

External Double Hashing with Choice

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 external double hashing providing significant reduction to both successful and unsuccessful search lengths is presented. The experimental and analytical results demonstrate the reductions possible. This method does not restrict the hashing table configuration parameters and utilizes very little additional storage space per bucket. The runtime performance for insertion is slightly greater than for ordinary external double hashing.*¹

1. Introduction

Hashing is a well-known data indexing technique for organizing data stored externally. Numerous schemes exist for handling collisions such as open-addressing; each record determines the probe sequence used to store or retrieve it. To store a record, it is placed in the first un-filled 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 un-filled bucket, not containing it, is encountered indicating the record is not present. Uniform hashing, introduced by Peterson [12], 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 Szemerédi [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 [12], Morris [11] and Knuth [7]. Ullman [15] 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 [2] provides analysis of buckets with capacity exceeding one using linear probing collision resolution.

Our approach to significantly improve the search lengths is to utilize more than one hash function. Use of more than one hash function has been previously presented and analyzed for load balancing by Azar et al. [1] and Vöcking [16]. Multiple hash functions were used to improve IP lookups by Broder and Mitzenmacher [3]. Separate chaining collision resolution is utilized within these efforts.

We consider open-addressing double hashing in which during an insertion, the search length for the each hash function is determined and one with the shortest is used to position the record. The probe sequence to follow, for future accesses, is recorded within a predictor bit array. During a fetch, each hash function is evaluated to determine the possible probe sequence but only those probe sequences marked within the predictor bit array are actually followed. With two hash functions significant improvements are possible.

The paper is divided into sections; external double hashing with choice is introduced in section two, the analysis of the run-time is presented in section three, in section four, the experimental and analytical results are presented graphically.

2. External Double Hashing with Choice

External double hashing with choice is an extension of double hashing in which d hash functions from an appropriate family of universal hash functions [4] are used rather than one. In case d is one, the search length performance of

¹ A preliminary version appears within the Proceedings of the 8th International Symposium on Parallel Architectures, Algorithms & Networks, ISPAN'05

the scheme reverts to that of ordinary double hashing. The table, configured with two records per bucket, is augmented with an array of s predictor bits. The hash domain is partitioned into s equal-sized blocks referred to as *varieties* and any variety i access will be associated with the i^{th} predictor bit. For any access, each of the d hash functions determine its associated predictor bit.

We present two table data type methods **insert** and **fetch** for double hashing with choice in figures 1 and 3. The algorithms are very similar to those of ordinary double hashing with the exception that the insert method must try all d probe sequences to select a shortest access path and the fetch method must interleave the appropriate subset of the d probe sequences. In both figures, the *hash* array, contains pointers to each of the d hash functions; this array is initialized during table construction. The **for** statement, in fig-

```

insert ( Data & data )
shortest = n + 1 ;
for ( j = 0 ; j < d ; j ++ ) {
    value = hash[j](data);
    start[j] = value % n ;
    stride[j] = 1 + value % (n - 1) ;
    variety[j] = (value / (n * (n - 1))) % s ;
    count = 0, location[j] = (start[j] - stride[j]) % n ;
    do { location[j] = (location[j] + stride[j]) % n ;
        if duplicate found in bucket[location[j]] return false ;
        count ++ ;
    } while bucket[location] is full ;
    if count < shortest
        shortest = count, best[0] = j, number = 1 ;
    elseif count == shortest
        best[number ++] = j ;
} // for ( ...

select one shortest path j from best array.

copy data into bucket[location[j]] ;
predictor[variety[j]] = true ; return true ;

```

Figure 1. Insert method

ure 1, determines which of the d probe sequences will locate an un-filled bucket with the fewest probes; the indices of these shortest probe sequences are stored within the array *best*. The second phase of the insert involves selecting one of the shortest probe sequences j to use to store the record. There are several strategies possible here; we view the *best* entries to be equally-likely to be chosen. Finally the data is copied into *bucket[location[j]]* and the *variety[j]* predictor is set.

Figure 2 contains an example table configuration created inserting eight records within a table consisting of five buckets and forty-eight predictor bits. Each record, cat, dog, sow . . . , doe, has a pair of probe sequences together with access varieties. The cat record could be inserted either in bucket 1 or 2 as its two probe sequences begin with these two values; both give rise to shortest possible insertion sequences.

		predictor bits											
		0	8	16	24	32	40			sequence	variety	sequence	variety
0:	doe	1	9	17	25	33	41			cat: 1, 2, 3, 4, 0	18	2, 4, 1, 3, 0	1
1:	cat	2	10	18	26	34	42			dog: 2, 4, 1, 3, 0	3	3, 2, 1, 0, 4	41
2:	dog	3	11	19	27	35	43			sow: 1, 4, 2, 0, 3	6	4, 2, 0, 3, 1	21
3:	yak	4	12	20	28	36	44			pig: 2, 3, 4, 0, 1	34	4, 2, 0, 3, 1	2
4:	sow	5	13	21	29	37	45			ape: 2, 0, 3, 1, 4	3	2, 4, 1, 3, 0	14
		6	14	22	30	38	46			bee: 1, 2, 3, 4, 0	43	4, 1, 3, 0, 2	10
		7	15	23	31	39	47			yak: 2, 1, 0, 4, 3	21	3, 2, 1, 0, 4	32
										doe: 1, 0, 4, 3, 2	14	1, 2, 3, 4, 0	3

Figure 2. External Double Hashing with Choice

```

fetch ( Data & data )
for ( j = 0 , number = 0 ; j < d ; j ++ ) {
    value = hash[j](data) ;
    start[j] = value % n ;
    stride[j] = 1 + value % (n - 1) ;
    variety = (value / (n * (n - 1))) % s ;
    location[j] = (start[j] - stride[j]) % n ;
    if predictor[variety] is set
        which[number ++] = j ;
} // for ( ... select possible paths.
while ( number > 0 ) {
    for ( j = 0 ; number > 0 ; j ++ ) {
        location[j] = (location[j] + stride[j]) % n ;
        if bucket[location[j]] contains data
            copy data record and return true ;
        if bucket[location[j]] is not full
            remove probe sequence which[j] from
            consideration and decrement number.
    } // for ( ...
} // while ( ...
return false ;

```

Figure 3. Fetch method

The first probe sequence with variety 18 is selected (arbitrarily); the record is inserted in bucket 1, and predictor 18 is set. The predictor array is indexed by *variety*; a set predictor bit is designated by a gray square ■. The dog record is similar; both probe sequences give rise to shortest possible insertion sequences. The first probe sequence with variety 3 is selected. The dog record is inserted in bucket 2 and predictor 3 is set. The sow record is similar with both probe sequences reaching an empty slot with one probe; the second sequence is selected. The record is inserted into bucket 4 and predictor 21 is set. When doe is inserted, the candidate insertion sequences are not the same length and the second probe sequence is shorter. The record doe is inserted in bucket 0 and predictor 3 is set (again).

The records cat, dog, sow, and doe are individually accessed to demonstrate the fetch method. We utilize the table within figure 2 and the fetch method of figure 3. The **for** statement determines the d probe sequences and the *which* array contains only those with the associated predictor bit set. Here *number* counts the number of probe sequences to follow. The **while** loop interleaves the probe sequences as

well as removing from consideration those that end without success. For the cat record, only its first probe sequence must be followed since predictor 18 is set while predictor 1 is not. The search length is one probe. For the dog record, only its first probe sequence must be followed since predictor 3 is set but not predictor 41; the search length is one probe. The sow record is similar with only one probe sequence being followed. When accessing the doe record, since predictors 21 and 3 both are set, we must interleave the probe sequences; the search length is 7/2 probes since the probe sequences are equally-likely to begin the search.

When accessing ant, a record not within the table, the probe sequences for ant, 2,3,4,0,1 with variety 18 and 4,2,0,3,1 with variety 3, are determined. Since predictors 18 and 3 are both set, the interleaved probe sequences are followed with a total of six probes to determine the record is not within the table.

3. Analysis of External Double Hashing with Choice

A table contains n buckets each with capacity for b records possesses s predictor bits. We associate with the table d hash functions from a universal family of hash functions [4]. The hashing process, when applied to a record, produces a permutation of the bucket addresses. Within double hashing, a probe sequence is an arithmetic progression with successive probes differing by a constant referred to as the *stride* which is determined via the process. When inserting a record, the probe sequence of each of the d hash functions is created and the number of probes X required to reach an unfilled bucket is determined. The record is inserted by following a probe sequence with the smallest X value; this sequence of probes is referred to as the *min-sequence*. Of course, the min-sequence and its associated X value will depend upon the number of records currently stored within the table. This smallest X is the minimum order statistic for the d values [5]. As the table fills, the average of individual minimum order statistics, denoted L_s^{min} , will be of interest. The d hash functions determine the d triples consisting of the initial probe, the step, and the variety values for a record; we assume that the $n(n-1)s$ triples are equally-likely; previous analyses of double hashing also assume the $n(n-1)$ pairs of initial probe and stride values are equally-likely [9, 6].

Hashing performance metrics are the *successful search length* as well as *unsuccessful search length* which are calculated analytically as expected values or measured experimentally 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 in the table. Within either, for a given record the fetch function first determines

whether the predictor bit is set for each hash function. Only the probe sequences associated with set predictor bits need be followed. The insertion method sets exactly one predictor bit; a predictor bit is set with probability $1/s$ during an insert operation. After m insertions, a predictor bit remains unset with probability $(1 - 1/s)^m$ and is set with probability

$$p_m = 1 - (1 - 1/s)^m. \quad (1)$$

A table containing nb records has $p_{nb} \approx 1 - e^{-nb/s}$.

During a successful search access operation, the d hash functions each determine a *variety* for the desired record; at least one of the associated predictor bits must be set. Let \hat{d} count the number of set associated predictor bits; the probability \hat{d} equals j for $1 \leq j \leq d$ is, within tables containing m records,

$$Prob\{\hat{d} = j\} = \binom{d-1}{j-1} p_m^{j-1} (1 - p_m)^{d-j} \quad (2)$$

and the expected number of set predictor bits is $1 + (d-1)p_m$.

Similarly, during an unsuccessful search operation, the probability \hat{d} equals j for $0 \leq j \leq d$ is, within tables containing m records,

$$Prob\{\hat{d} = j\} = \binom{d}{j} p_m^j (1 - p_m)^{d-j}. \quad (3)$$

and the expected number of set predictor bits is $d p_m$.

Both the successful and unsuccessful search lengths are calculated using the probability a bucket is filled together with the expected number of set associated predictor bits. Tables with single record bucket capacity have been analyzed [7, 12]; the analysis appears within data structure and algorithm textbooks circa 2005. This analysis utilizes a sampling without replacement approach to calculate the probability an insertion requires exactly r probes to locate the desired bucket in a table containing m records. Evidently extending this approach to larger buckets is difficult. Tables also have been analyzed using sampling with replacement; this approach readily extends to larger bucket capacities [8, 10] and will be utilized here.

We continue by calculating *filled*(α) the probability an arbitrary, capacity b bucket contains b records within a table with loading factor α . Both the expected successful and unsuccessful search lengths will utilize this probability. The occupancy count of the first un-filled bucket encountered on a probe sequence will be arbitrary. The likelihood a probe sequence accesses a bucket containing i records depends only on the number of such buckets. This insight is utilized to calculate the probability a bucket is filled.

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 [13], obtain the $b + 1$ expected values as a function of m and n

$$\begin{aligned} N_0(m+1) &= N_0(m) - N_0(m)/(n - N_2(m)) \\ N_1(m+1) &= N_1(m) + (N_0(m) - N_1(m))/(n - N_2(m)) \\ N_2(m+1) &= N_2(m) + N_1(m)/(n - N_2(m)) \end{aligned} \quad (4)$$

with $0 \leq m < 2n$ with initial conditions $N_i(0) = n \delta_{i,0}$. Of course, when the buckets are filled completely with $m = 2n$ records, the final values are $N_i(2n) = n \delta_{i,2}$.

Equations (4) are rationalized as follows. Suppose m records have been inserted, during insertion of the $m + 1^{st}$ record the number of buckets containing one record can change by at most one. The probability of these events will vary as the number of unfilled buckets varies; the expression $(n - N_2(m))$ designates the number of unfilled buckets after m insertions. The probability the number of buckets containing one record increases by one is $N_0(m)/(n - N_2(m))$. The probability the number of buckets containing one record decreases by one is $N_1(m)/(n - N_2(m))$. Finally, the probability the number containing one record remains the same is zero since $N_1(m)$ must either increase or decrease with each insertion. Then using the total expectation rule,

$$N_1(m+1) = N_1(m) + (N_0(m) - N_1(m))/(n - N_2(m))$$

This is the principal equation within (4).

The empty bucket count as m increases is a special case

$$N_0(m+1) = N_0(m) - N_0(m)/(n - N_2(m))$$

in which it is impossible for the count to increase. Similarly, the full bucket count as m increases is a special case

$$N_2(m+1) = N_2(m) + N_1(m)/(n - N_2(m))$$

in which it is impossible for the count to decrease.

In general, a bucket is filled with probability

$$filled(\alpha) = N(m, 2)/n \quad (5)$$

where $\alpha = m/(2n)$ is the loading factor. Figure 4 presents the probability values calculated using equations 4; these recurrence equations evidently do not have an elementary closed form solution. However, a useful approximation exists; a bucket is filled with probability approximately

$$filled(\alpha) \approx \alpha^{5/3} \quad (6)$$

which is also presented within figure 4. The largest difference between any probability and its approximation values is less than 0.0046 as the table size increases.

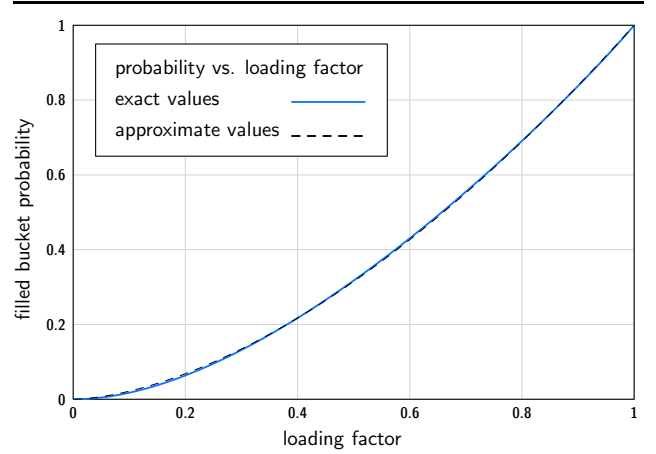


Figure 4. The filled bucket probability versus loading factor.

3.1. Successful Search Length

A successful search is conducted for a record residing within the table; the successful search length L_S measures the total number of buckets examined to conduct the search. First we calculate the expected average of the min-sequence length $E[L_S^{min}]$ for a table. Then we can calculate the expected successful search length $E[L_S]$. Since records are not moved within the table once they are inserted, we anticipate

$$E[L_S] \geq E[L_S^{min}].$$

Moreover, when d is one, we note $E[L_S]$ is $E[L_S^{min}]$.

Suppose the table with n buckets has loading factor α ; the probability a bucket is filled $filled(\alpha)$ has been calculated above. The probability an insertion, using d hash functions, will require at least $\lambda + 1$ probes is $filled(\alpha)^{d\lambda}$. That is, each of the d probe sequences must hit a filled bucket λ times. Finally, when a table contains m records, the probability that L_S^{min} is greater than λ is

$$Prob\{L_S^{min} > \lambda\} = \frac{1}{m} \sum_{j=1}^m filled(j/(2n))^{d\lambda}.$$

Thus we have

$$E[L_S^{min}] = \frac{1}{m} \sum_{\lambda \geq 0} \sum_{j=1}^m filled(j/(2n))^{d\lambda}. \quad (7)$$

The successful search is conducted by interleaving \hat{d} probe sequences where $\hat{d} \leq d$ is the number of set predictor bits associated with the search. One of these probe sequences will hit the desired record during its L_S^{min} probe on average. Within the search process, each of the last \hat{d}

probes will access the desired record with equal probability. Accordingly, the successful search length L_S is given by

$$L_S = \begin{cases} L_S^{min}, & \text{if } \hat{d} \text{ is } 1; \\ 2(L_S^{min} - 1) + \frac{3}{2}, & \text{if } \hat{d} \text{ is } 2; \\ 3(L_S^{min} - 1) + 2, & \text{if } \hat{d} \text{ is } 3; \\ \vdots \\ d(L_S^{min} - 1) + \frac{d+1}{2}, & \text{if } \hat{d} \text{ is } d. \end{cases} \quad (8)$$

Theorem: For tables configured with n buckets of size two containing m records using d hash functions, the expected successful search length $E[L_S]$ is

$$E[L_S^{min}] ((d-1)p_m + 1) - \frac{(d-1)}{2} p_m. \quad (9)$$

The theorem follows by combining equations 2 and 7 via the total expectation rule.

3.2. Unsuccessful Search Length

An unsuccessful search is conducted for a record not residing within the table; the unsuccessful search length L_U measures the total number of buckets examined to conduct the search. First we calculate the expected number of buckets accessed $E[L_U^1]$ by a single probe sequence to first touch an unfilled bucket and then calculate the expected unsuccessful search length $E[L_U]$.

For unsuccessful searches, the probability a single probe sequence accesses i buckets to first touch an unfilled bucket follows the geometric distribution. For a single hash function

$$Prob\{L_U^1 > \lambda\} = filled(\alpha)^\lambda$$

from which it follows

$$E[L_U^1] = \frac{1}{1 - filled(\alpha)}.$$

This formula is, of course, a direct extension of the result for ordinary double hashing [7, 8, 11, 12] where the expected value is $1/(1 - \alpha)$.

The unsuccessful search length $E[L_U]$ is given by

$$E[\hat{d} L_U^1] \quad (10)$$

since all of the \hat{d} probe sequences must be followed until each locates an unfilled bucket.

Theorem: For tables configured with n buckets of size two containing m records using d hash functions, the expected unsuccessful search length $E[L_U]$ is

$$\frac{d p_m}{1 - filled(m/(2n))}. \quad (11)$$

The theorem follows by combining equations 3 and 9 via the total expectation rule.

Both the successful and unsuccessful search length analyses follow the sampling with replacement paradigm. Since probe sequences are permutations, the analysis would be more accurately served using sampling without replacement; however, our expected value models provide very usable results as shown in the experimental results section.

3.3. Asymptotic Choice

Several configurations are considered to demonstrate the significance of the approach. Since there are several parameters, the number of hash functions d and the number of predictor bits s , we determine a parameter space partitioning useful for presenting similar results. The ratio

$$\xi = \frac{s}{2nd}$$

in which n represents the number of buckets within the table provides such a partitioning.

Theorem: For tables configured with n buckets of size two and s predictor bits containing m records let $n\xi = s/2d$ remain constant while d and s increase. The asymptotic search lengths are

$$\begin{aligned} E[L_S] &= 1 + \frac{\alpha}{2\xi} \\ E[L_U] &= \frac{\alpha}{\xi(1 - filled(\alpha))} \end{aligned} \quad (12)$$

where α is the loading factor $m/(2n)$.

The limiting $E[L_S]$ is similar to the separate chaining [7] successful search length. Moreover the expected search lengths can be arbitrarily close to one by increasing ξ .

The $E[L_S]$ formula comes from expression (9). Increasing d improves the chances of obtaining a shorter min-probe sequence,

$$\lim_{d \rightarrow \infty} E[L_S^{min}] = 1$$

since the shortest possible min-probe sequence has length one. Then $(d-1)(1 - (1 - 1/s)^m) = (d-1)p_m$ and $\xi = s/(2nd)$ we have

$$\begin{aligned} \lim_{d \rightarrow \infty} (d-1)p_m &= \lim_{d \rightarrow \infty} (d-1) \left(1 - \left(1 - \frac{1}{2nd\xi}\right)^m\right) \\ &= m/(2n\xi) = \alpha/\xi. \end{aligned}$$

Combining the two limit expressions in formula (9) yields formula (11).

The $E[L_U]$ expression has a similar development since $filled(\alpha)$ does not vary with s .

4. Experimental Paradigm and Results

The experimental effort provides average successful and unsuccessful search lengths for comparison with our analytical results as well as individual table-method timings. External double hashing with choice is implemented in C; the code compiled via the gcc compiler with -O3 optimizations. Timing was done using the gettimeofday function which “ticks” once per microsecond. The computational environment consists of a one GHz Pentium III processor configured as a server. For our experiments, we utilized tables configured with 257 buckets; records were generated via the Linux random function producing long unsigned values with a period of approximately $16(2^{31} - 1)$. The approximately uniformly distributed values provide a challenging workload without locality; the analysis also assumes such a workload. Reference locality will provide improvements depending upon the degree of locality; locality is not considered further here.

For load factors from zero to one, the successful and unsuccessful search length averages were calculated as well as the associated timings determined. An average value is determined experimentally together with its 95% confidence interval of width 1% of the to-be-determined average. Since the data sample standard deviation is unknown and will vary with the load factor, the two-stage approach of Stein [14] is utilized to determine the number of samples sufficing to obtain the confidence interval of desired width. For a given load factor, Stein’s approach involves obtaining a fixed number of samples (40 in our experiments) to determine the sample mean and standard deviation. The number of additional samples required, for the desired 95% confidence interval, is the larger of zero and

$$\left\lceil \left(\frac{\text{sample standard deviation} * 2.021}{0.005 * \text{sample mean}} \right)^2 \right\rceil - 40.$$

This process is followed for the successful and unsuccessful search lengths as well as the successful and unsuccessful search timings; the largest of the four numbers is used to complete the experiment for the load factor. There is no control on the final number of samples required especially when the sample standard deviation is large; in our experiments with 257 buckets per table, the number ranged between 40 and a few thousand per data point.

Both experimental and analytical results regarding successful and unsuccessful search lengths are presented graphically. Figure 5 presents the successful and unsuccessful search length experimental as well as expected values for $\xi = s/514d = 1$. The 95% confidence interval has width $\frac{1}{100} * \text{average search length}$; the confidence intervals, while not shown, are easily determined.

The successful search curve for $d = 1$ is best for very small loading factors and the worst for very large load-

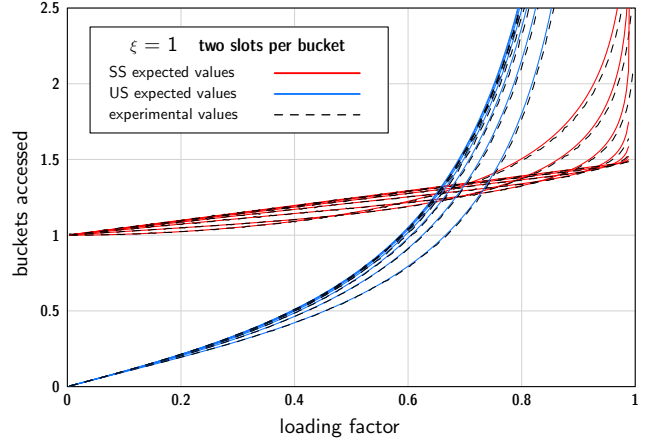


Figure 5. Successful and unsuccessful search lengths for $\xi = s/514d = 1$ with $d = 1, 2, 4, 8, 16, 32$ and 64

ing factors; the successful search curve for $d = 64$ is the flattest within our experiments. As d increases, the unsuccessful search curves cluster approaching $\alpha / (\xi(1 - \text{filled}(\alpha)))$; the unsuccessful search length increases with increasing d . These remarks are valid for all the search length figures 5 through 11.

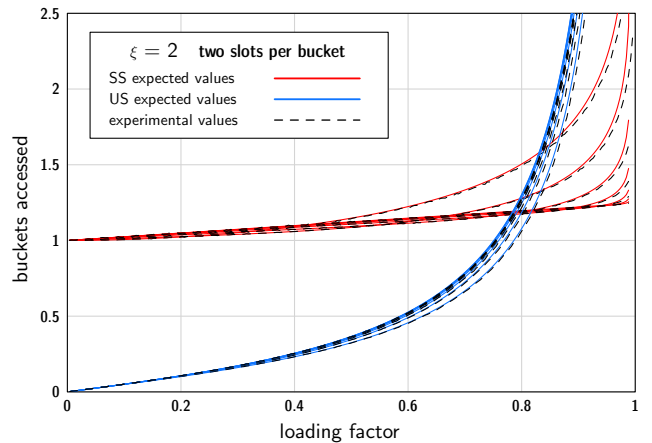


Figure 6. Successful and unsuccessful search lengths for $\xi = s/514d = 2$ with $d = 1, 2, 4, 8, 16, 32$ and 64

Figure 6 presents the successful and unsuccessful search lengths for $\xi = 2$; comparing with the figure 5 values, both the unsuccessful search and the successful search lengths have decreased. Moreover as the number of hash functions increases, the search lengths will more closely approximate the values from expression (11) demonstrated in figures 8 through 11 as well.

Access calculation timing is an issue when using more than one hash function; figure 7 show the increase in “internal” calculation time as the number of hash functions increases. The insertion time is estimated using $dE[L_U^1]$ disk accesses plus the internal time.

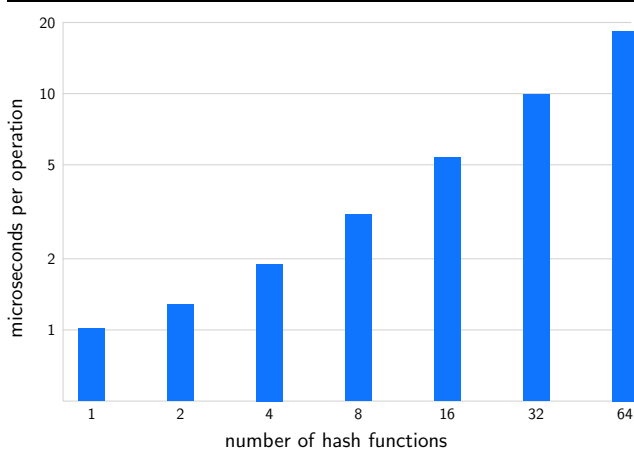


Figure 7. Successful and unsuccessful search average internal timings

Table 1 presents some “external” timings using formulae 7, 9, and 11 as follows in which the expected external latency is 4900 microseconds per disk access.

$$\begin{aligned}
 IN_{time} &= d * E[L_U^1] * 4900 + \text{internal time} \\
 SS_{time} &= E[L_S] * 4900 + \text{internal time} \\
 US_{time} &= E[L_U] * 4900 + \text{internal time.}
 \end{aligned}$$

The figures clearly show for as few as eight hash functions which require approximately 3 microseconds to evaluate,

d	ξ	SS	US	IN
1	2	7351	5244	15926
	4	7351	2941	15926
	8	7351	1520	15926
2	2	6273	5832	31851
	4	5783	6127	31851
	8	5783	1471	31851
8	2	5734	6275	127403
	4	5734	3188	127403
	8	5099	1620	127403
64	2	5851	6341	1019220
	4	5361	3206	1019220
	8	5117	1638	1019220

Table 1. Expected access times in microseconds; $\alpha = 0.8$ and the average disk drive seek time is 4900 microseconds.

the results are very close to what could be obtained using an “unlimited” number of hash functions; evaluation of the

hash functions will dominate the access time as d increases without bound.

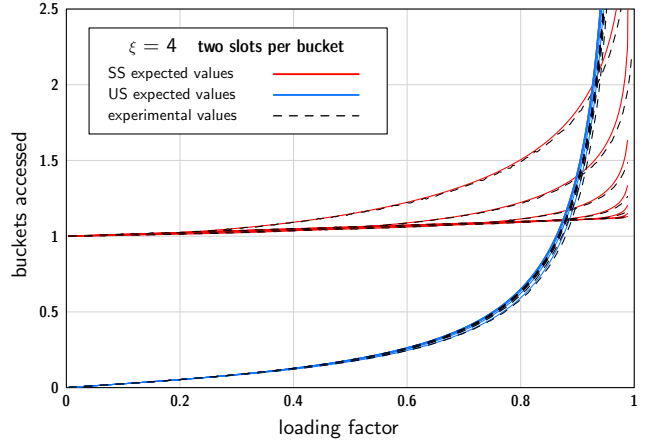


Figure 8. Successful and unsuccessful search lengths for $\xi = 4$ with $d = 1, 2, 4, 8, 16, 32$ and 64

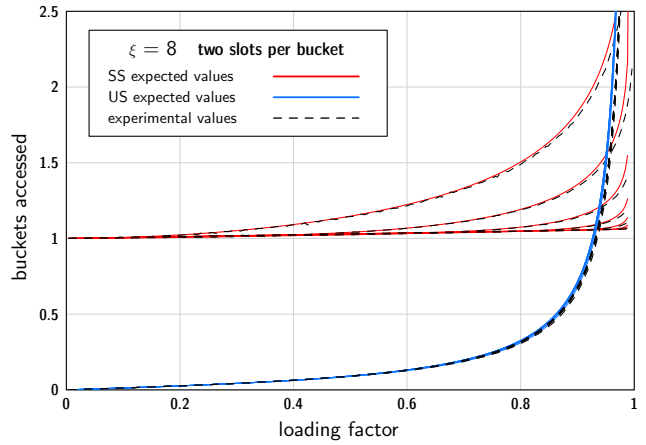


Figure 9. Successful and unsuccessful search lengths for $\xi = 8$ with $d = 1, 2, 4, 8, 16, 32$ and 64

5. Conclusions

External double hashing with choice provides a very convenient extension to ordinary double hashing providing improved successful and unsuccessful search lengths. Within external storage environments, the extra space required of predictor bits is *extremely* modest and realistically could be maintained within cache memory. For example, with two hash functions, the successful search length diminishes for $\alpha = 0.8$ from approximately 1.27 when $\xi = 2$ to 1.18 when $\xi = 8$ as noted in figures 6 and 9; ordinary external double hashing, with one hash function, obtains suc-

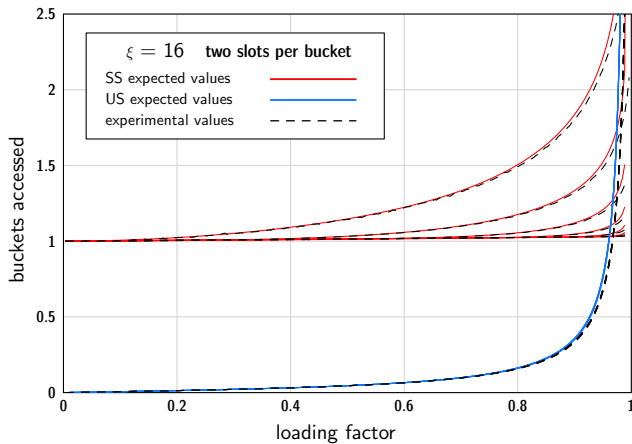


Figure 10. Successful and unsuccessful search lengths for $\xi = 16$ with $d = 1, 2, 4, 8, 16, 32$ and 64

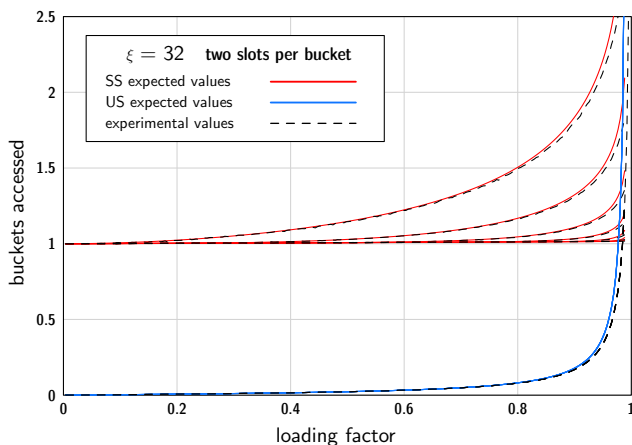


Figure 11. Successful and unsuccessful search lengths for $\xi = 32$ with $d = 1, 2, 4, 8, 16, 32$ and 64

successful search length approximately 1.5. The unsuccessful search length for $\alpha = 0.8$ diminishes from approximately 1.19 to 0.32 as well.

It will be interesting to combine choice, passbits as well as expand the table configuration to include larger bucket capacities; passbits improve the unsuccessful search length [10]. The use of larger buckets will improve both the successful and unsuccessful search lengths; calculation of the *filled* probability as a function of bucket capacity via recurrence equations similar to (4) is possible. Moreover, these curves do not seem to have elementary closed form expressions useful for analytic design; however, useful approxi-

mations seem to exist.

References

- [1] Y. Azar, A. Broder, A. Karlin, and E. Upfal. Balanced allocations. In *Proceedings of the ACM Symposium on Theory of Computing (STOC 94)*, pages 593–602, Montreal, Quebec, 1994.
- [2] I. F. Blake and A. G. Konheim. Big buckets are (are not) better. *Journal of the Association of Computing Machinery*, 24(4):591–606, 1977.
- [3] A. Broder and M. Mitzenmacher. Using multiple hash functions to improve IP lookups. In *Proceedings of the 20th IEEE Computer and Communications Conference (INFOCOM 01)*, number 3, pages 1454–1463, 2001.
- [4] J. Carter and M. Wegman. Universal classes of hash functions. *Journal of Computer and System Sciences*, 18(2):143–154, 1979.
- [5] W. Feller. *An Introduction to Probability Theory and Its Applications*, volume I. John Wiley, 1968.
- [6] L. Guibas and E. Szemerédi. 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. Martini and W. 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] W. Peterson. Addressing for random-access storage. *IBM Journal of Research and Development*, 1(2):130–146, 1957.
- [13] A. Rényi. *Probability Theory*. North-Holland Publishers, 1970.
- [14] C. Stein. A two sample test for a linear hypothesis whose power is independent of the variance. *Annals of Mathematical Statistics*, 16(3):243–258, 1945.
- [15] J. Ullman. A note on the efficiency of hash functions. *Journal of the Association for Computing Machinery*, 19(3):569–575, July 1972.
- [16] B. Vöcking. How asymmetry helps load balancing. *Proceedings of the 40th IEEE Symposium on Foundations of Computer Science*, pages 131–141, 1999.
- [17] A. Yao. Uniform hashing is optimal. *Journal of the Association for Computing Machinery*, 32(3):687–693, July 1985.