Skewed Binary Search Trees

Gerth Stølting Brodal[1,*] and Gabriel Moruz[1]

BRICS**, Department of Computer Science, University of Aarhus, IT Parken, Åbogade 34, DK-8200 Århus N, Denmark. E-mail: {gerth,gabi}@daimi.au.dk

Abstract. It is well-known that to minimize the number of comparisons a binary search tree should be perfectly balanced. Previous work has shown that a dominating factor over the running time for a search is the number of cache faults performed, and that an appropriate memory layout of a binary search tree can reduce the number of cache faults by several hundred percent. Motivated by the fact that during a search branching to the left or right at a node does not necessarily have the same cost, e.g. because of branch prediction schemes, we in this paper study the class of skewed binary search trees. For all nodes in a skewed binary search tree the ratio between the size of the left subtree and the size of the tree is a fixed constant (a ratio of 1/2 gives perfect balanced trees). In this paper we present an experimental study of various memory layouts of static skewed binary search trees, where each element in the tree is accessed with a uniform probability. Our results show that for many of the memory layouts we consider skewed binary search trees can perform better than perfect balanced search trees. The improvements in the running time are on the order of 15%.

1 Introduction

In this paper we discuss the problem of building binary search trees that achieve good running times in practice for random queries. Theoretically, the minimum number of comparisons is achieved by perfectly balanced binary search trees, where for each given node the number of nodes in the left subtree is approximately equal to the number of nodes in the right subtree [15]. We show that in practice better running times can be achieved if we allow the search tree to be skewed, i.e. allow one of the subtrees to have more nodes than the other subtree.

When analyzing the complexity of an algorithm, usually the number of instructions performed by the CPU is counted. However, in practice there are other hardware issues besides the amount of computation that can affect the running time. During a search in a binary search tree, it is usually assumed that branching left and right at any given node inflicts the same cost on the running time. This does not always hold, since modern processors prefetch the instructions in a pipeline and therefore must predict the outcome of the conditional branches as

* Supported by the Danish Natural Science Foundation.
** Basic Research in Computer Science, www.brics.dk, funded by the Danish National Research Foundation.

their outcome is not known when they enter the instruction pipeline. If a branch is incorrectly predicted, the entire pipeline must be flushed, which results in a performance loss which is proportional with the pipeline size.

In the design of algorithms, in the RAM model it is assumed that all the memory accesses take constant time. Due to the memory hierarchy on modern computers, this hardly happens in practice. The access time for a given item can vary from one CPU cycle if it is stored in a CPU register to over 10,000,000 CPU cycles if the item must be fetched from the hard-disk. Due to the high costs of memory transfers between the different levels, data is not transfered in individual items, but in contiguous *blocks*. If the memory size and the block size are known, B-trees [5] support random searches in $O(\log_B N)$ block transfers, where B is the block size. If the memory parameters are not known, cache-oblivious B-trees [6, 7] achieve the same bound. Given a tree stored in memory, Gil and Itai [14] gave optimal algorithms for computing optimal layouts, while Alstrup et al. [2] introduced faster approximate algorithms for minimizing the expected number of memory transfers and Demaine et al. [12] proved worst case bounds. Brodal et al. [9] introduced an efficient version of cache-oblivious search trees and gave experimental results on the performance of some different memory layouts for search tress.

Recently, Sanders and Winkel [16] studied the influence that branch mispredictions have over the running time of algorithms in practice. They gave a distribution based sorting algorithm and show that in certain cases branch mispredictions can be avoided by using certain processor specific instructions, namely predicative instructions. Some other works focused on the influence of branch mispredictions over the running time of sorting algorithms, both theoretically and experimentally [10, 11].

Outline. The paper is structured as follows. In Section 2 we describe skewed balanced search trees and give an upper bound on the running time performed for a random query. In Section 3 we give brief insights on the hardware issues that affect the running time in practice. For a random query we give upper bounds on the number of branch mispredictions in Section 4, while in Section 5 we introduce different memory layouts and give upper bounds on the number of cache misses. In Section 6 we describe the setup for the experiments we perform and in Section 7 we show and discuss our experimental results.

2 Skewed binary search trees

A skewed binary search tree is a binary search tree where there exists a constant α, $0 < \alpha \leq 1/2$, such that for each node v there is a fixed ratio between the number of nodes in the subtree rooted in the left child and the subtrees rooted at v. More precisely, $\text{size}(\text{left}(v)) = \lfloor \alpha \cdot \text{size}(v) \rfloor$, where $\text{size}(v)$ denotes the number of nodes in the subtree rooted at v.

Skewed binary search trees are the extreme unbalanced cases of BB[α] trees of Nievergelt and Reingold [15].

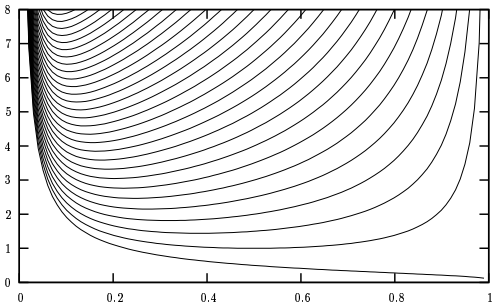

Fig. 1. Bound on the expected cost for a random search, where the cost for visiting the left child is $c_l = 1$ and the cost for processing the right child is $c_r = 0, 1, 2, \ldots, 28$ ($c_r = 0$ being the lowest curve).

Theorem 1 (Mehlhorn, Theorem 2, Section III.5.1). *The average path length P, is at most $(1 + 1/n)\log(n+1)/H(\alpha)$, where $H(\alpha) = -\alpha\log\alpha - (1-\alpha)\log(1-\alpha)$.*

In practice, due to hardware issues, the running time spent at a given node might depend on the next node to process, i.e. the left or right child. In Corollary 1 we analyze the running time for a random search in the case where the costs for visiting the left and right children of a given node are different.

Corollary 1. *Consider a skewed search tree T of balance α, and let c_l and c_r be the costs for branching left and right respectively. A random search has*

$$O((\alpha c_l + (1-\alpha)c_r)\log n/H(\alpha)) \tag{1}$$

expected cost, where $H(\alpha) = -\alpha\log\alpha - (1-\alpha)\log(1-\alpha)$.

Proof. Due to the linearity of expectation, the expected number of comparisons performed for a random search is equal to the average path length, which is $O(\log n/H(\alpha))$ cf. Theorem 1. If for branching left and right we have costs c_l and c_r, we obtain at a given node an expected cost of $\alpha c_l + (1-\alpha)c_r$, since the probabilities of branching left and right are α and $1-\alpha$ respectively. We conclude that the expected cost of a random search is $O((\alpha c_l + (1-\alpha)c_r)\log n/H(\alpha))$. □

In Figure 1 we show the function from the bound (1) on the expected cost for a random search where we consider different costs for visiting the left and the right child respectively. We note that in all the cases where $c_l \neq c_r$ the minimum occurs for α values different than $1/2$.

3 Hardware discussion

The running time of algorithms is usually analyzed by counting the instructions performed by the CPU. However, in practice, the running time of an algorithm

can be severely affected by some other hardware factors besides the CPU instructions. We show that the branch mispredictions that occur in the CPU and the cache faults can have a major effect over the running time of searching in skewed binary search trees.

To increase the clock speed, modern CPUs include instruction pipelines in their architecture, where the instructions are prefetched before being executed. When a conditional branch enters the pipeline, its outcome is not known prior to its execution and thus its direction must be predicted to ensure the prefetching of the following instructions. If the branch is incorrectly predicted, the whole pipeline must be flushed, since the instructions in the pipeline correspond to a wrong execution path. This obviously leads to a performance loss, which increases proportionally with the length of the pipeline. In such a case, we say that a *branch misprediction* occurs. Since the pipelines are getting longer and longer (e.g. 18 instructions for Pentium P4 and 31 for Intel Prescott), branch mispredictions are having an increasing influence over the running time of algorithms in practice.

In the traditional RAM model, all memory accesses are considered to have equal access times. In practice, nowadays computers have a hierarchy of memory layers, each of them having smaller size and access time than the next one, from the CPU registers to the hard-disk. The data can be transfered only between consecutive layers, and is performed in *blocks* of consecutive data rather than individual items.

4 Branch mispredictions

Branch mispredictions can dramatically affect the running time in practice. Even though in most of the cases the branch predictors incorporated in the CPU architectures are accurate and yield good performances, in certain algorithms the outcome of certain branches is hard to guess. Sorting and searching are two such examples, since they involve comparisons among elements and the outcome of an element comparison is usually hard to predict.

There are two major types of branch prediction schemes, namely static and dynamic. In static branch predictors, each branch is predicted in the same direction at all times, and the direction of the branch is either given at compile time or it follows some simple heuristics, e.g. forward branches predicted taken and backward branches predicted not taken. On the other hand, the dynamic branch prediction schemes predict the direction of the branches at runtime, taking advantage of the execution history. In the case of searching in a balanced search tree, since the number of nodes in the left and right subtrees of a given node are approximately the same, the outcome of any branch is hard to predict and hence we expect branch mispredictions in around half of the cases. On the other hand, for the skewed search trees, we expect the number of branch mispredictions to decrease when increasing the skewness, since the probability that the search key lies in the larger subtree is increasing. In Theorem 2 we prove an upper bound

on the number of branch mispredictions performed for a skewed binary search tree when a static branch predictor is used.

Theorem 2. *The expected number of branch mispredictions performed for a random search in a skewed binary search tree of balance α is $O(\alpha \log n / H(\alpha))$, where $H(\alpha) = -\alpha \log \alpha - (1-\alpha)\log(1-\alpha)$, assuming a static branch predictor and $0 \leq \alpha \leq 1/2$.*

Proof. Since we consider $\alpha \leq 1/2$, for each non-leaf node of the search tree, the right subtree will have more nodes than the left subtree, hence visiting the right subtree next is more likely than visiting the left subtree. We use a static prediction scheme where for each node we predict that the search key is larger than the key stored at the given node. Using Corollary 1 with $c_l = 1$ and $c_r = 0$, we obtain that for a random search we perform expected $O(\alpha \log n / H(\alpha))$ branch mispredictions. □

5 Memory layouts

The difference in access times between the different layers of the memory hierarchy, especially from the internal memory to the hard disk, has led to several models that deal with capturing the cache effect. One of the most successful is the I/O model introduced by Aggarwal and Vitter [1] and consist of a two level memory hierarchy, containing a fast memory of bounded size M and a slow, infinite memory. The computation is performed in the fast memory and the data is transfered between the slow and fast memories in blocks of B consecutive items. The I/O complexity of an algorithm is given by the number of blocks transfered. Since in practice hardware architectures contain several memory levels with different values for the fast memory size M and the block size B, Frigo et al. [13] introduced the cache oblivious model. A cache oblivious algorithm is an algorithm whose analysis holds for any values of M and B. Most of the algorithms in this model assume a tall cache, i.e. $M = \Omega(B^2)$. For a comprehensive list of efficient external memory algorithms, e.g. refer to [3, 4, 8, 17].

We analyze different memory layouts for the static skewed binary search trees. For all the layouts the tree is stored as an array of n nodes, where each node is a structure containing two pointers to the left and the right subtree respectively together with an integer key. We note that the number of comparisons and branch mispredictions performed for searches is not affected by the way the tree is laid in memory, as they only depend on the height of the tree and the number of left turns on a path from the root to a certain leaf (for $\alpha < 1/2$, assuming a static branch prediction scheme). However, the number of cache faults can be dramatically affected by the memory layout, ranging from $O(1/\log B)$ to $O(1)$ I/Os for each node on a search path.

Consider a balanced binary search tree T of n nodes. The different memory layouts that we consider together with the expected number of I/Os for a random search are introduced below.

Random. Each node of T is stored at a random position in the array.

Since in this layout the nodes are stored at random locations in the array, for each node on a search path we perform an I/O, hence the expected number of I/Os is given by the average path length.

BFS. In this layout the nodes of the tree are stored according to the BFS traversal of T, where the nodes at a level are processed in a left-to-right order.

The first B nodes of the array contain the topmost subtree. In any practical setting, i.e. the tree is not severely skewed, the length of any path in this subtree is $\Theta(\log B)$. The top subtree is loaded into memory using in a single I/O, hence for the first $O(\log B)$ nodes on any path we use $O(1)$ I/Os. Afterward, for the remaining nodes on any search path we consume $O(1)$ I/Os per node, thus obtaining expected $O(1 + |P| - \log B)$ I/Os for a following a search path P.

Inorder. The tree is stored in the array according to the inorder traversal, i.e. the array is sorted.

Following a path from the root to a leaf takes $O(1)$ I/Os per node, except for possibly the last subtree of $\Theta(B)$ nodes, since they will be loaded using a single I/O. Considering the case when in a subtree of size B the length of a search path is $O(\log B)$, we obtain that for a search path P in this layout we perform between $O(|P|)$ and $O(1+|P|-\log B)$ I/Os, where $|P|$ denotes the length of P, depending whether P reaches the bottom levels of the tree or not.

DFSl. The tree is laid out in the array according to a DFS traversal, where after visiting the root, the left child is traversed before the right child.

Since the left child is stored next to the parent, they are stored in the same block, hence branching left takes $O(1/B)$ I/Os. In what concerns the right child, accessing it requires $O(1)$ I/Os. Using Corollary 1 we obtain that for a random search we perform expected $O((\alpha/B+(1-\alpha))\log n/H(\alpha))$ I/Os, where $H(\alpha) = -\alpha\log\alpha-(1-\alpha)\log(1-\alpha)$.

DFSr. This layout is similar to DFSl, except for the fact that the right child is traversed first and the left child afterwards. Using a similar argument, we obtain that the number of I/Os performed for a random search is expected $O(\alpha + (1-\alpha)/B)\log n/H(\alpha)$.

k-level grouping. Given a tree T, in this layout we first store the first k levels of T in the order given by a BFS traversal and then recursively store the subtrees rooted in the nodes at level $k+1$, in a right-to-left order.

Choosing $k = \log B$, we obtain that following a search path P takes $P(1 + |P|/\log B)$ I/Os, each block is loaded using $O(1)$ I/Os and in each block we process $\Theta(\log B)$ nodes of the search path, except for possibly the last block loaded. Since the expected length of P is $O(\log n/H(\alpha))$, we obtain that the expected number of memory transfers is $O(1 + (1/H(\alpha)) \cdot \log_B n)$.

pqDFS. In a preprocessing phase, for each node v we assign its weight $w(v)$ as the number of nodes contained by the subtree rooted at v. Given a parameter p, we first store consecutively the p heaviest nodes in decreasing order with respect to their weights. The subtrees rooted at the children of the nodes on the frontier, if any, are then recursively stored. The children are laid out in decreasing order of their weights. If two or more nodes have the same weight, no assumption can be made with respect to the order in which they will be stored. To implement this layout we use a priority queue, hence its name.

To optimize the number of memory transfers, we choose $p = \Theta(B)$ and thus the group of the p heaviest nodes is stored in $O(1)$ memory blocks. For the children of the frontier of a group of p nodes the ratio between the weight of the lightest and heaviest child0 is at least α (for $0 \leq \alpha \leq 1/2$). This implies that each subtree in the frontier of the group has at most a fraction of $1/(B\alpha + 1)$ of the size of the subtree rooted at the root of the group. It follows that a search uses $O(\log_{B\alpha+1} n)$ I/Os.

Skewed van Emde Boas. This layout is a variation of the van Emde Boas layout, which is known to match in the cache-oblivious model, i.e. where the parameters M and B are not known, the best bounds known for searching in the I/O-model. Given a node v and a tree, the weight of the node is given by the number of nodes in the subtree rooted in v. Given a tree of n nodes, we split it into a top subtree containing $\lceil\sqrt{n}\rceil$ nodes and $O(\sqrt{n})$ bottom subtrees. The top subtree contains the nodes with the highest weights and the bottom subtrees have as roots the children of the leaves of the top subtree. After the splitting phase, the top and the botttom subtrees are recursively stored in consecutive memory locations.

Since the top subtree contains the heaviest $\lceil\sqrt{n}\rceil$) nodes, by a similar argument to pqDFS the ratio between the weights of the lightest and heaviest root of the bottom subtrees is at least α (for $0 \leq \alpha \leq 1/2$). If the root of the tree has weight n, we obtain that the number of nodes in each of the bottom subtrees is at most $n/(\alpha\sqrt{n} + 1)$ nodes. In the recursive layout, when $n = \Theta(B)$ searching in the corresponding subtree takes $O(1)$ I/Os. We obtain that a search takes $O(\log_{B\alpha+1} n)$ I/Os.

6 Experimental setup

We analyze how the skewness factor α of the binary tree affects the running time in practice for the different layouts. To avoid additional costs inflicted over the running time by recursive calls, we use the iterative searching procedure in Figure 2. We generate a large sequence of random successful queries and measure the running time together with the number of comparisons, the number of branch mispredictions and the L1 data cache misses performed. We conduct our experiments on two standard Linux machines, having two different architectures. One of them has a P4 3.4 GHz CPU and 1 GB RAM, running linux 2.6.10. The other one has an AMD Athlon XP 2400+ 2.0 GHz CPU with 1GB RAM, running linux 2.6.8.1. To count the number of branch mispredictions and

```
while(root!=NULLV)
{
if(key==t[root].key)
   return root;
if(key>t[root].key)
   root=t[root].right;
else
   root=t[root].left;
}
```

Fig. 2. An iterative C source code for searching.

L1 data cache misses we use the PAPI 3.0 library. The code is compiled with gcc 3.3.2 using optimization level -O3. We will restrict ourselves to showing in the paper empirical results for AMD architecture. For the Pentium 4 processor the same behavior was observed as for the AMD architecture. The source code together with the scripts running the experiments and the plotted resulting data are available at `www.daimi.au.dk/~gabi/esa06.tar.gz`.

7 Experimental results

We demonstrate experimentally that in practice the skewed binary search trees can outperform the theoretically better balanced binary search trees, because of the different costs for branching left or right.

Since the number of branch mispredictions and the amount of computation (i.e. the number of comparisons) are independent on the memory layout, we can count them on any layout. The charts in Figure 3 are obtained by counting the number of comparisons (left) and the number of branch mispredictions (right) for a tree of 25×10^3 items and 10^6 queries. As expected, the number of comparisons achieves a minimum for perfectly balanced trees, i.e. for $\alpha \approx 0.5$, and increases with the skewness of the tree. In what concerns branch mispredictions, their number increases by a factor of 350% when decreasing the skewness, following the expectation in Theorem 2. Intuitively, this happens because the more nodes one of the subtrees rooted at the children of a given node has, the more likely is that a random search path will contain that child, hence the more likely the searching conditional branch will be correctly predicted. We observe that the number of branch mispredictions has a maximum for $\alpha \approx 0.52$ and that for very high values of α the number of branch mispredictions is greater by about 25% than for very low values. This is because of the rounding for small instances, i.e. the number in the left subtree is $\lfloor \alpha n \rfloor$ which yields a rightmost path for $\alpha n < 1$.

As previously stated, the number of cache faults performed for a random search depends not only on the skewness factor α, but also on the memory layout of the tree. We first analyze the layouts that do not use blocking, that is DFSl, DFSr, BFS, Inord and Rand. In Figure 4 we give the running time (left) and the number of cache misses (right) performed by 10^6 queries in a skewed

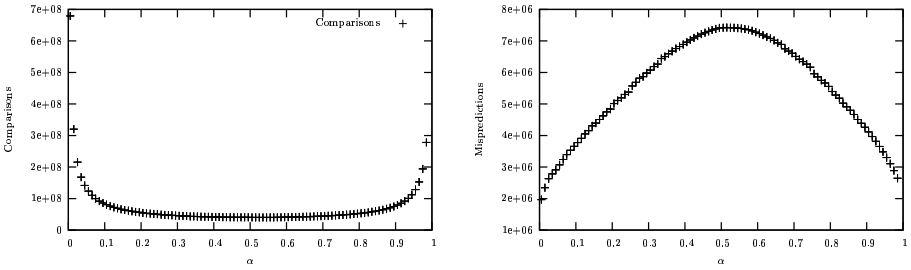

Fig. 3. The number of comparisons (left) and branch mispredictions (right) performed by a skewed search tree of 25×10^3 items for 10^6 queries.

search tree of 25×10^3 nodes. As expected, the Rand layout achieves the worst running time, since it performs one cache fault for each element on a given path. Inord and BFS achieve competitive running times, whereas DFSl and DFSr are best layouts that do not use blocking, with respect to both running time and cache misses performed. We note that the Inord layout performs less cache faults and achieves better running times than BFS for very skewed trees, i.e. very small or very large values of α, whereas when the trees are almost balanced BFS outperforms Inord. Also, it is expected that DFSl and DFSr have symmetric charts for the number of cache misses and implicitly the running time, since they are symmetric layouts, where DFSr is efficient for $\alpha < 0.5$, since there are more nodes in the right subtree, and DFSl is more efficient for $\alpha > 0.5$. We recall that in the case of DFSr, since the right child is recursively stored after the root, branching right takes $O(1/B)$ I/Os whereas we spend $O(1)$ I/Os for branching left, whereas in the case of DFSl we spend $O(1/B)$ I/Os for branching left and $O(1)$ I/Os for branching right. We note that the minimum running time is achieved for $\alpha \approx 0.2$ in the case of DFSr and for $\alpha \approx 0.75$, and is better by around 15% compared to $\alpha = 0.5$. In DFSr, for $0.2 < \alpha \leq 0.5$, even though less comparisons are performed, both cache faults and branch mispredictions increase and the overall running time increases too.

We now analyze the blocked layouts. We conduct experiments for tuning the parameterized layouts, i.e. k-level grouping and pqDFS. Again, we perform 10^6 queries on a skewed search tree of 25×10^3 nodes, for different values of the parameters. For k-level grouping, we give experimental data for different values of the parameter k, i.e. the number of levels grouped together in the layout, for different values of α. For each pair of values for k and α, we perform three series of queries and select the median of the running times. For each value of the parameter k we choose the smallest running time among the different possible skewness factors. The data we obtained is shown in Figure 5 (left). The differences in the running times are up to 5%, and the minimum running time is achieved for $k = 2$, i.e. when two levels of the tree are grouped together. Thus, in our further experiments involving this layout we use this value.

We perform the same experiments for the pqDFS layout, varying the number p of the heaviest nodes grouped in a block, see Figure 5 (right). Unlike the k-level grouping, in this case the differences in the running times are very small. Since the minimum running time was obtained when grouping $p \approx 40$ nodes together, in the further experiments we are using $p = 40$.

We perform a comparative study for the blocked layouts, i.e. k-level grouping, pqDFS, and skewed van Emde Boas, together with DFSr, since it is the non-blocked layout that achieved the best running time. In Figure 6 we show the running times (left) and the number of cache misses (right) performed for these layouts on a skewed binary search tree of 25×10^3 nodes for 10^6 queries. We note that even though all layouts achieve approximately the same running times, at all times the skewed van Emde Boas is the fastest. The heuristics of grouping the heavy nodes achieves good results in practice, since pqDFS is faster than blocking k levels (bDFS). Finally, we note that DFSr is slightly slower than the blocked layouts. In what concerns the data cache misses, for all the algorithms the number of data cache misses is almost similar and is approximately the same regardless of the skewness factor for $\alpha < 0.5$, except for the case when the tree is extremely skewed, i.e. for very small values of α. We note when increasing the skewness factor α up to 0.5, the number of comparisons decreases, the number of cache misses is approximately the same except for extremely low values of α, whereas the number of branch mispredictions is increasing. The resulting effect is that the minimum running time is achieved for $\alpha \approx 0.3$, and is better by a factor of 5% compared to the perfectly balanced search trees for all the blocked layouts. As stated before, for DFSr, the observed improvement in the running time is up to 15%. In what concerns the number of caches, the blocked layouts performed much better than the non-blocked layouts, as the skewed van Emde Boas and pqDFS layouts achieve significant improvements against BFS, Inord and Rand.

Finally, we study for which values of the skewness factor α we achieve the minimum running time when varying the size of the tree. We choose to perform our experiments on two of the layouts that achieved the best running times, namely pqDFS and the skewed van Emde Boas. For a given tree size, we vary the skewness factor α and for each value of α we perform three series of 10^6 queries and pick the median of the running times. We then measure the skewness factor for which the minimum running time was achieved. In Figure 7, we show the resulting data for both the AMD (left) and P4 (right) architectures. We notice that for both architectures the pqDFS achieves its best running time for smaller values of α than skewed van Emde Boas. Also, the best skewness factor is increasing while increasing the input size in the case of the AMD architecture, whereas for the P4 it has a constant behavior when increasing the input size.

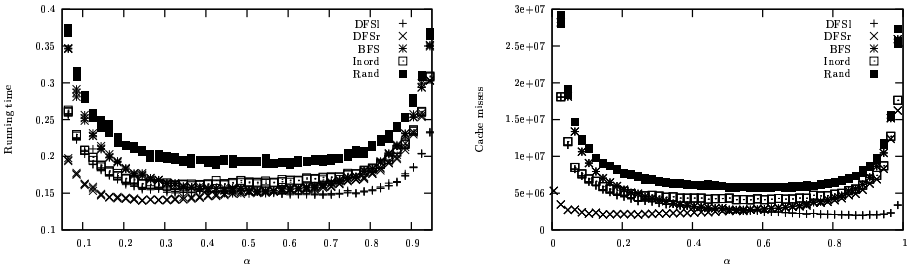

Fig. 4. The running time (left) and the number of L1 data cache misses (right) performed by a skewed search tree of 25×10^3 items for 10^6 queries for the non-blocked layouts.

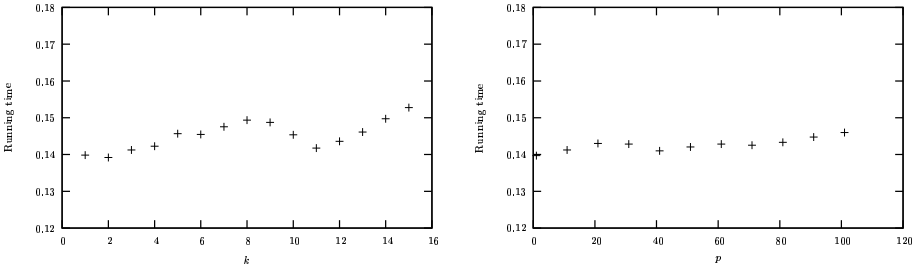

Fig. 5. The best running times for k-level grouping (left) and pqDFS where p nodes are grouped together (right), for 10^6 queries and a skewed search tree of 25×10^3 nodes.

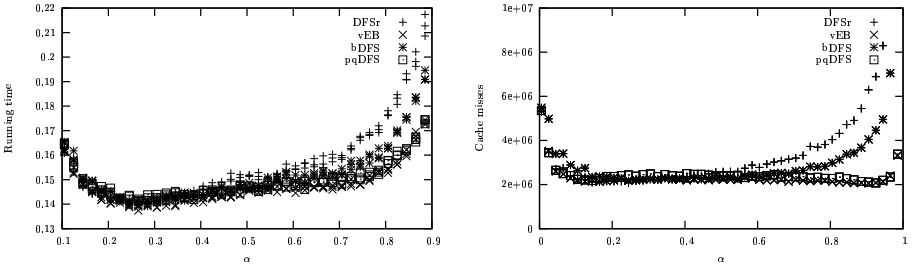

Fig. 6. The running time (left) and the number of L1 data cache misses (right) performed by a skewed search tree of 25×10^3 nodes for 10^6 queries for the blocked layouts.

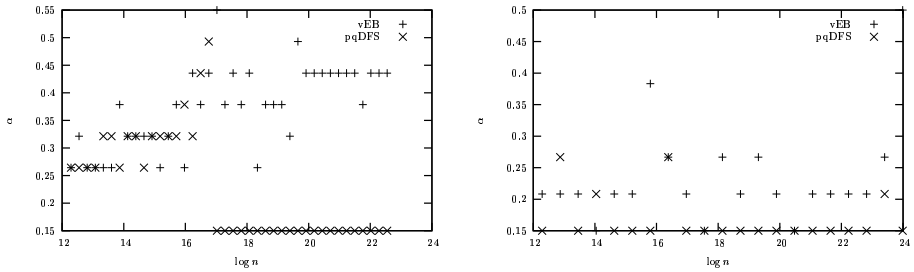

Fig. 7. The skewness factors that achieved the minimum running times for different tree sizes for the Athlon (left) and P4 (right) architectures.

References

1. A. Aggarwal and J. S. Vitter. The input/output complexity of sorting and related problems. *Communications of the ACM*, 31(9):1116–1127, 1988.
2. S. Alstrup, M. A. Bender, E. D. Demaine, M. Farach-Colton, J. I. Munro, T. Rauhe, and M. Thorup. Efficient tree layout in a multilevel memory hierarchy. Manuscript, 2003.
3. L. Arge. External memory data structures. In J. Abello, P. M. Pardalos, and M. G. C. Resende, editors, *Handbook of Massive Data Sets*, pages 313–358. Kluwer Academic Publishers, 2002.
4. L. Arge, G. S. Brodal, and R. Fagerberg. Cache-oblivious data structures. In D. Mehta and S. Sahni, editors, *Handbook of Data Structures and Applications*, page 27. CRC Press, 2004.
5. R. Bayer and E. M. McCreight. Organization and maintenance of large ordered indices. *Acta Informatica*, 1:173–189, 1972.
6. M. A. Bender, E. D. Demaine, and M. Farach-Colton. Cache-oblivious B-trees. In *Proc. 41st Annual Symposium on Foundations of Computer Science*, pages 399–409, 2000.
7. M. A. Bender, Z. Duan, J. Iacono, and J. Wu. A locality-preserving cache-oblivious dynamic dictionary. In *Proc. 13th Annual ACM-SIAM symposium on Discrete algorithms*, pages 29–38, 2002.
8. G. S. Brodal. Cache-oblivious algorithms and data structures. In *Proc. 9th Scandinavian Workshop on Algorithm Theory*, volume 3111 of *Lecture Notes in Computer Science*, pages 3–13. Springer Verlag, Berlin, 2004.
9. G. S. Brodal, R. Fagerberg, and R. Jacob. Cache-oblivious search trees via binary trees of small height. In *Proc. 13th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 39–48, 2002.
10. G. S. Brodal, R. Fagerberg, and G. Moruz. On the adaptiveness of quicksort. In *Proc. 7th Workshop on Algorithm Engineering and Experiments*, pages 130–140, 2005.
11. G. S. Brodal and G. Moruz. Tradeoffs between branch mispredictions and comparisons for sorting algorithms. In *Proc. 9th International Workshop on Algorithms and Data Structures*, volume 3608 of *Lecture Notes in Computer Science*, pages 385–395. Springer Verlag, Berlin, 2005.
12. E. D. Demaine, J. Iacono, and S. Langerman. Worst-case optimal tree layout in a memory hierarchy. Manuscript, August 2004.
13. M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. Cache oblivious algorithms. In *40th Annual IEEE Symposium on Foundations of Computer Science*, pages 285–298, 1999.
14. J. Gil and A. Itai. How to pack trees. *Journal of Algorithms*, 32(2):108–132, 1999.
15. J. Nievergelt and E. M. Reingold. Binary search trees of bounded balance. In *Proc. 4th Annual ACM Symposium on Theory of Computing*, pages 137–142, 1972.
16. P. Sanders and S. Winkel. Super scalar sample sort. In *Proc. 12th European Symposium on Algorithms*, volume 3221 of *Lecture Notes in Computer Science*, pages 784–796. Springer Verlag, Berlin, 2004.
17. J. S. Vitter. External memory algorithms and data structures: Dealing with massive data. *ACM Computing Surveys*, 33(2):209–271, 2001.