

Fig. 16. Uniform Distribution, 5%-95%, 4M/200K

insertion ratio of the benchmark increases. We observed the highest speedups for the lookup dominated benchmarks (`lnorm1` and `uniform`). The speedup observed is also influenced by the platform cache size – it decreases as the cache size increases.

Comparing the behavior of the B-tree based implementations (`btree` and `star`) against each other we see the following trends. First of all, the regular B-tree implementation outperforms B*-trees. This is because of the better cache behavior of B-tree. In the `star` case, the packing modification of the algorithm determines accesses to nodes that were not in the search path. These extra accesses degrade the cache performance by a factor that cannot be compensated for by the lower instruction count of `star`. We expected B*-trees to outperform B-trees for *lookup* only benchmarks. To verify, we built a data structure containing one million random numbers and then we run a lookup only phase using 600K numbers randomly picked from the original sequence. The results for Ultra1 are presented in Figure 10.

From the set of benchmarks we analyzed, the UCB NFS trace is the only one where the splay tree based implementations outperformed the multiway tree based implementations. As it can be seen from Figure 12, for this benchmark, splay tree based implementations outperform all the other implementations by a factor of at least 2 on all platforms .

In the rest of this section we will analyze the performance of `btree` against the performance of `splay` for the UCB benchmark. While for all the other benchmarks the instruction count for `btree` is within a factor of 2 higher than the instruction count of `splay` ($IC_{btree} < 2IC_{splay}$), in this case $IC_{btree} \geq 3IC_{splay}$.

We can explain the qualitatively higher instruction count of the B-tree algorithm using the particular characteristics of this benchmark. The UCB trace consists of 25M operations with a lookup-insert ratio of 97%-3% and very high data locality. For this benchmark 90% of the lookup requests are made only for 8% of the data, which amounts to the fact that 22M operations request only 40000 different keys. The interval locality of this benchmark is also very high, 50% of the subintervals of length 50 containing only 2 keys. In the `splay` case, because of the move-to-front characteristic, the keys that are looked will be stored very close to the tree root. Furthermore, because of the small tree node size (12 bytes) and the small working set size, the nodes containing these keys fit completely within the L2 cache.

In the `btree` case the node size is larger. This will make the memory footprint larger. As it can be seen from Figure 12, this is reflected in the lower L2 hit ratio (0.959 vs 0.994). Also, for roughly 50% of the lookup operations, the `btree` algorithm will perform sequential searches and will find the key low in the tree, while in the `splay` case the key is found within the first three tree levels.

A. Predicting Runtime Behavior Based on the Dataset Skew

While the lookup-insertion ratio and the data distribution are both good indicators of the overall locality of a dataset, one cannot predict the qualitative performance difference between splay trees and B-trees by using only these two metrics. As shown in Section IV, with respect to the lookup-insert ratio of a dataset, we obtained contradictory results. For all lookup dominated synthetic benchmarks ($5\% - 95\%$), B-trees outperformed splay trees by a factor of two, while for the UCB benchmark ($3\% - 97\%$) they were slower than splay trees by a factor of two. With respect to the benchmark data distribution, B-trees outperformed splay trees for all synthetic benchmarks, regardless of the data locality.

In the following, we introduce the interval locality of a dataset as an alternate metric that can be used for the performance analysis of the two data structures considered. Using this metric we can predict more accurately the qualitative performance difference between splay trees and B-trees, regardless of the lookup-insert ratio or benchmark data distribution.

As the main performance indicator we use the average number of memory accesses made by `btree` and `splay` during a sequence of operations on the tree. We chose this metric because it is both an indication of the memory activity and the instruction count (IC). In the following, let B be a tree containing N keys. The dataset interval locality is described by the tuple (S, M, K) where S is a sequence of keys of length M, S = $(m_0, m_1 ... m_M)$, and K is the number of distinct keys within S, $(k_0 ... k_K)$. Let p_i be the probability that k_i is already in the tree. Let f_i be the number of occurrences of k_i within S ($\sum_{i=0}^{K} f_i = M$).

Considering the sequence of operations in S, the average cost per key C_{k_i} is the cost to insert the key in the tree, if the key is not already in the tree, and the cost of the subsequent lookup operations.

A.1 BTree

Let deg denote the tree node degree. Let l denote the number of nodes accessed for a lookup in the worst case. Knuth [10] shows that $l < 1 + log_{\lceil deg \rceil} \frac{N+1}{2}$.

We compute the cost to lookup a key in the tree as the expected cost for a sequential search within a node multiplied by the expected number of nodes accessed. We compute the cost for an insertion as the sum of the cost for a lookup and the cost of the actual insertion operation. The cost of the actual insertion is computed as the cost of the expected number of split operations.

As the expected number of nodes traversed in a search we chose the half of the theoretical worst case bound. As the expected number of sequential comparisons in a node we chose half the node size. As the expected number of split operations per insertion we chose half of the theoretical worst case bound.

Summarizing we obtain :

(1) $C_{k_i} = (1 - p_i)(exp\_search\_cost + exp\_split\_cost * exp\_splits) + p_i * exp\_search\_cost + (f_i - 1) * exp\_search\_cost$

(2) $exp\_search\_cost = exp\_nodes\_accessed * exp\_comparison\_cost$

(3) $exp\_nodes\_accessed = \frac{l}{2}$

(4) $exp\_comparison\_cost = deg$

(5) $exp\_split\_cost = 2deg$

The cost of handling the operations in the sequence S is $C_B = \sum_{i=0}^{K} C_{k_i}$. Substituting the above values we obtain

$$C_B = \frac{2*K*deg}{deg-1} + \frac{K*l*deg}{2} \ (6).$$

A.2 Splay

In this case we compute both the insertion cost and the lookup cost as half of the amortized cost of a lookup operation. Summarizing we have:

$C_{k_i} = (1 - p_i)exp\_insert\_cost + p_i * exp\_lookup\_cost + (f_i - 1)\frac{M}{K}$

(7) $exp\_insert\_cost = \frac{log(N)}{2}$

(8) $exp\_lookup\_cost = \frac{log(N)}{2}$

Substituting :

$$C_S = \sum_{i=0}^{K} C_{k_i} = \frac{Klog(N)}{2} + \frac{M^2}{K} - M \ (9)$$

A.3 BTree vs Splay

Given the dataset interval locality metric (S, M, K) and the expected costs C_B and C_S, for B-tree and splay tree respectively, we are interested in determining the number of distinct keys K per interval of length M for which $C_B \geq C_S$. Solving for $M = 50, N = 1M, deg = 15$ we obtain the solution $K = 10$.

This solution can be interpreted as follows. Given a dataset, we expect B-trees to outperform splay trees if any subinterval of length $M = 50$ contains more than $K = 10$ distinct keys. For $K < 10$ we expect splay trees to be faster. To validate our model we generated additional datasets. These datasets contain two million requests on 400K distinct keys, and the interval locality is varied between five and fifteen. Figure 17 presents the variation of the end-to-end running time of the two algorithms on the Ultra-30 and Intel PII machine. As it can be seen, B-trees start outperforming

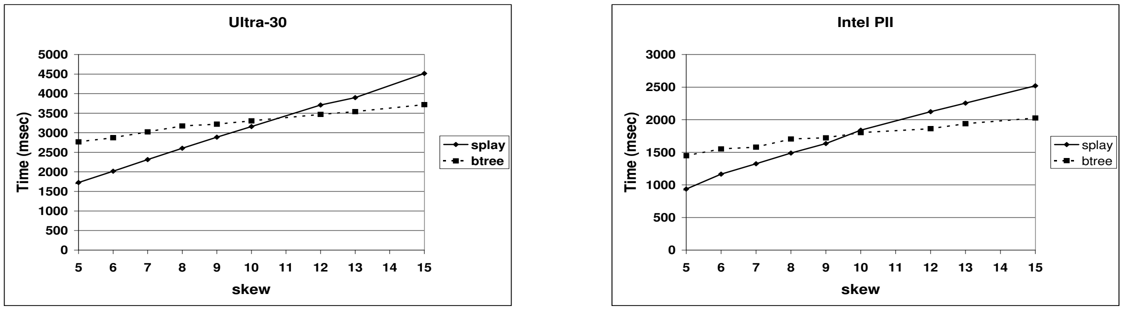


Fig. 17. End-to-end running time of splay trees and B-trees for datasets with different skew factors. The data sets contain two million requests on 400K distinct keys.

splay trees when the interval locality of the benchmark is "greater"[3] than ten ($K > 10$).

All benchmarks generated for the validation of our model contain a fixed value K of distinct keys per interval of length M. In Section III-A we introduced $skewfactor_M$ as the average number of distinct keys per interval of length M. In the following we analyze the qualitative performance difference based on values of $skewfactor_M$ and describe how this value can be used as an aid in deciding which algorithm to choose.

From the analysis above, we determined that for $K \leq 10$ different keys per interval of length 50, splay trees are faster than B-trees. The extent to which this happens is dependent of K and the hardware characteristics of the platform and we could not find a good way to quantify it.

Let w denote the percentage of the intervals of the dataset for which btree is slower than splay. Accordingly, $1 - w$ represents the percentage of the intervals for which btree is faster. We will look closer at the borderline case in order to determine a crossover value for w.

The borderline case happens when both classes of intervals contain a number of different keys close to the theoretical threshold determined (K = 10). For our analysis we chose these numbers as 5 and 11.

The total cost for B-tree is $\sum_{i=1}^{Nintervals} C_{B_i}$. This sum can be further expressed as $Nintervals * (w * C_{B_5} + (1 - w) * C_{B_{11}})$, where C_{B_5} and $C_{B_{11}}$ is the cost as determined in (6) with K = 5, K=11 respectively.

Doing a similar substitution for C_S and solving for `w` we obtain a crossover point, for our initial conditions, of 39% [4] What this means is that for datasets where more than 40% of the intervals

[3]Each interval of length fifty contains more than ten distinct keys.

[4]In the analysis we did not consider the variance of the total number of keys in the tree (N) as a factor that influences the behavior. Since we present only guidelines and the decision should be aided by the interval distribution histogram we feel that this assumption does not significantly change our recommendation.

contain less than 10 keys, splay trees are likely to outperform B-trees. For the other datasets, the interval histogram and the skew factor should be examined together.

For values of $skefactor_{50}$ less than 7.5 (determined by substituting the borderline case parameters $w = 0.40$, $K = 5, 11$) splay trees will outperform B-Trees. For values of $skewfactor_{50} \geq 25$ B-trees will outperform splay trees. For values within the range 7.5 - 25 the interval histogram needs to be examined.

V. Related Work

There has been a large amount of research on analyzing and improving the performance of search data structures both in terms of the operation complexity and the cache performance.

Bell&Gupta [7] conducted a evaluation of the self-adjusting binary tree techniques. They compared three flavors of the move-to-front technique, namely splaying, exchange and move-to-front with AVL and random binary search trees for different operation mixes and data distributions. They used synthetic benchmarks and reported results only for the operation count of the algorithms they evaluated. They did not take into account in their study the memory hierarchy and the results reported are only for benchmarks with a very small memory footprint (4K nodes).

Allen&Munro [4] conducted a study of the performance of self organizing search trees for various data distributions. They did not take the memory hierarchy into account and reported results only for datasets with small memory footprints.

Martel [12] presents a theoretical analysis of self adjusting multiway trees. He introduces a new data structure, called k-forest, and compares its theoretical complexity with B-Trees and splay trees.

LaMarca&Ladner [2] conduct a quantitative study on the performance of heaps against the performance of splay trees. They target directly the memory hierarchy influence on the algorithm performance and introduce collective analysis a a theoretical tool that can be used to predict performance. They also present results for cache optimized implementations of the algorithms they studied. As datasets they used only insertion dominated data sets with uniform distribution.

Acharya et al. [1] conduct a study of the cache performance of trie search. They develop adaptive algorithms based on the alphabet size. Their implementation changes the representation of the data structure from arrays to B-Trees to hash tables depending on the problem size. They study also the impact of different cache characteristics on their technique.

A related research direction is the study of the cache performance of numerical algorithms. Lam et al. [11] conducted a thorough study of the cache performance of blocked algorithms. They used a mixed approach, experimentation and theoretical modeling, as we did in this study. Based on their experimental results and their model, they make recommendations on how to chose implementation parameters for blocked algorithms for different cache parameters.

Our study is the only study that we are aware of that compares the influence of the memory hierarchy and the dataset distribution on the performance of both splay trees and multiway trees. Also, our study is the only study we are aware of that tries to predict the difference in performance between splay trees and B-trees depending on the data distribution.

VI. Conclusions

The results of this study are twofold. First, we conduct a performance study for two distinct algorithm design techniques for search trees – *move-to-front* and *keep-it-balanced*. We also present guidelines for estimating the qualitative performance difference of two search tree algorithms, i.e. splay tree and B-Tree, selected as representatives of the two design techniques considered. We determined that knowing the distribution of the input data set and the operation mix is not enough information to decide which of the two alternatives performs better. We introduced interval locality as a metric that can be used in the performance assessment of the two algorithms and presented guidelines on how this metric can be used.

Second, we present guidelines for choosing implementation parameters for best performance in the presence of a cache hierarchy for the two classes of search algorithms evaluated. For splay trees we present cache optimization techniques that worked well in practice and determined speedups ranging from 1% to 41%. For multiway trees we present guidelines for choosing the node degree in order to maximize runtime performance and memory hierarchy behavior.

Our recommendations for choosing the best implementation alternative for searching depending on the problem characteristics are :

- unless the input is prior known to have a very high degree of interval locality, B-trees are likely to outperform splay trees. For the B-tree implementation the node degree should be chosen in such a way that maximizes cache line usage (i.e. if possible one node occupies entirely one L2 cache line).
- when deciding between variants of B-tree, B*-tree will outperform B-tree for lookup only scenar-

ios. We do not think that the performance difference justifies the implementation complexity of the B*-tree.

- splay trees perform well for highly skewed datasets. For these datasets memory optimizations do not improve the end-to-end running time. If a splay tree implementation is already available, for most input data sets, memory optimizations are likely to improve the performance.

After conducting this study, we feel that while theoretical complexity analysis of algorithms is a powerful tool for predicting performance, in the presence of non uniform memory hierarchies, this alone is not enough for predicting the run time performance of the implementation. We think that in order to get a more accurate estimation of the expected performance, this analysis should be combined with a study of the memory hierarchy behavior of the algorithm and we urge researchers to combine the two when analyzing an algorithm behavior.

References

[1] K. S. A. Acharya, H. Zhu. Adaptive algorithms for cache-efficient trie search. 1999.

[2] R. L. A. LaMarca. The influence of caches on the performance of heaps. Technical report, University of Washington, 1997.

[3] G. Adel'son-Vel'skii and E. Landis. An algorithm for the organization of information. *Soviet Mathematics Doklady*, 3:1259–63, 1962.

[4] B. Allen and I. Munro. Self-organizing search trees. *Journal of the ACM*, 25(4):525–36, Oct 1978.

[5] R. Bayer. Symmetric binary B-trees: Data structure and maintenance algorithms. *Acta Informatica*, 1:290–306, 1972.

[6] R. Bayer and E. McCreight. Organization and maintenance of large ordered indexes. *Acta Informatica*, 1(3):173–89, 1972.

[7] J. Bell and G. Gupta. An evaluation of self-adjusting binary search tree techniques. *Software: Practice and Experience*, 23(4):369–82, Apr 1993.

[8] T. Cormen, C. Leiserson, and R. Rivest. *Introduction to Algorithms*. McGraw-Hill, 1989.

[9] L. Guibas and R. Sedgewick. A dichromatic framework for balanced trees. In *Proc. of FOCS'78*, pages 8–21, 1978.

[10] D. Knuth. *The Art of Computer Programming, Volume 3 (Sorting and Searching)*. Addison Wesley, 1998.

[11] M. W. M. Lam, E. Rothberg. The cache performance and optimizations of blocked algorithms. *ASPLOS-IV Proceedings*, pages 63–74, 1991.

[12] C. Martel. Self-organizing multi-way search trees. *Information Processing Letters*, 38(3):135–41, May 1991.

[13] H. Z. R. Berrendorf. Pcl 1.3 - performance counter library. Available at http://www.fz-juelich.de/zam/PCL/.

[14] M. Sherk. Self-adjusting k-ary search trees. *Journal of Algorithms*, 19(1):25–44, Jul 1995.

[15] D. Sleator and R. Tarjan. Self-adjusting binary search trees. *Journal of the ACM*, 32(3):652–86, Jul 1985.