The Dense Skip Tree: A Cache-Conscious Randomized Data Structure

Michael Spiegel and Paul F. Reynolds, Jr.

March 4, 2009

Department of Computer Science
Technical Report CS-2009-05
School of Engineering and Applied Science
University of Virginia
151 Engineer's Way, P.O. Box 400740
Charlottesville, Virginia 22904-4740

Abstract

We introduce the dense skip tree, a novel cache-conscious randomized data structure. Algorithms for search, insertion, and deletion are presented, and they are shown to have expected cost $O(\log n)$. The dense skip tree obeys the same asymptotic properties as the skip list and the skip tree. A series of properties on the dense skip tree is proven, in order to show the probabilistic organization of data in a cache-conscious design. Performance benchmarks show the dense skip tree to outperform the skip list and the self-balancing binary search tree when the working set cannot be contained in cache.

1 Introduction

The persistent disparity between processor and memory speeds, the so-called "memory wall" [1, 2], is a pervasive component of modern computing. Over the past two decades, while processor performance has doubled every 18 months, memory latency has improved only by about 7% a year [3]. The lag of latency versus bandwidth is observed across all layers of information processing, from the microprocessor, to the memory, the hard drive, and the network interconnect. In the time that bandwidth doubles, latency improves by no more than a factor of 1.2 to 1.4 [4]. Numerous techniques have been developed for the cache-conscious access of data within the framework of the multilevel memory hierarchy. These techniques include loop transformation [5, 6], data clustering [7], data coloring [8], and heap reorganization through garbage collection [9], or dynamic profiling [10]. These techniques are deterministic in nature. This paper introduces the dense skip tree, a variant of the skip list, as a novel cache-conscious randomized data structure that is designed to probabilistically achieve a cache-conscious access of data. The randomization technique is a novel addition to the set of deterministic techniques that are used to reduce latency effects associated with the memory wall.

The skip list [11] is a randomized data structure that is balanced probabilistically in order to yield expected $O(\log n)$ complexity costs for common operations. The local balancing of the skip list

has made it an attractive data structure in a variety of applications such as peer-to-peer distributed search [12, 13], multiple predicate matching in database systems [14], high-dimensional approximate nearest neighbor search [15], and concurrent lock-free information retrieval [16–18]. The skip list provides fast algorithms for insertion and deletion, but its search performance is restricted as only one key is stored per data structure node. The dense skip tree allows multiple keys to be stored per node, yet with no global constraints on the distribution of nodes. Furthermore, the number of keys stored per node is shown to be a derived property of the dense skip tree. It will be shown that the dense skip tree is a compelling alternative to the skip list and the skip tree as a cache-conscious randomized data structure.

2 Dense Skip Tree Algorithms

We define the dense skip tree data structure and provide algorithms for search, insert, and delete operations. The search operation is identical to any other search on a k-ary tree, such as searching a binary tree or a B-tree. The insert operation will be described in detail in this section. The delete operation will not be discussed. However, the algorithm for deletion can be deduced from the insertion algorithm.

Definition 1. A *skip tree* [19] is defined to be a multiway search tree, such that the following properties hold for all nodes:

(D1) Each node contains k keys in sorted order, k values, and possibly-null references to $k+1$ children for $k \geq 0$.

(D2) Each node contains a height h for $h \geq 0$.

(D3) Each key is assigned a random height at insertion.

(D4) A node with height h has children with height $h-1$. Nodes at zero height do not have children.

(D5) The left subtree of any key contains only keys of lesser value.

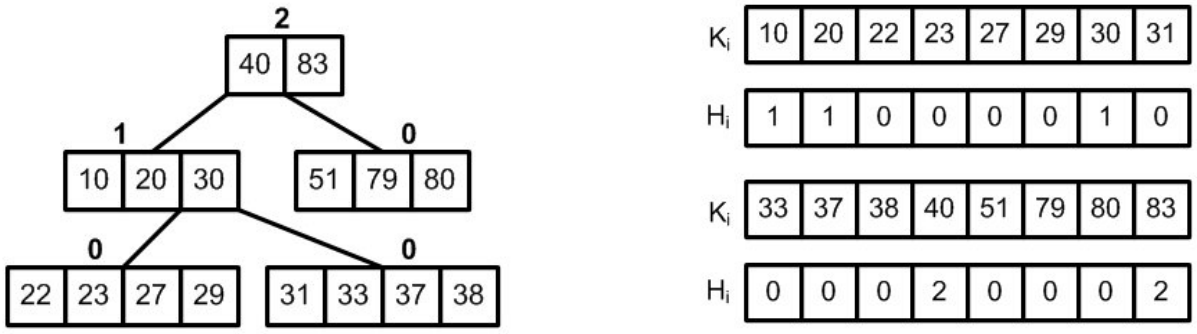
(D6) The right subtree of any key contains only keys of greater value.

A leaf node is defined as a node with height 0. The root node is defined as the node with the maximum height. Notice that in the skip tree, all paths from the root node to a leaf node have the same length. A skip tree node with 0 keys is defined as a white node. The skip tree consists predominantly of white nodes, in order to maintain the path length invariant. The dense skip tree is defined as a compact variation on the skip tree data structure. The dense skip tree definition shares properties (D2), (D3), (D5), and (D6) from the original skip tree. (D4) is relaxed in the dense skip tree to permit any monotonic decreasing relationship between parent and children heights. With this relaxation, the existence of nodes with zero keys is no longer necessary.

Definition 2. A *dense skip tree* is defined to be a variant of the skip tree, such that the following properties are different:

(D1') Each node contains k keys in sorted order, k values, and possibly-null references to $k+1$ children for $k \geq 1$.

(D4') A node with height h has children with heights that are less than h.



(a) (b)

Figure 1: (a) A dense skip tree with (b) corresponding K_i and H_i vectors

An example of a dense skip tree is shown in Figure 1(a). Nodes are shown with their keys but without their values. The number above each node indicates the height of that node. Insertion on a dense skip tree is defined as a recursive algorithm that traverses from the root of the tree to the leaves. The accompanying pseudocode is shown in Algorithm blocks 1-4. The insert operation first checks the current node to determine if the target key is present (Algorithm 1).

Algorithm 1 Dense skip tree insertion, preamble

```
1: procedure INSERT(node : Node, key : Key, value : Value, height : ℕ)
2:     Search for key in current node
3:     if (key is found) then
4:         update the value in node and exit
5:     end if
```

There are three possible cases to consider for the insertion of a key. The height of the new key is either less than, equal to, or greater than the height of the current node. If the height of the new key is less than the height of the current node, then the new key must be inserted into the appropriate child of the current node (Algorithm 2).

Algorithm 2 Dense skip tree insertion, case one

```
6:     if (height < node.height) then
7:         find the insertion point of the new key in the current node
8:         if (∃ a child at the insertion point) then
9:             recursively insert (key, value, height) in child node
10:        else
11:            create singleton child (key, value, height)
12:            add child to the current node at the insertion point
13:        end if
14:        return the current node
```

A *singleton node* is a node that contains exactly one key, one value, and two **NIL** children. The *insertion point* of a key is the index at which a key would be inserted in order to maintain the ordering of the keys. The insertion point of an array of size n is a value in the interval $[0, n]$, which may lie outside the bounds of the key array but always lies inside the bounds of the children array.

The insertion of a new key at the insertion point preserves properties (D5) and (D6).

In the second case of the insert operation, the height of a new key is equal to the height of the current node (Algorithm 3). At this point the *split operation* is defined. The split operation takes a target node, a target key, and a new value as input parameters. It returns either one node or a pair of nodes as the result. If the target node or any of its descendant nodes contains the target key, then the value associated with the target key is updated and the target node is returned. Otherwise the node is split into two trees, a left tree that contains keys less than the target key, and a right tree that contains keys greater than the target key.

Algorithm 3 Dense skip tree insertion, case two

```
15:     else if (height = node.height) then
16:         find the insertion point of key in node
17:         split the child at the insertion point
18:         if (split returns two nodes) then
19:             add the key to this node at the insertion point
20:             add the two children to this node at the insertion point
21:         end if
22:         return the current node
```

In the last case, the height of a new key is greater than the height of the current node (Algorithm 4). The current node is split into a left subtree and a right subtree. If the split operation returns a single result, then update the existing key with a new value. If the split operation returns two nodes, then create a singleton node with the $(key, value)$ pair and insert the two subtrees as the children of this singleton node.

Algorithm 4 Dense skip tree insertion, case three

```
23:     else ▷ (height > node.height)
24:         split the current node
25:         if (split returns two nodes) then
26:             create a singleton node
27:             add (key, value, height) to the singleton node
28:             add the two children to the singleton node
29:             return the singleton node
30:         end if
31:     end if
32: end procedure
```

The split operation of a dense skip tree is similar to the split operation of a B-tree or the split operation of a regular skip tree. The split algorithm proceeds down a path of skip tree nodes and partitions the path according to the new key. The new paths must be created in order to preserve properties (D5) and (D6) of the skip tree. When the split algorithm terminates, one of the created paths will hold those keys that are less than the new key, while the other path will hold keys greater than the new key.

3 Dense Skip Tree Properties

In this section we will show that the number of keys per node of a dense skip tree is distributed according to a geometric distribution, and that the expected height of the tree of N keys is bounded

by $\log_Q N$. Combining these two results will yield the claim that the expected time for search, insert, and delete operations is $O(\log_Q n)$. Property (D3) of the dense skip tree definition assigns random heights to each key, which is clearly not a specification of the number of keys per node. The proofs are novel to the dense skip tree design, and serve as a completion of the skip tree proofs that are outlined by Messeguer [19]. The proof techniques used in this paper represent a significant departure from the methods outlined in the original skip tree paper. Let H indicate the random variable for the height of some key in the tree. We specify that the heights of the keys are distributed according to a geometric distribution:

$$\Pr(H = h) = q^h p \text{ where } p + q = 1 \text{ and } Q = 1/q$$

Theorem 1. Let S indicate the number of keys per node, which is defined to be the size of a node. The mean value of S is $1/q$ and the variance of S is p/q^2.

Because the heights are assigned to keys independently of natural ordering of the keys, we may sort the heights based on the natural ordering of the keys in the data structure. Then let H_i represent the height of key K_i in some sorted sequence of keys $\{K_0, K_1, K_2, \ldots\}$. The size of a node is the the number of keys in the sequence observed with height $H_o = h$, before a key is found with height greater than h. This is because smaller heights do not divide the current node, while larger heights signal the start of a new node (see Figure 1b). The sequence of heights $\{H_0, H_1, H_2, \ldots\}$ can be characterized by a Markov chain with three possible states for each H_i random variable:

(INC) Increment state :
$(H_i = h)$ and $(H_0, H_1, H_2, \ldots, H_{i-1} \leq h)$

(NEU) Neutral state :
$(H_i < h)$ and $(H_0, H_1, H_2, \ldots, H_{i-1} \leq h)$

(TER) Terminating state :
At least one of $\{H_0, H_1, H_2, \ldots, H_i\}$ is greater than h. This is an absorbing state.

The transition matrix A is given by:

$$A = \begin{array}{c} \\ INC \\ NEU \\ TER \end{array} \begin{array}{c} \begin{array}{ccc} INC & NEU & TER \end{array} \\ \begin{pmatrix} p(H=h) & p(H<h) & p(H>h) \\ p(H=h) & p(H<h) & p(H>h) \\ 0 & 0 & 1 \end{pmatrix} \end{array} = \begin{bmatrix} q^h p & 1-q^h & q^{h+1} \\ q^h p & 1-q^h & q^{h+1} \\ 0 & 0 & 1 \end{bmatrix}$$

The expected value for the size of a node, $E(S)$, is entry $(1,1)$ of the fundamental matrix F, which gives the expected number of times the process will be in transient state INC given that it started in state INC [20, 21]. The variance $Var(S)$ is entry $(1,1)$ of the variance matrix V. Let Θ represent the transition matrix for the transient states.

$$\Theta = \begin{bmatrix} q^h p & 1-q^h \\ q^h p & 1-q^h \end{bmatrix}$$

$$F = (I-\Theta)^{-1} = \begin{bmatrix} 1-rp & 1-r \\ -rp & r \end{bmatrix}^{-1} = \begin{bmatrix} 1/q & 1/qr - 1/q \\ p/q & 1/qr - p/q \end{bmatrix} \quad \text{where } r = q^h$$

Let $s = 1/q(1/r - 1)$ and $t = 1/q(1/r - p)$.

$$
\begin{aligned}
V &= F(2F_{dg} - I) - F_{sq} \\
&= \begin{bmatrix} 2/q^2 - 1/q & s(2t-1) \\ 2p/q^2 - p/q & s(2t-1) \end{bmatrix} - \begin{bmatrix} 1/q^2 & s^2 \\ p^2/q^2 & t^2 \end{bmatrix} = \begin{bmatrix} p/q^2 & s(2t-1) - s^2 \\ p/q^2 & s(2t-1) - t^2 \end{bmatrix}
\end{aligned}
$$

Theorem 2. The probability mass function of S is $\Pr(S = s) = p^{s-1}q$ for $s \geq 1$.

To determine the probability mass function of S, reduce the problem to a well-known game from probability analysis. Suppose a game consists of independent turns (τ) at which one of three mutually exclusive events can occur [22, 23]. The events have been labeled with their transition states from the Markov process.

(INC) The game continues with the addition of one point to the score. Let $a = \Pr(\tau = INC) = \Pr(H = h)$.

(NEU) The game continues without addition to the score. Let $b = \Pr(\tau = NEU) = \Pr(H < h)$.

(TER) The game terminates without addition to the score. Let $c = \Pr(\tau = TER) = \Pr(H > h)$.

Let X represent the random variable for the number of points scored in a game. The probability mass function of X will be determined through its probability generating function. The generating function of X is determined by application of the partition theorem for expectation. The reduction to the problem of computing node size is made by $S = X + 1$, as X fails to account for the initial key K_0 with height $H_0 = h$.

$$
\begin{aligned}
E(X) &= a{\cdot}E(X|INC) + b \cdot E(X|NEU) + c \cdot E(X|TER) \\
g_X(z) &= E(z^X) = a \cdot E(z^X|INC) + b \cdot E(z^X|NEU) + c \cdot E(z^X|TER) \\
E(z^X|INC) &= E(z^{X+1}) = zE(z^X) \\
E(z^X|NEU) &= E(z^X) \\
E(z^X|TER) &= \textstyle\sum_{x=0}^{\infty} z^x \Pr(X = x|TER) = (s^0)(1) + \sum_{x=1}^{\infty}(s^x)(0) = 1 \\
g_X(z) &= a \cdot zg_X(z) + b \cdot g_X(z) + c \\
&= \frac{c}{1-b-za} = \frac{c}{1-b}\left(1 - \frac{za}{1-b}\right)^{-1} = \frac{c}{1-b}\sum_{k=0}^{\infty}\left(\frac{za}{1-b}\right)^k \\
\Pr(X = x) &= \frac{c}{1-b}\left(\frac{a}{1-b}\right)^x = \frac{q^{h+1}}{1-(1-q^h)}\left(\frac{q^h p}{1-(1-q^h)}\right)^x = p^x q \\
\Pr(S = s) &= \Pr(X = s-1) = p^{s-1}q \quad \text{for } s \geq 1
\end{aligned}
$$

Corollary 1. Search, insert, and delete operations on a dense skip tree have expected cost $O(\log_Q n)$.

To derive estimates for the height of the tree, let M_n represent the maximum height observed on a sequence of n keys. M_n is an upper bound on the longest path from the root to the leaves of the tree. It has been shown that $E(M_n) = \log_Q n + \frac{\gamma}{L} + \frac{1}{2} - \delta(\log_Q n) + O\left(\frac{1}{n}\right)$, where $L = \log Q$, γ is Euler's constant, and $\delta(x)$ is a periodic function of period 1 and mean 0 [24]. Thus the expected height of the tree is bounded by $\log_Q n$ for $n \gg 0$ as $\log_Q n$ is the dominating term. Theorem 1 has shown that the expected size of a node is a constant. If the expected height of the tree is bounded by $\log_Q n$, then the expected cost for search, insert, and delete operations is $O(\log_Q n)$.

Theorem 3. The dense skip tree has fewer pointers per item than either a binary tree or a skip list.

The space efficiency of the dense skip tree can be estimated by the expected number of pointers per element of the tree. Keys cannot migrate up or down the tree once they have been inserted. Therefore leaf nodes do not need storage allocated for children pointers. p is the fraction of keys that reside at the leaves of the tree, while q is the fraction of keys that reside above the leaves. The expected number of pointers per node in the upper level of the tree is equal to the expected number of keys per node plus one. Therefore the expected number of pointers per key is: $q \cdot (E(S) + 1) + p \cdot (0) = q \cdot (\frac{1}{q} + 1) + 0 = 1 + q$. This is more space-efficient than the expected number of pointers per key in a binary tree (2 pointers per key) and the expected number of pointers per key in a skip list ($1/p$ pointers per key).

In summary, the basic operations on a dense skip tree share the same asymptotic costs for their expected values as the skip list and the skip tree. The asymptotic expected costs of search, insert, and delete operations on dense skip trees, skip trees, and skip lists equal the asymptotic worst-case costs of these operations on balanced binary trees. In section 5, it is shown that dense skip trees make efficient use of spatial locality to outperform the other data structures in practice, when the working set cannot be contained in cache.

4 Related Work

The skip tree was introduced by Messeguer [19] as a generalization of the skip list with simpler algorithms for concurrent insert and delete operations. Several papers have noted the similarities between skip lists and other related data structures [25–28]. Comparisons are discussed between skip lists and randomized treaps [26], and skip lists and randomized binary search trees [29]. The treap data structure is a hybrid of a tree and a heap; hence the name 'treap'. Each node of a treap stores a key, a priority, and left and right children pointers. The keys of a treap are arranged in sorted order and the priorities are arranged in heap order. Randomized binary search trees are a refinement of randomized treaps in which the priorities are random integers drawn from the interval $[0, n]$ where n is the current size of the tree.

Two related techniques for reducing cache misses in data structure design are cache-oblivious data structures [30, 31] and cache-conscious data structures [8]. Cache-oblivious algorithms are designed to perform an asymptotically optimal number of memory transfers given any memory hierarchy and at all levels of the hierarchy. In contrast, cache-conscious data structures are designed with explicit knowledge of either one or several block line sizes in the memory hierarchy. Cache-conscious B-trees store fewer pointers than regular B-tree by storing all children nodes contiguously and keeping only the pointer to the first child node. The dense skip tree is distinguished from the cache-conscious B-trees by the employment of internal fragmentation in cache-conscious B-tree nodes to maintain spatial locality.

5 Performance Analysis

The dense skip tree, the skip list, and the redblack tree were compared in these performance benchmarks. Throughput tests are conducted on the dense skip tree measuring the number of operations per millisecond under varying workloads of search, insert, and delete operations. Three synthetic performance benchmarks were created using a random uniform distribution of integer keys and values. All keys per run are guaranteed to be unique. The dense skip tree, the skip list, the red-black tree were compared in these performance benchmarks. The dense skip tree and the skip list were implemented in C by the authors, while the red-black tree implementation comes

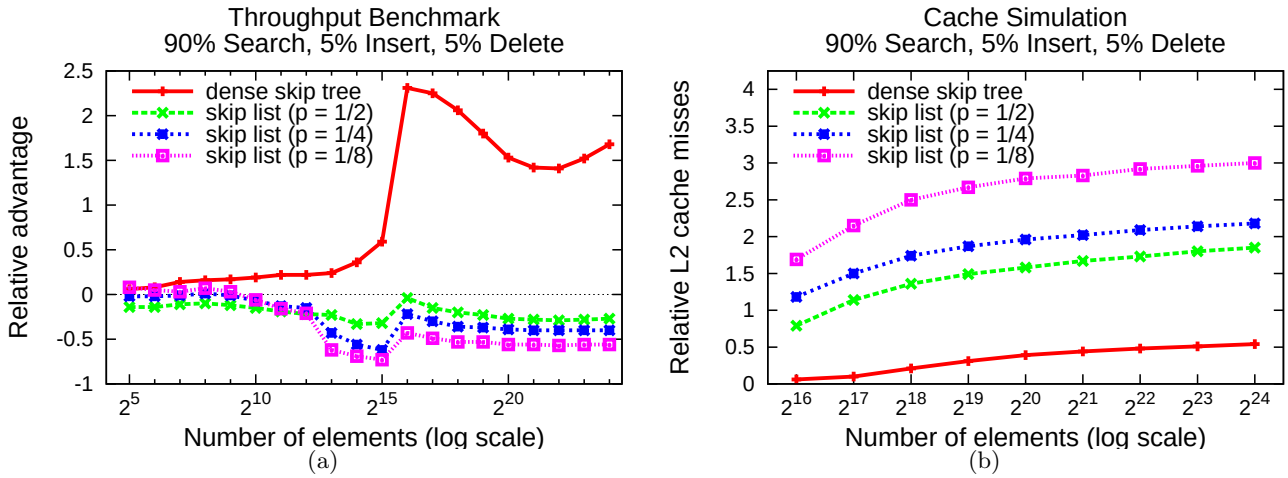


Figure 2: (a) Benchmark performance and (b) simulation results for 90% search, 5% insert, and 5% delete operations.

from the standard template library (STL) map container that is provided by C++. The tests were compiled using g++ version 4.2.3 with -O2 and -finline-functions. The benchmarks were run on a 3.0 GHz Pentium 4 workstation. The skip list was measured with p values of $1/2$, $1/4$, or $1/8$, while the dense skip tree was measured with a q value of $1/16$.

Performance benchmarks were completed under three different loads of search, insert, and delete operations. Figure 2a shows the results for a workload of 90% search, 5% insert, and 5% delete operations. Figure 3a shows 33% search, 33% insert, and 33% delete operations, and Figure 3b shows 0% search, 50% insert, and 50% delete operations. All the figures are plotted with relative advantage on the y-axis, as compared to the C++ STL red-black implementation as a baseline (horizontal dashed line in the figure). If the throughput ratio as compared to the red-black tree implementation is greater than one ($r > 1$), then relative advantage is defined as $1 - r$. If r is less than one, then relative advantage is defined as $1/(1-r)$. Relative advantage allows performance gains and losses to be plotted together, such that their respective distances from the x-axis can be compared in a meaningful way. For example, a relative advantage of 1.0 implies a $\times 2$ change in throughput, while a relative advantage of -1.0 implies a $\times 1/2$ change in throughput.

Both advantages and disadvantages of the skip list are best observed under the scenario of intense insertion and deletion (Figure 3b). When the working set fits entirely in cache, the lack of global rebalancing in skip list algorithms provides a performance gain. Note that the relative advantage is greatest when the skip list behaves as a linked list (when p is $1/8$, and thus $7/8$ of the items have zero height). However, the smaller parameters of p are disadvantageous when the working set extends beyond the memory wall. For large working sets, it is shown that a p-value of $1/8$ performs the worst under intense insertion and deletion.

In order to test the hypothesis that dense skip tree search operations efficiently exploit the memory hierarchy, the benchmark from Figure 2a was run in a cache simulator. The results of the cachegrind simulator are shown in Figure 2b. Cachegrind is a cache profiler from the valgrind instrumentation framework [32]. L1 and L2 cache sizes and cache line associativity parameters were set to match the Pentium 4 architecture (L1 data cache 16 kbytes, 8-way set associative, 64-byte line size; L2 cache 1024 kbytes, 8-way set associative, 64-byte line size). No cache misses were observed for data sets smaller than 2^{16} elements. L2 cache misses are plotted on the y-axis, as

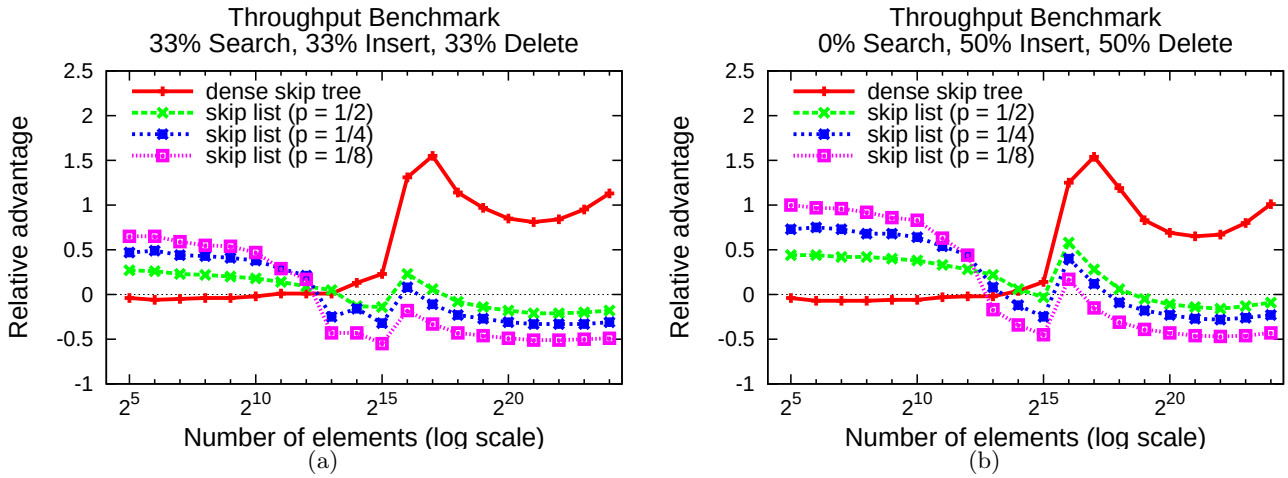


Figure 3: Benchmark performance under (a) 33% search, 33% insert, and 33% delete or (b) 0% search, 50% insert, and 50% delete.

compared to the cache misses incurred by the C++ STL red-black implementation. The simulation confirms that spatial locality of reference is highly preserved in dense skip trees as compared to the skip list or the self-balancing binary tree. The dense skip tree incurs 0.10 to 0.54 as many cache misses per operation as compared to the red-black tree, while the skip list incurs 0.8 to 3.0 cache misses per operation as compared to the red-black tree.

6 Summary

We have introduced the dense skip tree, a cache-conscious randomized data structure. The dense skip tree does not have any global rebalancing requirements; the sizes of nodes are a probabilistic quantity that is derived from the distribution of the height of keys. The dense skip tree was proven to have the same asymptotic expected costs as the skip list for search, insertion, and deletion operations. It has been shown to outperform the skip list when the data structures cannot fit in cache memory. A cache simulation of the performance benchmarks support the claim that the dense skip tree is an attractive alternative to the skip list for large working sets. The randomization technique for the probabilistic cache-conscious access of data is a novel addition to the set of deterministic techniques that have been employed to mask the effects of the "memory wall". The cache-conscious dense skip tree is particularly appealing when considering a concurrent shared memory architecture. A concurrent dense skip tree should offer performance improvements over a concurrent skip list, as the memory wall for multicore processors must now contend with the requirements imposed by the cache coherence protocol.

References

[1] William A. Wulf and Sally A. McKee. Hitting the memory wall: Implications of the obvious. *ACM SIGARCH Computer Architecture News*, 23:20–24, March 1995.

[2] Sally A. McKee. Reflections on the memory wall. In *Proceedings of the 1st conference on Computing frontiers*, April 2004.

[3] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, fourth edition edition, 2007.

[4] David A. Patterson. Latency lags bandwidth. *Communications of the ACM*, 47:71–75, 2004.

[5] Michael E. Wolf and Monica S. Lam. A data locality optimizing algorithm. In *Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation*, 1991.

[6] Steve Carr, Kathryn S. McKinley, and Chau-Wen Tseng. Compiler optimizations for improving data locality. *ACM SIGOPS Operating Systems Review*, 28(28):252 – 262, December 1994.

[7] Jun Rao and Kenneth A. Ross. Making B+-trees cache conscious in main memory. In *Proceedings of the 2000 SIGMOD Conference*, 2000.

[8] Trishul M. Chilimbi, Mark D. Hill, and James R. Larus. Cache-conscious structure layout. In *Proceedings of the ACM SIGPLAN 1999 conference on Programming language design and implementation*, 1999.

[9] Trishul M. Chilimbi and James R. Larus. Using generational garbage collection to implement cache-conscious data placement. In *Proceedings of the 1st international symposium on Memory management*, 1998.

[10] Brad Chalder, Chandra Krintz, Simmi John, and Todd Austin. Cache-conscious data placement. In *Proceedings of the Eighth International Conference on Architecture Support for Programming Languages and Operating Systems*, 1998.

[11] William Pugh. Skip lists: A probablistic alternative to balanced trees. *Communications of the ACM*, 33(6):668–676, June 1990.

[12] Nicholas J. A. Harvey, Michael B. Jones, Stefan Saroiu, Marvin Theimer, and Alec Wolman. Skipnet: a scalable overlay network with practical locality properties. In *Proceedings of the 4th conference on USENIX Symposium on Internet Technologies and Systems*, volume 4, 2003.

[13] James Aspnes and Gauri Shah. Skip graphs. *ACM Transactions on Algorithms*, 3(4):384–393, 2007.

[14] Eric N. Hanson and Theodore Johnson. Selection predicate indexing for active databases using interval skip lists. *Information Systems*, 21(3):269–298, 1996.

[15] Sunil Arya and David M. Mount. Approximate nearest neighbor queries in fixed dimensions. In *Proceedings of the fourth annual ACM-SIAM Symposium on Discrete algorithms*, pages 271–280, 1993.

[16] Mikhail Fomitchev and Eric Ruppert. Lock-free linked lists and skip lists. In *Proceedings of the twenty-third annual ACM symposium on Principles of distributed computing*, pages 50–59, 2004.

[17] Hakan Sundell and Philippas Tsigas. Scalable and lock-free concurrent dictionaries. In *Proceedings of the 2004 ACM symposium on Applied computing*, pages 1438–1445, 2004.

[18] Maurice Herlihy, Yossi Lev, Victor Luchangco, and Nir Shavit. A provably correct scalable concurrent skip list. In *Proceedings of the 10th International Conference On Principles of Distributed Systems (OPODIS)*, 2006.

[19] Xavier Messeguer. Skip trees, an alternative data structure to skip lists in a concurrent approach. *Theoretical Informatics and Applications*, 31(3):251–269, 1997.

[20] John G. Kemeny and J. Laurie Snell. *Finite Markov Chains*. Springer-Verlag, 1976.

[21] Charles M. Grinstead and J. Laurie Snell. *Introduction to Probability: Second Revised Edition*. American Mathematical Society, 1997. Available online under the GNU Free Documentation License.

[22] D.V. Lindley. *Introduction to Probability and Statistics From a Bayesian Viewpoint*, volume 1. Cambridge University Press, 1965.

[23] A. Thavaneswaran. Course notes for 5.305: Probability models. Available online at `http://home.cc.umanitoba.ca/~thavane`., January 2005.

[24] Wojciech Szpankowski and Vernon Rego. Yet another application of a binomial recurrence. order statistics. *Computing*, 43:401–410, 1990.

[25] J. Ian Munro, Thomas Papadakis, and Robert Sedgewick. Deterministic skip lists. In *Proceedings of the third annual ACM-SIAM symposium on Discrete algorithms*, pages 367 – 375, Orlando, Florida, 1992. Society for Industrial and Applied Mathematics.

[26] Raimund Seidel and Cecilia R. Aragon. Randomized search trees. *Algorithmica*, 16:464–497, 1996.

[27] Ketan Mulmuley. *Computational Geometry: An Introduction Through Randomized Algorithms*. Prentice Hall, 1994.

[28] Brian C. Dean and Zachary H. Jones. Exploring the duality between skip lists and binary search trees. In *Proceedings of the 45th annual ACM southeast regional conference*, 2007.

[29] Conrado Martinez and Salvador Roura. Randomized binary search trees. *Journal of the ACM*, 45(2):288–323, March 1998.

[30] Michael A. Bender, Erik D. Demaine, and Martin Farach-Colton. Cache-oblivious B-trees. *SIAM Journal on Computing*, 35:341–358, 2005.

[31] Lars Arge, Gerth Stolting Brødal, and Rolf Fagerberg. *Handbook on Data Structures and Applications*, chapter 38 Cache-Oblivious Data Structures. CRC Press, 2004.

[32] Nicholas Nethercote and Julian Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *Proceedings of ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation (PLDI 2007)*, 2007.