Efficient External Table Reordering

Walter A. Burkhard
Gemini Storage Systems Laboratory
Department of Computer Science and Engineering
University of California, San Diego
La Jolla, CA 92093-0404, USA
burkhard@cs.ucsd.edu

Abstract

A novel extension to binary-tree table organization suitable for external storage providing successful searches for nearly full tables requiring less than two accesses is presented. The generalizes the "efficient ordering of hash tables" effort presented in 1979. Both the experimental and analytical results demonstrate the reductions possible. This method places no restrictions on the the table configuration parameters and requires no additional space per bucket. The insertion runtime is larger slightly than for ordinary external double hashing.

I. Introduction

Hashing is a well-known data indexing technique organizing externally stored data. Numerous schemes exist for handling collisions such as open-addressing; each record determines the probe sequence used to store or retrieve it. Open addressing hashing is an ideal table implementation scheme for fixed capacity devices such as disk drives. To store a record, it is placed in the first unfilled bucket of its probe sequence; to search for a record, buckets designated by its probe sequence are examined in order until finding it or a unfilled bucket, not containing it, is encountered indicating the record is not present. Uniform hashing, introduced by Peterson [13], is an idealized model that maps records to random permutation probe sequences. Double hashing is an efficient scheme to generate the probe sequence for a record. Recently Lueker and Molodowitch [9] as well as Guibas and Szemeredi[6] have shown double hashing to be asymptotically equivalent to the ideal uniform hashing.

In case the bucket capacity is one, the performance of uniform hashing has been analyzed by Peterson [13], Morris [11] and Knuth [7]. Ullman [16] raised an optimality question and presented a model for discussing it. More recently, Yao [17] has shown that uniform hashing is optimal among all open-addressing schemes with respect to the expected successful search length; he poses the question – is uniform hashing also optimal among all open-addressing schemes with respect to the expected unsuccessful search length. Larson [8] analyzes uniform hashing for bucket capacity exceeding one. Blake and Konheim [1] provides analysis of buckets with capacity exceeding one using linear probing collision resolution.

A number of other open addressing schemes have been proposed to improve successful search length that move records during insertion – Pagh and Rodler[12], Brent[2], Gonnet and Munro[5]. The Brent as well as Gonnet and Munro binary tree hashing schemes move records forward in their probe sequence; Pagh and Rodler's Cuckoo hashing moves records either direction in their probe sequences. Rivest [15] considered the optimal arrangement for best possible average successful search length. All of these schemes utilize single record bucket configurations.

Our contribution is to analyze an extended version of tree hashing [5] which features increased bucket sizes compatible with external storage devices. With increased bucket capacities, the successful search length is reduced approaching 1 as the table (disk drive) fills; the average unsuccessful search length decreases as well. For a table configured with 32 slots per bucket, the expected and average successful search lengths are approximately 1.1 for a full table. Several simulation results are presented as well.

II. External Tree Hashing

External tree hashing is an extension of binary-tree hashing[5] in which each bucket contains b slots. In case b is one, the scheme reverts to binary-tree hashing. The scheme is a "logically-recursive" version of Brent hashing[2].

External tree hashing centers on the idea that if the to-be-inserted record collides with a bucket within the table,

Proceedings of the Fifteenth IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems (MASCOTS'07)
978-1-4244-1854-1/08/$25.00 ©2008 IEEE

it makes no difference which of these $b + 1$ records is moved to an unfilled bucketr; more choices provide performance opportunities. We use the term *collide* to designate a record probing a bucket containing b records. The next probe sequence positions for the to-be-inserted record and the b records within the bucket are considered now. If the bucket next on the probe sequence of the to-be-inserted record is unfilled, the to-be-inserted record is placed here. If the bucket next on the probe sequence of the ith record $0 \leq i < b$ within the bucket is unfilled, the ith record within the bucket is moved one bucket further down its probe sequence and the to-be-inserted record is included in its place. In the event all the next $b + 1$ buckets are filled, each of the next $(b + 1)^2$ nodes at the next level are considered in exactly the same manner. The **insert** method, which is to include the to-be-inserted record within the table, continues these ideas by traversing a $b+1$–ary tree in breadth-first order until an unfilled bucket is found.

The insert method is demonstrated within figure 1 with insertion of *gnu* into a table with $b = 2$ slots per bucket. The records *cat*, *dog*, *ape*, ... , *fox* have already been inserted. Accordingly *gnu* is included in bucket 2 after removing *yak*;

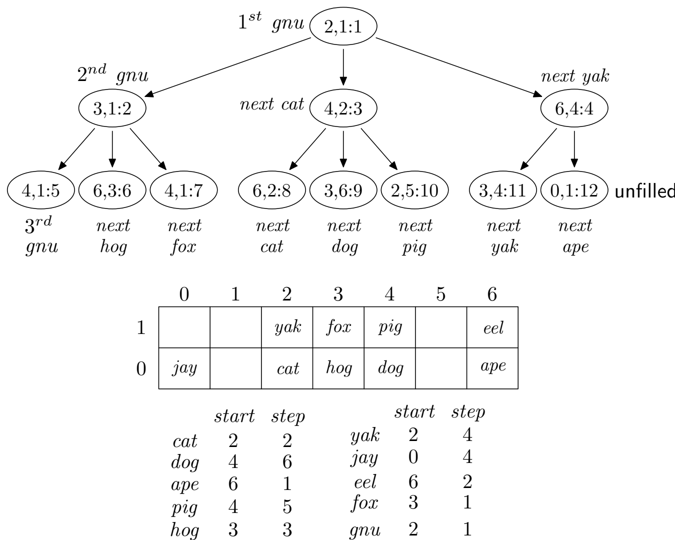

Figure 1. Insertion of *gnu* with table prior to insertion.

yak is included in bucket 6 after removing *ape*; finally *ape* is included in bucket 0.

For a table configured with b slots per bucket, the insert algorithm traverses a $b+1$–ary tree. It manipulates records of the form (*location, step* : *position*) in which *location* is a bucket index, *step* is the double hashing increment for the record being inserted/moved into the *location* bucket, and *position* is the tree array index of the associated node. Figure 2 contains the insert algorithm with queue operations designated *insertQ* and *fetchQ*.

The insert algorithm for ordinary double hashing runs faster on average than our algorithm; nevertheless the expected number of records moved here is less than one as noted by Gonnet and Munro [5]. Our implementation, following [5] represents the insert tree using heap style numbering of nodes with the root at position 1 and node x has children $(b + 1)(x - 1) + i + 2$ for $0 \leq i \leq b$ and parent $(x + b - 1)/(b + 1)$ when $x > 1$, provides reasonable run-times. In the implementation, the tree position of the first node to designate an unfilled bucket specifies how to rearrange the records.

III. Analysis of External Tree Hashing

A table contains n buckets each with capacity for b records. Hashing performance metrics are the *successful search length* as well as *unsuccessful search length* which are calculated analytically as expected values or measured as average values. The successful search length is the number of probes to access a record stored within the table. The unsuccessful search length is the number of probes to determine a record is not stored within the table.

```
insert–1: // insert record data into the queue.
    counter = 1 ;
    insertQ(start = start(data), step = step(data), counter++ ) ;

insert–2: // breath-first tree traversal looking for unfilled table location
    fetchQ(location, step, position ) ;
    if location designates an unfilled bucket then goto insert–3.
    insertQ( (location + step ) mod tableSize, step, counter++ ) ;
    for ( i = 0 ; i < b ; i ++ )
      insertQ( (location + step( table[location][i] ) mod tableSize ,
                                step( table[location][i] ) , counter++ ) ;
    continue insert–2 ;

insert–3: // determine path from tree position to root.
    for ( i = 0, path [0] = position ; path[i] > 1 ; i ++ )
      path[i+1] = ( path[i] + b – 1 )/(b+1) ;

insert–4: // reorganize records following path from root to position
    for ( index = start ; index != location ;
                                index = ( index + step ) mod tableSize )
      if ( ( slot = ( path[ ––i] – 2 ) ) mod (b+1) > 0 ) // not leftmost link.
        step = step( table[index][slot – 1] ) ;
        swap ( table[index][slot – 1], data ) ;

insert–5: // insert data into location in first available slot.
    table[location][first available slot] = data ;
```

Figure 2. Insert algorithm: b slots per bucket.

For either calculation, the probability a bucket is unfilled $\beta = \beta(b, m, n)$ will be needed; m is the number of records stored. Let $N_i(m)$, for $0 \leq i \leq b$, be the expected number of buckets containing exactly i records when m records have been inserted into a table configured with n buckets.

The recurrences, following the total expectation rule [14], obtain the $b+1$ expected values as a function of m and n

$$\begin{aligned} N_0(m+1) &= N_0(m) - N_0(m)/(n - N_b(m)) \\ N_i(m+1) &= \\ &\quad N_i(m) + (N_{i-1}(m) - N_i(m))/(n - N_b(m)) \\ N_b(m+1) &= N_b(m) + N_{b-1}(m)/(n - N_b(m)) \end{aligned} \quad (1)$$

for $0 \le m \le b\,n$ and $0 \le i \le b$. The initial conditions, with all buckets empty, are $N_i(0) = n\,\delta_{i,0}$. Finally, the buckets are filled completely with $b\,n$ records and the final values are $N_i(b\,n) = n\,\delta_{i,b}$.

Equation 1 is determined as follows. Suppose m records have been inserted; during insertion of the $m+1^{st}$ record, the number of buckets containing i records can change by at most one. The probability of these events will vary as the number of unfilled buckets varies; the expression $(n - N_b(m))$ designates the number of unfilled buckets after m insertions. The probability the number of buckets containing i records increases by one is $N_{i-1}(m)/(n - N_b(m))$ and the probability the number decreases by one is $N_i(m)/(n - N_b(m))$. Furthermore, the probability the number containing i records does not change is given by $1 - (N_{i-1}(m) - N_i(m))/(n - N_b(m))$. Then, using the total expectation rule,

$$N_i(m+1) = \\ N_i(m) + (N_{i-1}(m) - N_i(m))/(n - N_b(m))$$

This is the principal equation within (1).

The empty bucket count cannot increase as m increases and

$$N_0(m+1) = N_0(m) - N_0(m)/(n - N_b(m)).$$

Similarly, the filled bucket count cannot decrease as m increases and

$$N_b(m+1) = N_b(m) + N_{b-1}(m)/(n - N_b(m)).$$

Let $q_i(\alpha)$ be the probability a bucket contains i records for a table with loading factor $\alpha = m/(bn)$. Then taking the limit as $n \to \infty$ in equations (1), we obtain

$$\begin{aligned} \frac{dq_0}{d\alpha} &= -\,b \times q_0(\alpha)/(1 - q_b(\alpha)) \\ \frac{dq_i}{d\alpha} &= b \times (q_{i-1}(\alpha) - q_i(\alpha))/(1 - q_b(\alpha)) \quad (2) \\ \frac{dq_b}{d\alpha} &= b \times q_{b-1}(\alpha)/(1 - q_b(\alpha)). \end{aligned}$$

A bucket is filled with probability

$$\beta = \beta(b, m/(bn)) = q_b(\alpha). \quad (3)$$

The system of $b+1$ non-linear equations evidently does not have a closed form solution. Figure 3 presents the probability values $q_b(\alpha)$ for various bucket sizes; the values are obtained from the difference equations 1.

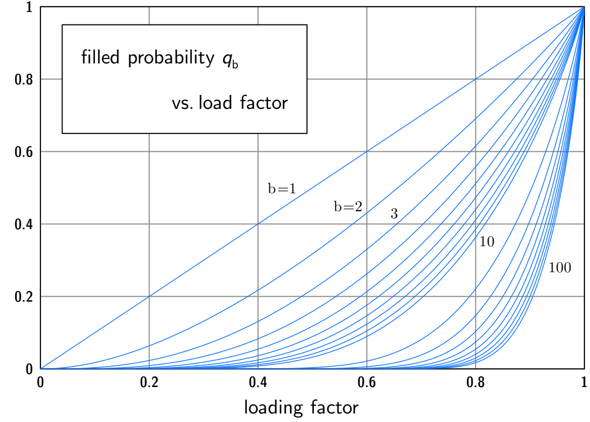

Figure 3. Filled bucket probability versus loading factor.

A. Successful search length

Initially assume the probes followed within a successful search to be random and independent; accordingly the number of probes j required to locate an unfilled bucket within the table has geometric distribution with parameter β, that is $(1-\beta)\beta^{j-1}$ where β is the probability a bucket is filled. Let IC designate an insertion incremental cost which is determined as the number of tree levels traversed to locate the first unfilled bucket; the root has cost = 1. The number of filled tree nodes traversed during an insertion must be less than $((b+1)^j - 1)/b$ if IC is no greater than j.

$$\begin{aligned} \text{Prob}\{IC \le j\} &= \\ &\quad (1-\beta)(1 + \beta + \cdots + \beta^{((b+1)^j - 1)/b - 1}), \\ \text{Prob}\{IC > j\} &= \beta^{((b+1)^j - 1)/b}. \end{aligned}$$

The expected incremental cost E[IC] is

$$\sum_{j \ge 0} \text{Prob}\{IC > j\} = \sum_{j \ge 0} \beta^{((b+1)^j - 1)/b}. \quad (4)$$

The doubly exponential E[IC] sum converges rapidly for $0 \le \beta < 1$. Recall β depends upon m the number of records within the table. The expected successful search length E[L_S] is

$$\begin{aligned} \text{E}[L_S] &= \frac{1}{m} \sum_{k=0}^{m-1} \text{E}[\,IC(\beta(b, k/(bn)))\,] \quad (5) \\ &= \frac{1}{m} \sum_{k=0}^{m-1} \sum_{j \ge 0} \beta(b, k/(bn))^{((b+1)^j - 1)/b}. \end{aligned}$$

This formulation for E[L_S] is quite reasonable for load factors no greater than 0.8; however as the table fills, equation 5 errs by a few percent. As b increases from 1 to 64,

the relative error diminishes from approximately 7% to less than 1%. Figure 6 presents these approximate values together with the experimental values.

To improve the expected successful search length values, we note that the "short chains of probe positions" growth rate exceeds the "random" growth rate. Many of the nodes within the insert tree designate a short chain; we follow the approach first used by Brent [2] and then Gonnet and Munro [5], capturing this increased growth rate phenomenon, to obtain a more accurate model. Define $p_i(\beta)$ to be the probability that, a record R in position s of its probe sequence, will find at least the next i buckets specified by its probe sequence to be filled. This sequence of i filled buckets will be referred to as a *chain* of R. The quantity $\beta p_i(\beta)$ represents the probability of finding a chain of length at least i starting at any location. We have, for $v = 1, 2, \ldots$,

$$\frac{\mathrm{d}(\beta p_v)}{\mathrm{d}\beta} = \beta^v + \frac{1}{1-\beta}\sum_{i=0}^{v-1} \beta^{v-i}(\, p_i(\beta) - p_{i+1}(\beta)\,) + \sum_{i=0}^{v-1} \beta^{v-i-1} Q_i.$$

With the initial conditions $p_v(0) = 0$ for $v = 1, 2, \ldots$, this system of differential equations, together with $p_0(\beta) = 1$, defines p_v.

The summand β^v is the probability of creating a new chain of length at least v; for example, in figure 1 the left-edge nodes $1, 2, 5$ designate the chain $2, 3, 4$ for *gnu*. The second summand is the probability of extending unrelated chains when filling a bucket; for example, in figure 1 when *ape* is included in bucket 0, any possibly unrelated chain that terminated in bucket 0 will now be extended. Note if a chain is extended by a single location, it may be extended several more positions by random placement of unrelated records. Finally the $\sum_{i=0}^{v-1} \beta^{v-i-1} Q_i$ summand represents the extension of a chain originating in the tree traversal. In figure 1, bucket 0 has higher than random probability of being filled since the buckets $2, 3, 4, 6$ are filled. Any chain positions beyond bucket 0 for *ape* in its new bucket are unrelated but are included to create a chain of length at least v. Without the final summand, the p_v solution is α^v for the $b = 1$ configuration; this is the double hashing solution.

The expression Q_i designates the sum of the probabilities of all trees for which a breadth-first traversal ends at a chain of length at least i. The first few levels of the tree in figure 4 for a $b = 2$ configuration shows the chain associated with each node; such a chain will exist if the node is first visited when associated with an unfilled bucket. For example, the zeros in the second level indicate that a chain of length 0 would be exist for the bucket associated with the node; for either node, a record from the bucket specified at the root node would move to the bucket specified at the node in one step. Similarly, the ones in the next level

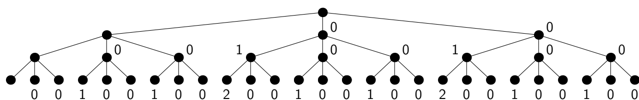

Figure 4. Chains associated with the nodes: b = 2 configuration.

indicate a chain of length 1 is associated with the node if the node designates an unfilled bucket; a record from the bucket specified at the root node would move there in two steps. Whenever the first visited node associated with an unfilled bucket has the label i, a record in the bucket specified by the parent $i+1$ levels above will move to the bucket in $i+1$ steps. All nodes with the non-zero labels are along left edges of their subtree. Each node labeled i within the tree provides a summand for the Q_i expression. Equation 6 contains the initial summands in Q_0. For example, the node in position 3 within the tree is the first node visited with an unfilled associated bucket with probability $\beta^2(1-p_1)$. The root node is associated with a filled bucket with probability β and the node in position 2 is also associated with a filled bucket with probability β and finally the record in the bucket associated with the root has a chain of length 0 with probability $(1-p_1)$; assuming these events are independent provides the result. The node in position 4 is the first node visited with an unfilled associated bucket with probability $\beta^2 p_1(1-p_1)$. The extra p_1 factor is to assure the node in position 3 is associated with a filled bucket; the record must have a chain of length at least 1. All summands are calculated in this manner.

$$\begin{aligned} Q_0 &= \beta^2(1-p_1)(\overbrace{1+p_1}^{second\ level} + \overbrace{\beta p_1^2 + \beta p_1^3 + \beta p_1^3 p_2 + \beta p_1^4 p_2 + \beta p_1^4 p_2^2 + \beta p_1^5 p_2^2}^{third\ level} + \cdots) \\ Q_1 &= \beta^3(p_1-p_2)(\overbrace{p_1^3 + p_1^4 p_2}^{third\ level} + \overbrace{\beta p_1^7 p_2^2 + \beta p_1^8 p_2^3 + \beta p_1^{11} p_2^3 p_3 + \beta p_1^{12} p_2^4 p_3 + \beta p_1^{15} p_2^4 p_3^2 + \beta p_1^{16} p_2^5 p_3^2}^{fourth\ level} + \cdots) \quad (6) \\ Q_2 &= \beta^4(p_2-p_3)(\overbrace{p_1^{10} p_2^3 + p_1^{14} p_2^4 p_3}^{fourth\ level} + \cdots) \end{aligned}$$

The system of differential equations can be simplified –

$$\frac{\mathrm{d}(\beta p_v)}{\mathrm{d}\beta} = (\beta^v - \beta p_v)/(1-\beta) + \sum_{i=0}^{v-1} \beta^{v-i} p_i + \sum_{i=0}^{i-1} \beta^{v-i-1} Q_i. \quad (7)$$

IC–1: $temp = 1.0$; $IC = 1.0$;

IC–2: for ($i = 1$; ; $i++$)
$$temp = temp \times (\, p_{i-1}\, temp\,)^b \;;$$
$$IC = IC + \beta^i\, temp \;;$$

Figure 5. Incremental cost: b slots per bucket.

The insertion incremental cost $IC(\alpha)$ for general b is shown in figure 5; for $b = 2$ slots per bucket, $IC(\alpha)$ is

$$1 + \beta + \beta^2 p_1^2 + \beta^3 p_1^6 p_2^2 + \beta^4 p_1^{18} p_2^6 p_3^2 + \cdots .$$

The expected successful search length is given by

$$\mathrm{E}[L_S] \;=\; \frac{1}{\alpha}\int_0^\alpha IC(t)\mathrm{d}t$$

which will be evaluated via its differential form –

$$\frac{\mathrm{d}(\mathrm{E}[L_S])}{\mathrm{d}\alpha} \;=\; (IC(\alpha) - \mathrm{E}[L_S])/\alpha. \qquad (8)$$

Our expected successful search values are obtained by numerically integrating the differential equations 2, 7 & 8; expected values as well as the experimental results are presented. The experimental and expected successful search length values together with approximate values, equation 5, are presented in figure 6; the experimental results are for a table configured with 257 buckets for sample size 500.

For $b < 64$ slots per bucket, calculations for the expected successful search length involving no more than a dozen p_v functions produced “identical” values on the 32-bit Intel architecture. Similar calculations for $b = 64$ slots per bucket involves 65 differential equations for the $\beta(64, \alpha)$ probabilities as well as the differential equations for the p_v and $\mathrm{E}[L_S]$ values were not fruitful. A reasonable approach might precalculate approximate β values using equations 1.

B. Unsuccessful Search Length

Tree hashing has no impact on the unsuccessful search length; the increased bucket capacity does however. A length ξ unsuccessful search begins with $\xi - 1$ probes to filled buckets and concludes with an unfilled bucket. Accordingly, with β the probability a bucket is filled,

$$\mathrm{Prob}\{\, L_U = \xi \,\} \;=\; (\, 1 - \beta\,)\beta^{\,\xi - 1}.$$

And the expected unsuccessful search length $\mathrm{E}[\, L_U\,]$ is

$$\mathrm{E}[\, L_U\,] \;=\; \sum_{\xi\,\geq\,1} \xi\, \mathrm{Prob}\,\{\, L_U = \xi\,\} \;=\; \frac{1}{1-\beta}. \qquad (9)$$

Figure 7 presents the expected and experimental unsuccessful search lengths; the table size is 257 and the sample size is 500.

Experimental successful and unsuccessful search length values for $b = 64$ slots per bucket were gathered as well; these are not presented in tables 2 and 3 due to space limitations. However for load factor 0.9922, the average successful search length is 1.063383 and the unsuccessful search length is 8.578711; increasing the load factor to 0.9961, increases the successful search length to 1.066442 and the unsuccessful search length to 15.184357.

IV. Conclusion

External tree hashing provides excellent improvement to the successful search length performance for tables; the expected successful search length for a full table approaches 1 as b increases and is strictly less than 2 whenever $b > 1$. External tree hashing provides performance improvements beyond external double hashing.

In the future, improving the unsuccessful search length beyond equation 9 will be considered; passbits [3, 10] are compatible with external tree hashing and will provide additional unsuccessful search length improvement. The successful search length can also be improved using more than one hash function [4]. These approaches both require additional space.

References

[1] I. F. Blake and A. G. Konheim. Big buckets are (are not) better! *Journal of the ACM*, 24(4):591–606, 1977.

[2] R. P. Brent. Reducing the retrieval time of scatter storage techniques. *Communications of the ACM*, 16:105–109, 1973.

[3] W. A. Burkhard. Double hashing with passbits. *Information Processing Letters*, 96:162–166, 2005.

[4] W. A. Burkhard. External double hashing with choice. In *Proceedings of the International Symposium on Parallel Architectures, Algorithms, and Networks* (ISPAN’05), pages 100–107, 2005.

[5] G. H. Gonnet and J. I. Munro. Efficient ordering of hash tables. *SIAM Journal on Computing*, 8:463–478, 1979.

[6] L. Guibas and E. Szemeridi. The analysis of double hashing. *Journal of Computer and System Sciences*, 16:226–274, 1978.

[7] D. Knuth. *The Art of Computer Programming, Volume 3: Sorting and Searching*. Addison-Wesley, Reading, second edition, 1998.

[8] P.-A. Larson. Analysis of uniform hashing. *Journal of the Association for Computing Machinery*, 30(4):805–819, October 1983.

[9] G. Lueker and M. Molodowitch. More analysis of double hashing. *Combinatorica*, 13(1):83–96, 1993.

[10] P. M. Martini and W. A. Burkhard. Double hashing with multiple passbits. *International Journal of Foundations of Computer Science*, 14(6):1165–1182, 2003.

[11] R. Morris. Scatter storage techniques. *Communications of the Association for Computing Machinery*, 11(1):38–44, January 1968.

[12] R. Pagh and F. R. Rodler. Cockoo hashing. *Journal of Algorithms*, 51(2):122–144, 2004.

[13] W. Peterson. Addressing for random-access storage. *IBM Journal of Research and Development*, 1(2):130–146, 1957.

[14] A. Rényi. *Probability Theory*. North-Holland Publishers, 1970.

[15] R. L. Rivest. Optimal arrangement of keys in a hash table. *Journal of the Association of Computing Machinery*, 25(2):200–209, 1978.

[16] J. Ullman. A note on the efficiency of hash functions. *Journal of the Association for Computing Machinery*, 19(3):569–575, July 1972.

[17] A. Yao. Uniform hashing is optimal. *Journal of the Association for Computing Machinery*, 32(3):687–693, July 1985.

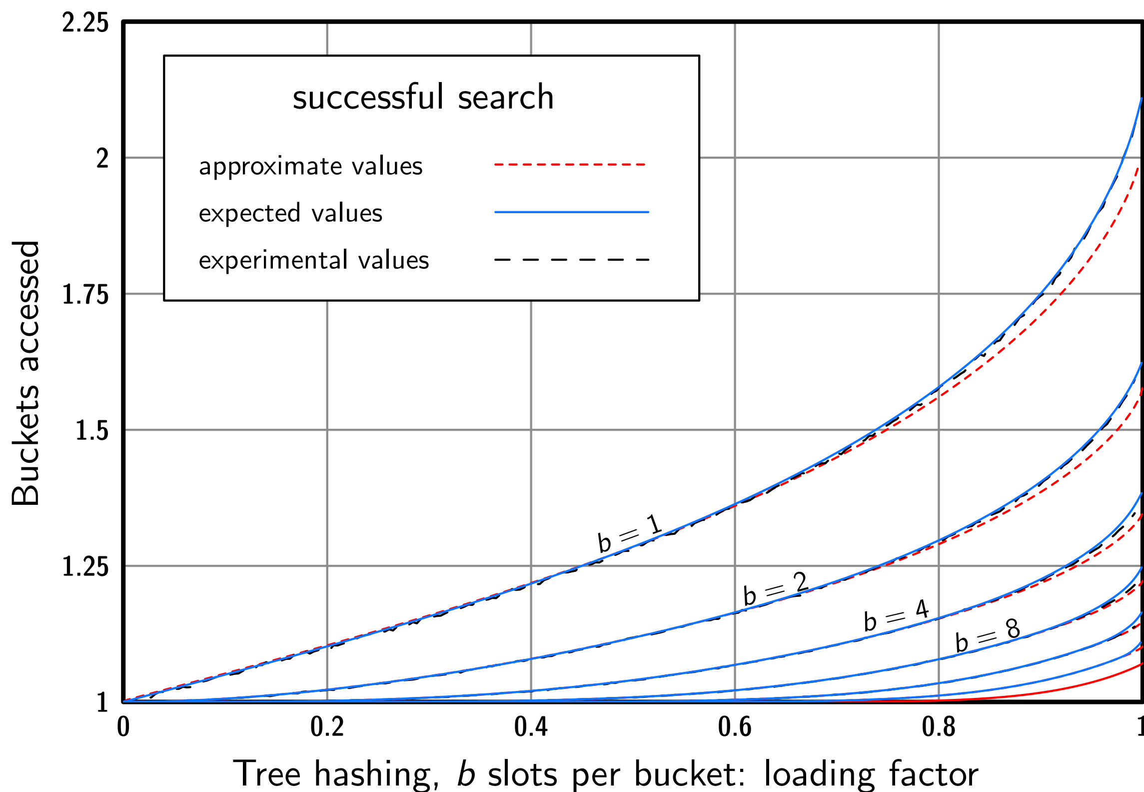

Figure 6. Tree hashing experimental and expected successful search lengths.

Table 1. Tree hashing expected successful search lengths for buckets configured with b = 1,2,4,8,16 and 32 slots.

α	$b=1$	2	4	8	16	32
0.200	1.102085	1.022471	1.002031	1.000034	1.000000	1.000000
0.300	1.157183	1.047190	1.008967	1.000466	1.000003	1.000000
0.400	1.217428	1.079074	1.020296	1.002564	1.000086	1.000000
0.500	1.285124	1.117845	1.040057	1.008604	1.000798	1.000015
0.600	1.363541	1.164518	1.068178	1.021400	1.004008	1.000291
0.700	1.458001	1.222106	1.105416	1.043791	1.013453	1.002453
0.800	1.578628	1.296913	1.154158	1.078458	1.034473	1.011698
0.900	1.750354	1.404389	1.225393	1.129084	1.073612	1.038242
0.950	1.879526	1.484984	1.280442	1.167549	1.102820	1.062840
0.960	1.912771	1.505522	1.294822	1.178013	1.110082	1.069670
0.970	1.950239	1.528474	1.311217	1.189995	1.118583	1.075725
0.980	1.993759	1.554758	1.330492	1.204076	1.129323	1.083256
0.990	2.047161	1.586200	1.354754	1.222356	1.143661	1.094067
0.995	2.079831	1.605220	1.370004	1.234534	1.153026	1.102399
0.999	2.109395	1.623752	1.383734	1.247350	1.164344	1.110000

Table 2. Tree hashing experimental average successful search lengths for buckets configured with b = 1,2,4,8,16 and 32 slots. Table size is 257 buckets, 500 samples per value.

α	$b=1$	2	4	8	16	32
0.1012	1.050923±0.003803	1.005808±0.000912	1.000135±0.000112	1.000000±0.000000	1.000000±0.000000	1.000000±0.000000
0.1984	1.097647±0.003569	1.021471±0.001281	1.001824±0.000286	1.000044±0.000029	1.000000±0.000000	1.000000±0.000000
0.2996	1.155818±0.003352	1.046429±0.001610	1.008026±0.000534	1.000334±0.000079	1.000000±0.000000	1.000000±0.000000
0.4008	1.214097±0.003784	1.076835±0.001653	1.019432±0.000695	1.002502±0.000191	1.000086±0.000031	1.000001±0.000001
0.5019	1.283938±0.003666	1.118884±0.001762	1.039578±0.000861	1.008229±0.000319	1.000802±0.000079	1.000013±0.000009
0.5992	1.360442±0.003847	1.163065±0.001901	1.067714±0.001004	1.020896±0.000488	1.003936±0.000171	1.000259±0.000039
0.7004	1.457056±0.004014	1.221756±0.002051	1.105008±0.001137	1.043304±0.000615	1.013603±0.000304	1.002356±0.000102
0.8016	1.580097±0.004098	1.295029±0.002243	1.154823±0.001211	1.078669±0.000716	1.034773±0.000452	1.011928±0.000222
0.8988	1.744528±0.004852	1.398238±0.002538	1.222134±0.001471	1.128083±0.000791	1.072522±0.000508	1.037634±0.000340
0.9494	1.877098±0.005320	1.478516±0.002760	1.275244±0.001480	1.163694±0.000879	1.101820±0.000555	1.062248±0.000384
0.9611	1.909725±0.005376	1.504425±0.002612	1.290964±0.001449	1.174926±0.000896	1.109585±0.000571	1.069051±0.000377
0.9689	1.944169±0.005309	1.520960±0.002804	1.301538±0.001577	1.183272±0.000902	1.116035±0.000559	1.074697±0.000393
0.9805	1.997794±0.005787	1.552937±0.003017	1.321490±0.001622	1.197600±0.000951	1.126155±0.000588	1.083022±0.000388
0.9883	2.030780±0.005993	1.576547±0.003035	1.337132±0.001582	1.208127±0.000919	1.134122±0.000568	1.089049±0.000383
0.9922	2.059725±0.005865	1.592106±0.002982	1.346786±0.001735	1.215010±0.000954	1.138284±0.000576	1.092767±0.000397

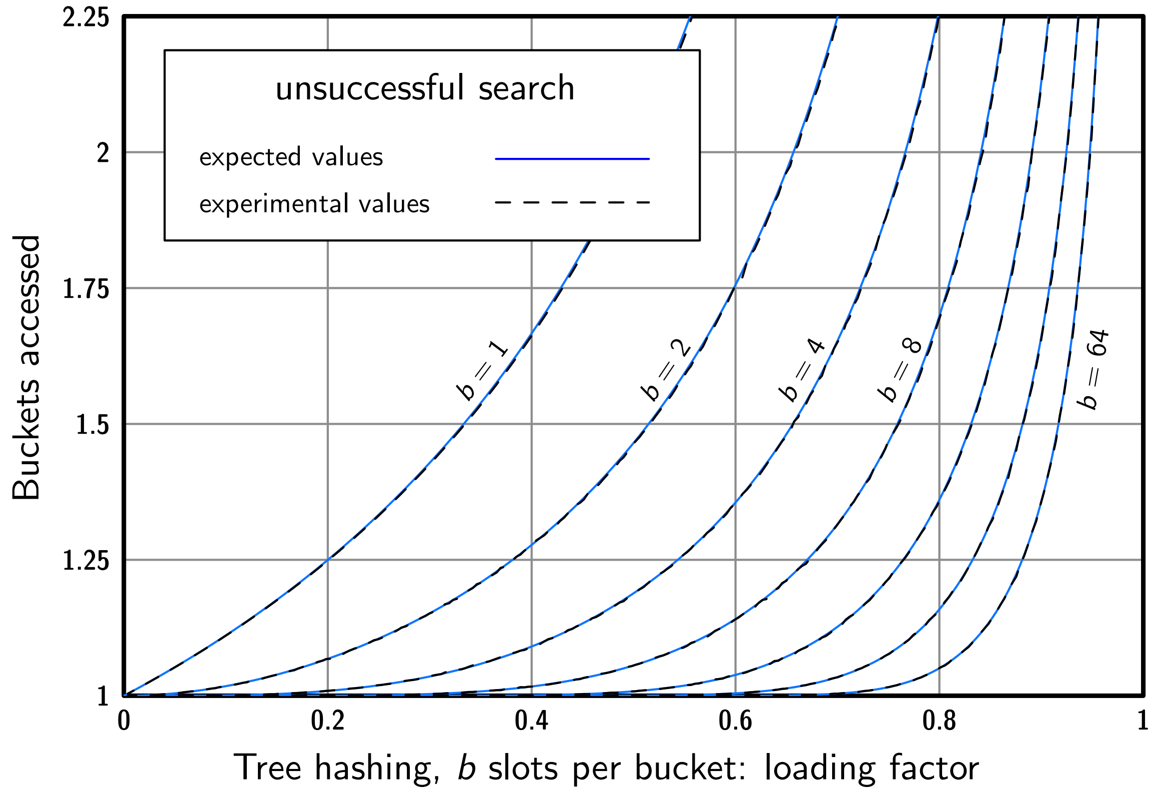

Figure 7. Tree hashing experimental and expected unsuccessful search lengths.

Table 3. Tree hashing experimental average unsuccessful search lengths for buckets configured with b = 1,2,4,8,16 and 32 slots. Table size is 257 buckets, 500 samples per value.

α	$b = 1$	2	4	8	16	32
0.1984	1.246374±0.000049	1.066246±0.001029	1.008933±0.000505	1.000187±0.000073	1.000000±0.000000	1.000000±0.000000
0.2996	1.425337±0.000120	1.150850±0.001543	1.035797±0.000972	1.003184±0.000290	1.000008±0.000015	1.000000±0.000000
0.4008	1.664742±0.000242	1.275862±0.002026	1.089916±0.001416	1.017181±0.000700	1.000943±0.000189	1.000008±0.000015
0.5019	2.000214±0.000426	1.468986±0.002767	1.189499±0.001886	1.054574±0.001155	1.008392±0.000465	1.000226±0.000080
0.5992	2.481650±0.000837	1.746667±0.003935	1.354947±0.002843	1.137848±0.001848	1.037502±0.000960	1.004408±0.000355
0.7004	3.308135±0.001610	2.238548±0.005756	1.652304±0.004280	1.316692±0.002926	1.128050±0.001833	1.033826±0.000979
0.8016	4.962584±0.003691	3.231956±0.009825	2.262825±0.007385	1.706779±0.005016	1.367530±0.003247	1.163321±0.002075
0.8988	9.561129±0.010098	5.960496±0.024213	3.994738±0.017629	2.824622±0.011486	2.103179±0.007493	1.647071±0.004811
0.9494	18.41894±0.020486	11.34992±0.057527	7.366953±0.039676	4.961603±0.025660	3.560006±0.017018	2.640055±0.011779
0.9611	23.44340±0.024505	14.46535±0.079616	9.232873±0.053897	6.216875±0.035780	4.392901±0.023738	3.210190±0.014869
0.9689	28.70215±0.029603	17.64260±0.102795	11.27774±0.071538	7.543621±0.046459	5.299766±0.030219	3.835066±0.020225
0.9805	42.98837±0.027464	26.86239±0.178683	17.19087±0.120713	11.36135±0.075423	7.858265±0.052634	5.547609±0.033350
0.9883	64.50000±0.000000	41.14462±0.299376	26.53968±0.202718	17.77441±0.136416	12.08751±0.089419	8.505359±0.060662
0.9922	86.00000±0.000000	56.46645±0.429307	36.95290±0.294740	24.82009±0.203329	17.17183±0.137737	11.80293±0.091105