

- [23] V. D. Milman and G. Schechtman. *Asymptotic Theory of Finite Dimensional Normed Spaces*. Number 1200 in Lecture Notes in Mathematics. Springer Verlag, New York, 1986.
- [24] B. K. Pittel. Linear probing the probable largest search time grows logarithmically with the number of records. *J. of Alg.*, 8(2):236–249, 1987.
- [25] B. K. Pittel and J.-H. Yu. On search times for early-insertion coalesced hashing. *SIAM J. Comput.*, 17(3):492–503, June 1988.
- [26] M. Sipser. A complexity theoretic approach to randomness. In *STOC '83*, pages 330–335, Apr. 1983.
- [27] L. J. Stockmeyer. The complexity of approximate counting. In *STOC '83*, pages 118–126, Apr. 1983.
- [28] R. E. Tarjan and A. C. Yao. Storing a sparse table. *Commun. ACM*, 21:606–611, Nov. 1979.
- [29] J. S. Vitter. *Analysis of Coalesced Hashing*. PhD thesis, Stanford University, Stanford, CA, Aug. 1982. Technical Report STAN-CS-80-817.
- [30] A. C. Yao. Should tables be sorted? *J. ACM*, 28(3):615–628, July 1981.
- [31] A. C. Yao. Uniform hashing is optimal. *J. ACM*, 32(3):687–693, 1985.

- [8] M. Dietzfelbinger, A. R. Karlin, K. Mehlhorn, F. Meyer auf der Heide, H. Rohnert, and R. E. Tarjan. Dynamic perfect hashing: Upper and lower bounds. In *FOCS '88*, pages 524–531, 1988. Revised Version: Tech. Report 77, University of Paderborn, FB 17 Mathematik-Informatik, 1991; to appear in *SIAM J. Comput.*
- [9] M. Dietzfelbinger and F. Meyer auf der Heide. A new universal class of hash functions and dynamic hashing in real time. In *ICALP '90*, pages 6–19, 1990.
- [10] M. L. Fredman, J. Komlós, and E. Szemerédi. Storing a sparse table with $O(1)$ worst case access time. *J. ACM*, 31(3):538–544, July 1984.
- [11] J. Gil. *Lower Bounds and Algorithms for Hashing and Parallel Processing*. PhD thesis, The Hebrew University of Jerusalem, Givat Ram 91904, Jerusalem, Israel, Nov. 1990.
- [12] J. Gil and Y. Matias. Fast and efficient simulations among CRCW models. Manuscript, 1990.
- [13] J. Gil and Y. Matias. Fast hashing on a PRAM—designing by expectation. In *SODA '91*, pages 271–280, Jan. 1991.
- [14] J. Gil, F. Meyer auf der Heide, and A. Wigderson. Not all keys can be hashed in constant time. In *STOC '90*, pages 244–253, 1990.
- [15] T. Hagerup and C. Rüb. A guided tour of chernoff bounds. *Inf. Process. Lett.*, 33:305–308, 1989/90.
- [16] A. R. Karlin and E. Upfal. Parallel hashing—an efficient implementation of shared memory. In *STOC '86*, pages 160–168, May 1986.
- [17] G. D. Knott. Direct chaining with coalescing lists. *J. of Alg.*, 4(1):7–21, 1984.
- [18] D. E. Knuth. *Sorting and Searching*, volume 3 of *The Art of Computer Programming*. Addison-Wesley, Reading, MA, 1973.
- [19] D. E. Knuth. Computer science and its relationship to mathematics. *Am. Math. Monthly*, 8:323–343, 1974.
- [20] Y. Matias and U. Vishkin. Converting high probability into nearly-constant time— with applications to parallel hashing. In *STOC '91*, pages 307–316, May 1991. Also in UMIACS-TR-91-65, Institute for Advanced Computer Studies, Univ. of Maryland, April 1991.
- [21] B. Maurey. Construction de suites symétriques. *Comptes Rendu Academie des Sciences Paris*, 288:679–681, 1979.
- [22] K. Mehlhorn. *Data Structures and Algorithms*. Springer-Verlag, Berlin Heidelberg, 1984.

6 Concluding Remarks

The hashing model proposed leads to an alternate view of hashing algorithms as an element distinctness proof generator. In our lower bound analysis we assumed that all the hash functions (the proof components) are completely random. However, it is not difficult to see that if the universe is not too large, say polynomial in the size of the input set, and if non-random functions can be used, then the lower bounds do not hold. (E.g., by an integer sorting algorithm.)

Is the random functions assumption essential? It was shown before [1, 31] that random functions are optimal in certain hashing situations. On the other hand, hashing algorithms that are based on open addressing or on the FKS scheme (such as the one described in [13]), as well as the upper bounds presented here do not assume the existence of random functions. We conjecture that the lower bound results hold for a super polynomial sized universe even if arbitrary functions are allowed.

Although the model proposed in this paper is very general, it does not cover all hashing algorithms (e.g., the one presented in [9]). It may be interesting to define more general models and to gain better understanding of these models as well.

Acknowledgments Fruitful comments made by Faith E. Fich are gratefully acknowledged.

References

- [1] M. Ajtai, J. Komlós, and E. Szemerédi. There is no fast single hashing function. *Inf. Process. Lett.*, 7:270–273, 1978.
- [2] N. Alon and J. H. Spencer. *The Probabilistic Method*. John Wiley & Sons, Inc., 1991.
- [3] D. Angluin and L. G. Valiant. Fast probabilistic algorithms for hamiltonian paths and matchings. *J. Comput. Syst. Sci.*, 18:155–193, 1979.
- [4] W. C. Chen and J. S. Vitter. Analysis of early-insertion standard coalesced hashing. *SIAM J. Comput.*, 12(4):667–676, 1983.
- [5] W. C. Chen and J. S. Vitter. Analysis of new variants of coalesced hashing. *ACM Trans. Database Syst.*, 9(4):616–645, 1984.
- [6] W. C. Chen and J. S. Vitter. Deletion algorithms for coalesced hashing. *The Comp. J.*, 29(5):436–450, 1986.
- [7] H. Chernoff. A measure of asymptotic efficiency for tests of a hypothesis based on the sum of observations. *Annals of Math. Statistics*, 23:493–507, 1952.

Phase IIa of `ChainShallow` deals therefore with sets of at least R elements; as soon as a subset is broken into smaller pieces, the Phase ceases to handle it. We need the hashing functions used to satisfy a “good breaking” property.

Definition 3 *Let \mathcal{H} be a class of hash functions mapping sets of size r into memory of size βr^2 . Let h be picked at random from \mathcal{H} . Then, \mathcal{H} is a good breaker iff*

$$\mathbf{Prob} \left(\forall i |h^{-1}| \leq R \right) \leq \beta^{1-R} .$$

This property is achieved by random functions [11, Chapter 2]. However, true random functions are not useful for hashing as they require huge space for representation and consequently non-constant evaluation time. The class of polynomial hash functions of degree R is a good breaker as well. Its members can be represented efficiently. Unfortunately, each application of a hash function of this class requires the order of R steps, which amounts to a total of $O(\lg \lg n)$ time, (although the number of hash function applications is still $O(\lg \lg n / \lg \lg \lg n)$).

Dietzfelbinger and Meyer auf der Heide [9] gave a construction of a good breaker class \mathcal{R} such that a function $h \in \mathcal{R}$ can be evaluated in constant time. This class \mathcal{R} is best suited for a more practical implementation of `ChainShallow` as it offers the advantages of sub-linear representation and constant time evaluation.

The sequence $\{\beta_t\}$ is defined in Algorithm `ChainShallow` by

$$\beta_{t+1} = \beta_t^R / 4 \qquad \beta_1 = O(1) . \tag{6}$$

As in `RetryShallow`, the basic unit of memory allocation is $\beta_t 2^{2^j}$ in an iteration t of `ChainShallowj`. A subset of size $r \leq 2^j$ belonging to this procedure is hashed using a good breaker into a $\beta_t r 2^j$ size memory block. The probability that the subset will not be broken into small enough subsets is at most β_t^{1-R} . The rounds of an iteration carry on until $N(r) \leq 2N_t(r)\beta_t^{1-R}$ then the expected number of rounds in an iteration will be ≤ 2 here too. Thus we have

$$N_{t+1}(r) \leq 2N_t(r)\beta_t^{1-R} . \tag{7}$$

Combining the recurrences (6) and (7) we see that memory requirements decrease geometrically:

$$N_{t+1}(r)\beta_{t+1} \leq N_t(r)\beta_t / 2 .$$

Adding to this the fact that the root node function must have satisfied Inequality (3), we infer that the total memory usage is linear.

Solving the recurrence (6) we have

$$\beta_t = \beta_1^{R^{t-1}} 4^{-(R^t - 1)/(R - 1)} ,$$

from which it follows that the number of iterations is $O(R)$, completing the proof of Theorem 6.

5.4 The Parallel Variant: Algorithm ParShallow

Both `RetryShallow` and `BasicShallow` can be implemented on a parallel machine where each processor is initially assigned a key and keys do not move among processors [12]. The `ParShallow` algorithm described below achieves $O(\lg^* n)$ depth in the parallel model by applying several hash functions to a key in each tree node. In a parallel setting, this corresponds to allowing multiple processors to hash the same key. In each successive iteration, more and more processors are drafted to hash fewer and fewer keys. The bookkeeping required in each iteration for identifying the active keys and assigning processors to them is not trivial. A PRAM algorithm running in $O(\lg^* n)$ time was first described by Matias and Vishkin [20]. Their algorithm is based in part on `ParShallow` which we describe next. We use the description and the analysis of `RetryShallow` as a skeleton.

The following differences apply: the sequence β_t starts with $\beta_1 = 4$ and increases at a quicker rate, $\beta_t = 2^{\beta_t - 2}$. In iteration t of `ParShallow`, the $\beta_t r 2^j$ size memory block allocated to a bucket of $r \leq 2^j$ keys is further divided into $\beta_t 2^j / 2r$ sub-blocks of $2r^2$ cells each. The parallel hashings are then done into these sub-blocks. The failure probability is thus at most

$$2^{-\beta_t r / 2r'} = 2^{-(r/r' - 1)\beta_t / 2} \cdot 2^{-\beta_t / 2} \leq 2^{-2(r/r' - 1)} \cdot 2^{-\beta_t / 2} < \frac{r'}{r} \cdot 2^{-\beta_t / 2} .$$

In an iteration, rounds continue until the number of active keys is at most $2 \cdot 2^{-\beta_t / 2} N_t(r)$, and hence

$$N_{t+1}(r) \leq 2 \cdot 2^{-\beta_t / 2} N_t(r) .$$

This last bound, together with the definition of the sequence $\{\beta_t\}$, proves that $\overline{\text{DEPTH}}(\text{ParShallow}) = O(\lg^* n)$. To see that $\overline{\text{SPACE}}(\text{ParShallow}) = \text{SPACE}_r(\text{ParShallow}) = O(n)$ note that the total memory used by a round of iteration $t + 1$ is at most $1/2$ of the total memory used by a round of iteration t . This completes the proof of the upper bound part of Theorem 9.

5.5 The Chaining Variant: Algorithm ChainShallow

The reduction in `DEPTH` to $O(\lg \lg n / \lg \lg \lg n)$ is achieved by replacing Phase II of `RetryShallow` by two phases:

Phase IIa Use the prototype of algorithm `RetryShallow` to continuously break subsets until they are all small enough instead of attempting to achieve a complete separation. In particular, hashing is conducted until all subsets have $\text{MaxLevel} = \lg \lg n / \lg \lg \lg n$ or fewer keys.

Phase IIb When all subsets are smaller than R , then in each tree level hashing is into a range of size 1. Recall the chaining model variant allows for storing a key in every internal node. At most R levels will thus be added to the tree regardless of the hash function employed.

of S into buckets S_1, S_2, \dots, S_n (some empty) such that Inequality (3) holds. While this goal is not achieved the algorithm iterates as follows: Each non-empty bucket S_i is hashed using a function selected at random from $\mathcal{H}^1(|S_i|)$. Since the total memory used for all buckets is n , this hashing forms a refined partition of S into n buckets. Let S_1, S_2, \dots, S_n denote now these newly created subsets of S . If Inequality (3) is attained then the phase ends; otherwise the algorithm iterates again hashing each non-empty subset into a range equal to its size using a linear hash function.

Phase II Similarly to Algorithm `RetryShallow`, Algorithm `BasicShallow` splits into procedures `BasicShallow`₁, \dots , `BasicShallow` _{$\lceil \lg n \rceil$} . Consider a bucket S_i formed in Phase I, $2^{j-1} < |S_i| \leq 2^j$, then S_i and all sub-buckets formed from it during Phase II are handled by procedure `BasicShallow` _{j} . Bucket and keys do not move between procedures.

Attempts of `RetryShallow` are translated to refining *rounds* of `BasicShallow`. Let $N_{t,j}$ be the number of keys in active buckets at the beginning of iteration $t \geq 1$ of the procedure `BasicShallow` _{j} . Similarly to `RetryShallow`, the iteration ends when its rounds achieve that the number of keys in its active buckets is no greater than $2N_{t,j}/\beta_t$, where β_t is as before. In a round a bucket of size r is hashed into a memory block of size $\beta_t r 2^j$ using a function selected at random from $\mathcal{H}^1(\beta_t r 2^j)$. If this function is injective then the bucket becomes inactive. Otherwise, the bucket is split into smaller sub-buckets and processing continues for all sub-buckets which have two or more keys.

Analysis It follows from the linearity of the expectation that Fact 3 holds also for each refining iteration of Phase I. The expected number of iterations in this phase is therefore $O(1)$. The contribution of the phase to the expected depth is constant and its contribution to the expected memory usage is $O(n)$. The analysis of Phase II follows.

Consider a round of iteration t of `BasicShallow` _{j} . The probability for a bucket of size r , active at the beginning of the round, to remain active after the round is at most $r/(2^j \beta_t) < 1/\beta_t$. As before, the expected number of active keys by the end of the round is at most $N_{t,j}/\beta_t$; the probability that the round will fail (i.e., that it will not terminate the iteration) is at most $1/2$; the expected number of rounds in an iteration is therefore at most 2. Following the same lines of the analysis of `RetryShallow` we have $\overline{\text{DEPTH}}(\text{BasicShallow}, n) = O(\lg \lg n)$ and therefore $\overline{\text{SEARCH}}(\text{BasicShallow}, n) = O(\lg \lg n)$.

The memory allocation scheme is such that memory allocated to all sub-buckets formed from a certain bucket is never greater than what would have been allocated to the bucket had it not been split. Memory cannot be re-used in the basic model, so even non-successful rounds contribute to the total memory usage. However, we have that the memory used in rounds of the same iteration is essentially the same and that the expected number of rounds in an iteration is $O(1)$. It follows that the expected memory used in an iteration of `BasicShallow` is no more than a constant times the memory used of the same iteration of `RetryShallow`. This completes the proof of Theorem 4.

probability of any given attempt to be successful is at least $1/2$. The expected number of attempts in an iteration is therefore at most 2 and the total expected number of attempts in any single procedure is $O(\lg \lg n)$.

Attempt failures are independent. We can therefore apply Chernoff bounds obtaining that there is a constant C , such that for any $\gamma > C$, the probability that the total number of attempts will be more than γ times its expected value is $o(1/\lg^{\gamma/C} n)$ in any given procedure. Since there are at most $\lceil \lg n \rceil$ procedures executing simultaneously, the expected parallel time until the slowest procedure terminates is also $O(\lg \lg n)$. We therefore have $\overline{\text{DEPTH}}(\text{RetryShallow}, n) = O(\lg \lg n)$.

Analysis of $\text{SPACE}(\text{RetryShallow}, n)$ As noted in the description of FKS, the contribution of Phase I to $\text{SPACE}(\text{RetryShallow}, n)$ is $O(n)$. Consider any procedure RetryShallow_j . In iteration t there are at most $N_{t,j}/2^j$ active buckets, each such bucket is hashed into a memory block of size $\beta_t 2^{2j}$. The total memory used in the iteration is therefore bounded above by $N_{t,j} \beta_t 2^j$. (Recall that no space is charged for retry nodes.) The memory used in the following iteration is at most half of that

$$N_{t+1,j} \beta_{t+1} 2^j \leq \frac{2N_{t,j}}{\beta_t} \cdot \frac{\beta_t^2}{4} 2^j = N_{t,j} \beta_t 2^j / 2 .$$

Thus, the total memory usage is, up to a constant factor, the same as memory used in iteration 1. Since initially $\mathbf{E}(\sum |S_i|) = n$ and $\beta_1 = O(1)$ we get that even accounting for the possible doubling of set sizes due to rounding, $\text{SPACE}(\text{RetryShallow}, n) = O(n)$.

Analysis of $\overline{\text{TIME}}(\text{RetryShallow}, n)$ Note that the expected number of times RetryShallow accesses any memory cell is constant. Then, $\overline{\text{TIME}}(\text{RetryShallow}, n) = O(n)$ follows from $\text{SPACE}(\text{RetryShallow}, n) = O(n)$. Since all functions used in RetryShallow are polynomials of degree 1, we get that the number of operations is linear even if arithmetic operations are counted and word size is limited to $O(\lg |U|)$.

Analysis of $\text{SEARCH}(\text{RetryShallow}, n)$ RetryShallow constructs a 2-level hash table for S . The first level consists of the function selected at phase I, which splits S to buckets. The second level consists of the injective hash functions found in phase II for each of the buckets. We therefore have $\text{SEARCH}(\text{RetryShallow}, n) = 2$. This completes the proof of Theorem 3.

5.3 The Basic Model: Algorithm BasicShallow

Algorithm BasicShallow which works in the basic model is derived from RetryShallow by replacing *retry* nodes in RetryShallow by *refining* nodes. A description of the modifications to the algorithm and to the analysis follows.

Phase I The algorithm creates an initial partition by selecting a hash function uniformly and at random from the class $\mathcal{H}^1(n)$. The goal of this phase is to achieve a partitioning

5.2 The Retries Variant: Algorithm RetryShallow

Algorithm **RetryShallow** is modeled after FKS. Phase I is identical and is concluded by finding a level-1 function h satisfying Inequality (3). The main change in Phase II is that **RetryShallow** finds an injective hash function for all buckets by trying memory blocks of rapidly growing size and not of constant size as in the FKS algorithm. The block size growth rate is characterized by the sequence $\{\beta_{t+1}\}$

$$\beta_1 = 16 \qquad \beta_{t+1} = \beta_t^2/4 \ , \qquad (4)$$

or, in an explicit form

$$\beta_t = 2^{2^t - 2} \ . \qquad (5)$$

Algorithm **RetryShallow** executes procedures $\text{RetryShallow}_1, \dots, \text{RetryShallow}_{\lceil \lg n \rceil}$ simultaneously. A procedure RetryShallow_j , $j = 1, \dots, \lceil \lg n \rceil$ handles buckets S_i for which

$$2^{j-1} < |S_i| \leq 2^j \ .$$

Initially, all these buckets are active. The procedure is a series of iterations in which buckets are inactivated by finding an injective hash function for them. Let $N_{t,j}$ be the number of keys in buckets belonging to procedure j which are active at the beginning of iteration t . Iteration t is a series of *attempts* to reduce $N_{t,j}$ by a factor of $\beta_t/2$. In each such attempt, each of the active buckets is hashed into a memory block of size $\beta_t 2^{2j}$ using a hash function selected at random from the class $\mathcal{H}^1(\beta_t 2^{2j})$. If the attempt fails to reduce the number of keys in active buckets to $2N_{t,j}/\beta_t$ then a “retry” is done in all nodes corresponding to active buckets; all separations obtained are disregarded. Otherwise, buckets for which an injective hash function was found become inactive; non-injective hash functions are disregarded (a retry) and the procedure carries on to iteration $t + 1$.

The procedure terminates when there are no more active buckets belonging to it. The algorithm terminates when all procedures terminate.

Analysis of $\overline{\text{DEPTH}}(\text{RetryShallow}, n)$ As noted in the description of FKS, the contribution of Phase I to $\overline{\text{DEPTH}}(\text{RetryShallow}, n)$ is at most constant.

If $\beta_t \geq 4n$, then by the end of iteration t the number of keys in active buckets is

$$2N(t, j)/\beta_t \leq 2n/4n = 1/2 \ ,$$

i.e., $Nt + 1, j = 0$. It follows from (5) that setting $t = O(\lg \lg n)$ suffices to ensure $\beta_t \geq 4n$. Hence, for all j the number of iterations of RetryShallow_j is $O(\lg \lg n)$.

It follows from Fact 4 that a function selected at random from $\mathcal{H}^1(\beta_t 2^{2j})$ is not injective on a bucket of size at most 2^j with probability at most $1/\beta_t$. Hence, the expected number of active keys by the end of an attempt is at most $N(t, j)/\beta_t$. By Markov’s inequality the

tions is

$$\mathcal{H}^d(m) := \left\{ h \mid h(x) := 1 + \left(\sum_{i=0}^d a_i x^i \bmod q \right) \bmod m, a_i \in U \right\} .$$

The hash functions used in both levels of FKS are drawn from the class of linear (i.e., polynomial of degree 1) hash functions. The main properties of the linear hash functions which enable the construction are given in the following facts proved in [10].

Fact 3 *Let S be fixed, and let h be chosen uniformly at random from the class $\mathcal{H}^1(|S|)$. Then,*

$$\mathbf{E} \left(\sum_{i=1}^m |S_i|^2 \right) \leq 2.5|S| ,$$

and consequently (by Markov's inequality)

$$\mathbf{Prob} \left(\sum_{i=1}^m |S_i|^2 \leq 5|S| \right) \geq 1/2 .$$

Fact 4 *Let S be fixed, and let h be chosen uniformly at random from the class $\mathcal{H}^1(m)$. Then,*

$$\mathbf{Prob}(h \text{ is not injective on } S) \leq |S|^2/m .$$

The algorithm works in two phases as follows:

Phase I For the given input set S , a level-1 hash function h is selected which satisfies

$$\sum_{i=1}^n |S_i^h|^2 < 5n . \tag{3}$$

This is done by repeatedly trying functions chosen at random from the class $\mathcal{H}^1(n)$. It follows from Fact 3 that the expected number of functions tried is constant. The memory used and the expected number of operations at this phase are therefore $O(n)$.

Phase II For each bucket S_i the algorithm allocates a range of size $2|S_i|^2$ and finds a level-2 function which injectively maps the bucket to this range. This function is constructed by repeatedly trying functions selected at random from the class $\mathcal{H}^1(2|S_i|^2)$. Each such function is non-injective with probability at most $1/2$ (Fact 4). The expected number of functions tried is therefore constant. The memory usage and the expected number of operations per bucket are therefore quadratic in the size of the bucket. Since the algorithm insisted on attaining Inequality (3) at Phase I, we have that the memory used and the expected number of operations at this phase are therefore $O(n)$ as well.

Viewing the algorithm within the tree model framework we see that the number of nodes ("active" buckets) drops only by approximately a factor of 2 from one level to the following one. This is the reason why the FKS algorithm has the order of $\lg n$ levels.

The equivalence of an algorithm without retries to an algorithm with retries was possible here because the **Opt** could use its total memory allowance in each and every iteration. In general, using this technique to transform **Alg**, an algorithm that uses retries into **Alg'**, an algorithm that avoids retries leads to an increase in the memory used by the algorithm by a factor of up to $\text{DEPTH}(\text{Alg}, n)$.

5 Proofs of Upper Bounds

As explained earlier, the main strategic decision made by a hashing algorithm is the allocation of memory to buckets. In the **FKS** algorithm, all hashing attempts (retries) of a bucket are into memory of fixed size. This scheme leads to $\text{DEPTH} = \Omega(\lg n)$. In order to further decrease the total number of hashing iterations a more flexible memory allocation must be employed.

The idealized algorithms **Opt** used the same memory size in all iterations. The upper bound algorithms in all model variants try to follow that scheme and use *almost* the same size of memory in every iteration. Thus, in every iteration the memory portion allocated to remaining buckets is increased.

A decreasing geometric series defines the partitioning of the total memory allowance between the iterations. Informally, we can say that although this series decreases quite rapidly, the decrease in the number of remaining buckets is so much quicker that the algorithm will find itself in conditions which are very similar to the **Extra Memory** assumption which **Opt** was permitted to make.

The rest of this section is outlined as follows: We begin by reviewing the **FKS** algorithm. Next, we describe how this algorithm is modified to form our main algorithm **RetryShallow** which works in the retries model. We proceed by presenting minimal changes to **RetryShallow** obtaining algorithm **BasicShallow** for the basic model. Algorithm **ParShallow** for the parallel model, and **ChainShallow** for the chaining model are presented next.

5.1 Foundation: Algorithm FKS

Algorithm **FKS** [10] takes an arbitrary set $S \subseteq U$ of size n as input and in $O(n)$ expected time generates a hash table for S . The resulting table uses $O(n)$ storage and supports $O(1)$ lookup time. The algorithm builds a *2-level* hash table: a *level-1* function splits S into subsets whose sizes are distributed in a favorable way. Then, an injective *level-2* hash function is built for each subset.

The **FKS** algorithm assumes that $U = \{0, 1, \dots, q - 1\}$ where q is prime. This assumption doesn't lead to any loss of generality and we will adhere to it henceforth.

Definition 2 (Polynomial Hash Functions) *The class of d -degree polynomial hash func-*

For $r_t \geq 4$ we have

$$\nu_{t+1} \leq 2^{r_t \nu_t} .$$

By setting $r_t = 4$, $m = O(n)$, we get that $\nu_0 = O(1)$. The number of iterations T required to decrease the number of subsets below $\lg n$ (i.e., until $\nu_T = n / \lg n$) is $\Omega(\lg^* n)$. This completes the proof of the lower bound part of Theorem 9.

The proof of (the chaining variant part of) Theorem 10 is conducted in a similar manner to the lower bound proof for the ordinary chaining variant. Let $r = r_1 = \lg^* n$, $r_{t+1} = r_t - 1$, $m = O(n)$. Then by Inequality (2)

$$\nu_0 = O\left((\lg^* n)^{\lg^* n}\right) .$$

We can also write

$$\nu_{t+1} \leq 2^{r \nu_t} \leq 2^{\nu_t^2} .$$

Now, the number of iterations required to achieve $\nu > \lg n$ is at least

$$\frac{\lg^* n - 1 - \lg^* \nu_0}{2} = \Omega(\lg^* n) .$$

4.4 The Retries Variants

To complete the lower bounds analysis we need to discuss the retries variant and provide the proofs for Theorems 2 items (a) and (c) of Theorem 7. Theorem 5 and Theorem 10 reference the retries variant as well. These references will be treated in a similar manner.

The retries technique is useful if a certain application of a hash function was not satisfactory according to some criteria. Then, instead of coping with it, the algorithm may try another hash function. However, our simplifications allow **Opt** a dichotomous classification of poll results. If there was a complete failure in a hash of a specific internal node, then doing a retry is the same as what **Opt** will do in the next iteration, but with less memory. On the other hand, if this internal node was not a complete failure, we allow **Opt** to ignore it without any further resource investment. If retry was done on such a node then this may only result in a deeper tree. Retries cannot help in the root node either, as the root node behavior is dominant (Inequality (2)), and even $O(\lg n)$ retries in the root node will not yield a significantly better value for $N_0(r)$.

Thus, the addition of retries to any model combination will not affect the **DEPTH** lower bounds. Upon further examination of the lower bound proof of the chaining variant, it can be seen that a parallel insertion time of $O(\lg \lg n / \lg \lg \lg n)$ cannot be achieved unless a key is left in $\Omega(\lg \lg n / \lg \lg \lg n)$ nodes, which will nullify the ability of the retries variant to achieve **SEARCH** = $O(1)$.

4.3 The Parallel Variant

The memory allocation scheme as used by **Opt** is slightly different here. Many hash functions can be applied in parallel to the same set. In an iteration t let $m_{i,1}, m_{i,2}, \dots$ be the cardinalities of ranges of those functions for some subset S_i , and let $m_i = m_{i,1} + m_{i,2} + \dots$ be the total range used for it. The probability that all those hash functions will be a complete failure is

$$\prod_j m_{i,j}^{1-r_t} .$$

This probability is minimized when $m_{i,j} = 2$ for all j . In this case the complete failure probability is

$$2^{m_i(1-r_t)/2} .$$

Note that if the set size is greater than 2 then a complete separation is not possible if only 2 cells are allocated to a set. This does not pose a problem in our lower bound analysis since we consider any partial separation to be a complete separation.

Let $m = m_1 + m_2 + \dots + m_{N_t}$ be a memory allocation of the m cells to the N_t sets. The expected value of N_{t+1} is

$$\mathbf{E}(N_{t+1}) = \sum_{i=1}^{N_t} 2^{m_i(1-r_t)} .$$

Once again, this is minimized when all the m_i are equal. Hence we have

Lemma 14 *In the parallel variant, all hash functions used by **Opt** are to a range of size 2. In iteration t , $m/2N_t$ functions with a total range of size m/N_t are applied to each one of the N_t sets of size r_t .*

From which we get a recursion formula for N_t

$$\mathbf{E}(N_{t+1}) = N_t 2^{m(1-r_t)/2N_t} .$$

Note that Lemma 12 also holds here, so we can write

$$N_{t+1} \geq \frac{N_t}{4} 2^{m(1-r_t)/2N_t}$$

which will take a simpler form using the definition $\nu_t = m/N_t$:

$$\nu_{t+1} = 4\nu_t 2^{(r_t-1)\nu_t/2} \leq 2^{(r_t-1)\nu_t/2 + \lg \nu_t + 2} .$$

- Setting $r = 2$ and $m = O(n)$ we have $-\lg \eta_0 = O(1)$ and hence

$$\begin{aligned}\overline{\text{DEPTH}}(\text{Opt}, n) &= \Omega\left(\frac{1}{\lg r} \lg \frac{\lg m}{2 - \lg \eta_0}\right) \\ &= \Omega\left(\lg \frac{\lg n + O(1)}{2 + O(1)}\right) \\ &= \Omega(\lg \lg n) .\end{aligned}$$

- Setting $r = 2$ and $m = n^{1+1/\lambda}$, λ fixed, we have $-\lg \eta_0 = \lg n/2\lambda + o(1)$ and hence

$$\begin{aligned}\overline{\text{DEPTH}}(\text{Opt}, n) &= \Omega\left(\frac{1}{\lg r} \lg \frac{\lg m}{2 - \lg \eta_0}\right) \\ &= \Omega\left(\lg \frac{(1 + 1/\lambda) \lg n}{2 + \lg n/2\lambda + o(1)}\right) \\ &= \Omega\left(\lg \frac{1 + 1/\lambda}{1/2\lambda + o(1)}\right) \\ &= \Omega(\lg \lambda) .\end{aligned}$$

- Setting $r = \lg n / \lg \lg n$ and $m = O(n)$ we have $-\lg \eta_0 = r \lg r + O(1)$ and hence

$$\begin{aligned}\overline{\text{DEPTH}}_r(\text{Opt}, n) &= \Omega\left(\frac{1}{\lg r} \lg \frac{\lg m}{2 - \lg \eta_0}\right) \\ &= \Omega\left(\frac{1}{\lg \lg \lg n} \lg \frac{\lg n + O(1)}{2 + r \lg r + O(1)}\right) \\ &= \Omega\left(\frac{\lg \lg n}{\lg \lg \lg n}\right) .\end{aligned}$$

- Setting $r = \lg \lambda / \lg \lg \lambda$, $m = n^{1+1/\lambda}$, λ fixed, we have $-\lg \eta_0 = -(r - 1)/\lambda \lg n + o(1)$ and hence

$$\begin{aligned}\overline{\text{DEPTH}}_r(\text{Opt}, n) &= \Omega\left(\frac{1}{\lg r} \lg \frac{\lg m}{2 - \lg \eta_0}\right) \\ &= \Omega\left(\frac{1}{\lg \lg \lambda} \lg \frac{(1 + 1/\lambda) \lg n}{2 + (r - 1) \lg n/\lambda}\right) \\ &= \Omega\left(\frac{1}{\lg \lg \lambda} \lg \frac{(1 + 1/\lambda)}{(r - 1)/\lambda}\right) \\ &= \Omega\left(\frac{1}{\lg \lg \lambda} \lg \frac{\lambda + 1}{r - 1}\right) \\ &= \Omega\left(\frac{\lg \lg \lambda}{\lg \lg \lg \lambda}\right) .\end{aligned}$$

Let $\eta_t = N_t/m$. Then, by dividing the above by m , we have

$$\eta_{t+1} = \eta_t^r/4 .$$

This representation demonstrates the fact that the fraction of sets of a given size decreases “only” double-exponentially, giving rise to the double logarithmic lower and upper bounds. The exact solution of the recurrence is given by

$$\eta_t = \eta_0^{r^t}/4^{(r^t-1)/(r-1)} .$$

For our purposes it is sufficient to write

$$\eta_t \leq \left(\frac{\eta_0}{4}\right)^{r^t}$$

which facilitates an easy counting of the minimal number of iterations:

Lemma 13 *If $m \leq n^3$, then the number of levels in Opt’s tree is*

$$\Omega\left(\frac{1}{\lg r} \lg \frac{\lg m}{2 - \lg \eta_0}\right) .$$

Proof. Let T be given by

$$T = \frac{1}{\lg r} \lg \frac{\lg \lg n - \lg m}{\lg \eta_0 - 2} .$$

Then, for $t < T$,

$$N_t = m\eta_t = m\left(\frac{\eta_0}{4}\right)^{r^t} > m\left(\frac{\eta_0}{4}\right)^{r^T} = \lg n .$$

It follows that if Opt executes less than T iterations it will have more than $\lg n$ sets and it cannot terminate. The proof is completed by noting that for $m \leq n^3$

$$T = \Omega\left(\frac{1}{\lg r} \lg \frac{\lg m}{2 - \lg \eta_0}\right) .$$

■

Applying this lemma to the estimates in Fact 2 will yield the proofs for the lower bounds set by Theorems 1, 5, Theorem 7(b) and Theorem 8. In particular,

Proof. Apply Inequality (1). ■

From the simplifying assumptions it follows that in iteration t , **Opt** uses m memory cells to deal with N_t sets of r_t keys each. The algorithm should allocate memory to cells in a way that will minimize the number of failures N_{t+1} . The following lemma reveals the memory allocation scheme used by **Opt**.

Lemma 11 *Opt uses a balanced memory allocation scheme; each of the N_t sets is hashed into m/N_t cells.*

Proof. In an iteration t , if a subset (that has r_t keys) is mapped by a random function into m_i memory cells, then the *complete failure* probability is $m_i^{1-r_t}$. This probability is inversely proportional to m_i , therefore a memory allocation is not optimal unless all the m cells are utilized. Let $m = m_1 + m_2 + \dots + m_{N_t}$ be a memory allocation of the m cells to the N_t sets. The expected value of N_{t+1} is given by

$$\mathbf{E}(N_{t+1}) = \sum_{i=1}^{N_t} m_i^{1-r_t}$$

and by convexity this is minimized when all m_i are equal. ■

The complete failure probability, $m_i^{1-r_t}$, increases as r_t decrease. Thus it is permissible to assume that **Opt** uses a complete failure probability derived from $r = r_1$, the initial size of the sets. We can then write

$$\mathbf{E}(N_{t+1}) = \sum_{i=1}^{N_t} m_i^{1-r}$$

and by Lemma 11

$$\mathbf{E}(N_{t+1}) = N_t \left(\frac{m}{N_t} \right)^{1-r} .$$

The probability that N_{t+1} will be much smaller than its expected value is estimated by

Lemma 12 *Let N_t be fixed. The event $N_{t+1} < \mathbf{E}(N_{t+1})/4$ is n -negligible if $\mathbf{E}(N_{t+1}) > \lg n$.*

Proof. Note that N_{t+1} is the sum of N_t independent random binary variables. The lemma is obtained from application of Chernoff [7] bounds. (See [3, 15] for a succinct statement of these bounds.) ■

Thus we can assume that $N_t \geq \mathbf{E}(N_t)/4$ simultaneously in all iterations. For simplicity we permit **Opt** to have

$$N_{t+1} = \frac{N_t}{4} \left(\frac{m}{N_t} \right)^{1-r} .$$

Proof. The fact is nothing but a reformulation of a general result on random graphs of Maurey [21] and Milman and Schechtman [23] based on martingales (Azuma's Inequality). (See [2, Chapter 7] for a systematic presentation.) ■

This fact can be directly applicable to $N(r)$ if x_i represents the cell into which i th member of S was mapped. We obtain that the probability of $N(r) < \mathbf{E}(N(r))/2$ is negligible. Consequently,

$$N(r) \geq \frac{1}{4} m e^{-\alpha} \frac{\alpha^r}{r!} , \quad (2)$$

except for a negligible number of cases.

4.2 The Basic Model and the Chaining Variant

In the basic model, we follow only sets of size 2, i.e., $r_t = 2$ for $t \geq 1$. When the usage of intermediate nodes (chaining) is possible, sets of fixed size r can no longer be tracked since the number of iterations will depend on n , and even complete failure to hash a set will decrease its size by 1. Instead define $r_0 = r = r(n)$ and $r_{t+1} = r_t - 1$. Note that, in this variant, r_0 must be greater than the desired lower bound for the number of iterations.

The following fact estimates $\mathbf{E}(N(r))$ for those pairs of r and m we are interested in. Note that in all of those cases $\mathbf{E}(N(r))$ is $\Omega(n^\epsilon)$ for some $\epsilon > 0$ and hence the event $N(r) < \mathbf{E}(N(r))/2$ is negligible.

Fact 2 *The expected value of $N_0(r)$, the initial (after the root node hashing) number of sets of size r , is given by:*

1. *If $r = 2$ and $m = O(n)$ then*

$$\mathbf{E}(N_0(r)) = \Omega(n) .$$

2. *If $r = 2$ and $m = n^{1+1/\lambda}$ for some fixed λ then*

$$\mathbf{E}(N_0(r)) = \Omega\left(n^{1-1/\lambda-1/n^{1/\lambda} \ln n}\right) = \Omega\left(n^{1-1/\lambda-o(1)}\right) .$$

3. *If $r = \lg \lg n / \lg \lg \lg n$ and $m = O(n)$ then*

$$\mathbf{E}(N_0(r)) = \Omega\left(n^{1-r(\lg r - \lg \alpha)/\lg n}\right) = \Omega\left(n^{1-o(1/\lg n)}\right) .$$

4. *If $r = \lg \lambda / \lg \lg \lambda$ and $m = n^{1+1/\lambda}$ for some fixed λ then*

$$\mathbf{E}(N_0(r)) = \Omega\left(n^{1-(r-1)/\lambda-o(1)}\right) ,$$

The rest of this section is outlined as follows. Suitable values for r_t will be set. Then we will compute a lower bound on the initial number of sets of size r_1 . (Since the algorithm is based on a random process, it may be extremely lucky and break this bound; thus the lower bound statement should be read with “ignoring negligible events” appended to it. Such quantification is implied henceforth.) We next estimate the number of sets of size r_{t+1} in iteration $t + 1$ as a function of the number of sets of size r_t in iteration t . Then an *explicit* lower bound for the number of sets of size r_t in iteration t is derived. The lower bound proofs are then completed by computing the minimal number of iterations \mathbf{Opt} must undergo before completion. The analysis is done for the basic model and the chaining variant together and then it is repeated for the parallel variant. We conclude with a remark explaining why all lower bound proofs are applicable to all the retries variants.

4.1 Root Node Hashing

The root node corresponds to the 0 th iteration. In it the set S of n keys is separated into m subsets using a random function $h : U \mapsto [m]$ into subsets S_1, S_2, \dots, S_m . Since h is a random function, the root node is accurately modeled by the well studied “balls into urns” model.

Let $\alpha = n/m$. For our needs it is sufficient to restrict attention to the case $\alpha = O(1)$. It is easy to verify that as n tends to infinity, the distribution of the number of balls (keys) in any single urn (cell) approaches the Poisson distribution parametrized by α . Let $N(r)$ be the number of subsets that have exactly r elements in them. Then, there exists n' such that for every $n > n'$

$$\mathbf{E}(N(r)) > \frac{1}{2} m e^{-\alpha} \frac{\alpha^r}{r!} \tag{1}$$

Without loss of generality assume that $n > n'$ from now on.

To see that $N(r)$ is “tightly concentrated” around its expectation we need the following fact.

Fact 1 *Let $F(x_1, \dots, x_n)$ be an arbitrary function of n variables which satisfies*

$$|F(x_1, \dots, x_n) - F(x_1, \dots, x_{i-1}, x'_i, x_{i+1}, \dots, x_n)| \leq 1$$

for any setting of i, x'_i and of x_1, \dots, x_n . Then, if x_1, \dots, x_n are independent random variables

$$\mathbf{Prob}(|F - \mathbf{E}(F)| > \lambda\sqrt{n}) < e^{-\lambda^2/2}$$

4 Proofs of Lower Bounds

We view hashing algorithms in the tree model from a parallel perspective. Each parallel iteration is an attempt to separate *all* subsets of S that were not previously separated, i.e., subsets that still have two or more keys in them. Thus successive iterations correspond to successive tree levels.

It should be obvious that with the usage of truly random functions the performance of the algorithm dependent on the size of S , but not on its content. Let **Opt** be the best possible algorithm for the current setting of the parameters (space and model variant). Our proofs are based upon showing that, with a dominant probability, there is a minimal number of iterations that **Opt** has to go through.

For simplicity in the analysis, we let **Opt** make the following assumptions:

Extra Memory Say that the problem restricts the memory usage to a total of m memory cells. This restriction will be weakened for **Opt** and it will be allowed to use m memory cells in *each* iteration.

Partial Separation A mapping of a set of keys to memory is called a *partial separation* if there exist two keys in the set that are mapped to distinct cells. **Opt** may consider any partial separation as being a total separation. The extremely unlikely case in which all keys from the set are mapped to the same memory is called a *complete failure*. Only complete failures need to be passed to the next iteration of **Opt**.

Restricted Set Size In iteration t , **Opt** has only to deal with (nodes containing) sets of r_t keys. Smaller or larger sets can be completely ignored. The exact value of r_t will be specified later.

Early Termination **Opt** need not be concerned with the case where there are fewer than $\lg n$ sets of size r_t . As soon as the number of sets drops below that bound, **Opt** can terminate immediately.

Higher Success Probability While analyzing **Opt** we will assume that failure probability is determined by $r = r_1$ although r_t keys are actually mapped. It will be shown that this assumption may only decrease the failure probability and works in **Opt** favor.

To account for the random nature of the hashing process, the following definition is introduced.

Definition 1 *Events that occur with probability smaller than $n^{-\epsilon}$ for some $\epsilon > 0$ are called negligible events. Dominating events are the complement of negligible events.*

Negligible events will be ignored in the following discussion, since even if they could be treated by **Opt** without *any* resource investment, the expected value of the performance measures will essentially be the same.

The algorithm behind this theorem uses truly random hash functions or, equivalently, high degree polynomials. As a more practical alternative, the class of pseudo-random hash functions defined in [9] can be used here too.

The next theorem shows that this meager improvement of $\lg \lg \lg n$ factor is the best possible and even it cannot coexist with employment of retries to achieve $O(1)$ search time. (As before, adding the power of retries to this variant of the model cannot improve $\overline{\text{DEPTH}}(n)$.)

Theorem 7

(a) If $\text{SPACE}_{cr}(\text{Alg}', n) = \text{SPACE}_c(\text{Alg}, n)$ then

$$\overline{\text{DEPTH}}_{cr}(\text{Alg}', n) = \Omega\left(\overline{\text{DEPTH}}_c(\text{Alg}, n)\right) .$$

(b) If $\text{SPACE}_c(\text{Alg}, n) = O(n)$ then

$$\Omega\left(\overline{\text{DEPTH}}_c(\text{Alg}, n)\right) = \Omega\left(\overline{\text{SEARCH}}_c(\text{Alg}, n)\right) = \Omega(\lg \lg n / \lg \lg \lg n) .$$

(c) If $\text{SPACE}_{cr}(\text{Alg}', n) = \text{SPACE}_c(\text{Alg}, n) = O(n)$ and $\overline{\text{DEPTH}}_{cr}(\text{Alg}, n) = O(\lg \lg n / \lg \lg \lg n)$ then

$$\overline{\text{SEARCH}}_{cr}(\text{Alg}', n) = \Omega\left(\overline{\text{DEPTH}}_c(\text{Alg}, n)\right) .$$

The general tradeoff is given by:

Theorem 8 If $\text{SPACE}_c(\text{Alg}, n) = n^{1+1/\lambda}$ then $\overline{\text{DEPTH}}_c(\text{Alg}, n) = \Omega(\lg \lambda / \lg \lg \lambda)$.

In a clear contrast to the first two variants, the ‘‘Simultaneous Retries’’, which may be applied in the parallel variant, lead to a significant improvement in DEPTH because they allow the folding of many iterations into one. Nevertheless, constant time hashing cannot be achieved in this case as well.

Theorem 9 If $\text{SPACE}_p(\text{Alg}, n) = O(n)$ then $\overline{\text{DEPTH}}_p(\text{Alg}, n) = \Theta(\lg^* n)$.

Neither retries nor chaining can further decrease the maximal insertion time of the parallel variant.

Theorem 10 If memory usage is restricted to $O(n)$ then

$$\overline{\text{DEPTH}}_{rcp}(n) = \Omega\left(\overline{\text{DEPTH}}_p(n)\right) .$$

Remark There exists an algorithm, **DM**, due to Dietzfelbinger and Meyer auf der Heide [9], for managing a dynamic data hash table that achieves, with very high probability, constant worst case performance. However, **DM** does not contradict the stated lower bounds since it does not fit in our basic model nor into any of its variants. In particular, **DM** pipelines the insertions; processing an inserted element can continue for up to n^ϵ steps after the insertion takes place; the algorithm allows keys to be fetched even if they are not “fully” inserted. Still, as the lower bounds indicate, there is no easy way of constructing a fast parallel version of **DM**. There are always keys for which **DM** requires as many as n^ϵ function applications.

With the help of retries, the lower bound of Theorem 1 can be met:

Theorem 3 *There is an algorithm **RetryShallow** which uses linear space (i.e., $\text{SPACE}_r(\text{RetryShallow}, n) = O(n)$), that gives*

- (i) $\overline{\text{TIME}}_r(\text{RetryShallow}, n) = O(n)$,
- (ii) $\overline{\text{DEPTH}}_r(\text{RetryShallow}, n) = O(\lg \lg n)$, and
- (iii) $\text{SEARCH}_r(\text{RetryShallow}, n) = O(1)$.

This algorithm is a variant of the **FKS** algorithm. The improvement in $\overline{\text{DEPTH}}(n)$ is accomplished by using a different, more adaptive, memory allocation scheme while executing the retries. This algorithm is optimal with respect to all parameters, even if we count arithmetic operations and limit word size to $O(\lg |U|)$. Moreover, if we restrict the algorithm to the basic model by eliminating retries, then all the parameters (except for $\text{SEARCH}(\text{Alg})$ which will be the same as $\text{DEPTH}(\text{Alg})$) will remain optimal:

Theorem 4 *There is an algorithm **BasicShallow** for which*

- (i) $\overline{\text{SPACE}}(\text{BasicShallow}, n) = O(n)$,
- (ii) $\overline{\text{TIME}}(\text{BasicShallow}, n) = O(n)$, and
- (iii) $\overline{\text{DEPTH}}(\text{BasicShallow}, n) = \overline{\text{SEARCH}}(\text{BasicShallow}, n) = O(\lg \lg n)$.

A non-trivial worst case upper bound for SPACE is not possible here because, for any such bound there are (admittedly rare) cases in which enough failures occur to force an algorithm to overflow this bound.

The general tradeoff between space and depth is given by

Theorem 5 *If $\text{SPACE}(\text{Alg}, n) = n^{1+1/\lambda}$ then $\overline{\text{DEPTH}}_r(\text{Alg}, n) = \Omega(\lg \lambda)$.*

Can the common practice of using internal nodes for storage help by more than a constant factor? Again, perhaps surprisingly, the answer is positive:

Theorem 6 *Both $\text{SPACE}_c(n) = O(n)$ and $\overline{\text{DEPTH}}_c(n) = O(\lg \lg n / \lg \lg \lg n)$ can be achieved simultaneously.*

exploiting internal nodes is possible then the algorithm can leave one element at v no matter which hash function is selected for the node.

Despite its name, this variant does not lead directly to a PRAM algorithm. The major difficulty is the assignment of processors, that have completed handling their original key, to assist the other processors with the yet unhashed keys.

One possible hashing variant was deliberately omitted from the above list; we do not permit the merging of nodes in the hash trees. Intuitively, merges *lose* separation information, and omitting merges from a hashing algorithm should only improve its performance. It is easy to verify that the TIME , DEPTH and SEARCH complexity measures can only decrease as a result of eliminating merge operations. The only possible merit of merging is to SPACE . It will be evident from the lower bounds proofs that merging cannot improve an algorithm with respect to all complexity measures defined above.¹

Most hashing algorithms deviate from our basic model by allowing one or both of the retries or the chaining variants. The parallel variant is mentioned as an alternate hashing idea in [12], and used by Matias and Vishkin [20].

3 Results

The most interesting algorithms are those that achieve $\text{SPACE}(\text{Alg}, n) = O(n)$ i.e., linear space. In his seminal paper “Should Tables be Sorted?” [30], Yao asked if one can simultaneously achieve $\text{SPACE}(n) = O(n)$ and $\overline{\text{SEARCH}}(n) = O(1)$. In our *basic* model, this is impossible.

Theorem 1 *If $\text{SPACE}(\text{Alg}, n) = O(n)$ then $\overline{\text{DEPTH}}(\text{Alg}, n) = \overline{\text{SEARCH}}(\text{Alg}, n) = \Omega(\lg \lg n)$.*

However, allowing retries, Yao gave an algorithm Y which achieves $\text{SPACE}_r(\text{Y}, n) = O(n)$ and $\text{SEARCH}_r(\text{Y}, n) = O(1)$ for large enough universes. For small universes, $q = n^{O(1)}$, Tarjan and Yao [28] showed how linear storage and constant search time can be maintained. Fredman, Komlós and Szemerédi [10] closed the gap by an algorithm FKS that satisfies $\text{SPACE}_r(\text{FKS}, n) = O(n)$ and $\text{SEARCH}_r(\text{FKS}, n) = O(1)$ for *any* universe size and any input set. Analyzing their algorithm, we find that while insertion time $\overline{\text{TIME}}_r(\text{FKS}, n) = O(n)$, (i.e., on the average we apply only a constant number of functions to each element), $\overline{\text{DEPTH}}_r(\text{FKS}, n) = \Omega(\lg n)$, so some element will be hashed $\Omega(\lg n)$ times, and this is the time required for hashing the elements in parallel using the FKS scheme.² A natural question that arises is whether this parameter can decrease to $O(1)$? We answer this question in the following theorem:

Theorem 2 *If $\text{SPACE}(\text{Alg}, n) = O(n)$ then $\overline{\text{DEPTH}}_r(n) = \Omega(\lg \lg n)$.*

¹However, it is interesting to note that merging technique is useful for the construction of good pseudo-random functions, which may be used for implementing hashing algorithms [9].

²Indeed the parallel hashing scheme of Matias and Vishkin [20] being based directly on FKS takes $O(\lg n)$ parallel time.

the performance of Alg on a worst case set S of size n , and by

$$\overline{\text{PARAM}}(n) = \min_{\text{Alg}} \overline{\text{PARAM}}(\text{Alg}, n)$$

the performance of the best algorithm on its worst case set S . At times it will be useful to ignore the probabilistic performance, and consider the worst possible performance of Alg (over all possible runs) on the worst case input, which we denote by $\text{PARAM}(\text{Alg}, n)$.

2.4 Variants of the Basic Model

Finally, we consider more powerful algorithms than those permitted by the basic model:

Retries An algorithm may allocate (say) m boxes (children) for n balls residing at a node v , and find that in throwing them randomly they all fall into one or very few cells. This is an unlikely event that causes a waste of space. The algorithm is allowed to consider this (or other more likely events) “bad” and try again. We do not charge for space used in v . To maintain the meaning of depth in this variant we create one single child for v and move all the balls there.

We attach the subscript r to the resources measures in this model, e.g., $\text{SPACE}_r(\text{Alg}, S)$ and $\overline{\text{DEPTH}}_r(n)$ etc. Note that SEARCH_r may be much smaller than DEPTH_r , since while a search is being performed no function application should be done at a node with only one child.

Chaining This variant allows the algorithm to store elements in internal nodes as well. Specifically, when m keys reach a node v , one of them is stored in v and the remaining $m - 1$ proceed to v 's children. The term *chaining* is used since this variant generalizes hashing techniques in which a chain (a linked list) of keys can be stored in hashing array positions.

The subscript c is added to the resources measures. Clearly SEARCH_c and DEPTH_c are the same again, since even if no branching occurs at node v , the element being searched for should be compared to the one residing at v .

We allow a combination of chaining and retries which is denoted by the double subscript cr . In this combination the algorithm may leave a key in an internal node, even if the function used at this node was discarded. Obviously, such a node cannot be skipped in a search.

Parallel Hashing In this variant of the model, we allow the algorithm to try in parallel several hash functions in a node v , and then pick one of them to create v 's children. Space here is counted as the sum of ranges of *all* those functions. Subscript p is added in this variant to the resources measures.

This variant may be combined with the two previous ones; if retries are permitted then the algorithm may choose not to use any of the hash functions that were tried; if

3. We deliberately deal here with the static case, i.e., when all the elements to be inserted are known in advance. This does not restrict the generality of the lower bounds. However, the algorithms presented are for the static case only.
4. Yao’s cell probe model [30], the standard general model for hashing, can also be described as a tree in a similar way. Our model differs from his in the way that a decision tree differs from a Turing machine. The cell probe model allows each cell a limited number of bits (depending on U), but these can encode arbitrary objects and be computed at no cost. Our cells contain either elements or functions. Functions can only be applied to elements and two elements can only be tested for equality. Our model, being more structured, is cleaner and easier to analyze, though less general.

2.3 Resources

We have seen that the stochastic process determined by a hashing algorithm Alg given $S \subset U$ of size n , is described by a random tree. The main resources of Alg operating on S are natural parameters of this tree.

Space The space required, or hash table size, denoted by $\text{SPACE}(\text{Alg}, S)$, is simply the total number of nodes in the tree.

Insertion Time We denote by $\text{TIME}(\text{Alg}, S)$ the total insertion time. This is the sum of depths of all leaves containing an element, i.e., the number of hash function applications to all the elements. Each application counts as one time unit. (Indeed, many practical algorithms use hash functions which can be evaluated in constant time.)

Parallel Insertion Time We denote the depth of the tree by $\text{DEPTH}(\text{Alg}, S)$. This is the parallel insertion time under the assumption that each processor is assigned one key and this processor alone is responsible for inserting this key. This parameter has two important meanings for sequential algorithms as well. It captures the number of functions needed to resolve the “worst” pair of elements, and the worst case insert and search time.

Search time will not be identical to insertion time in the more general models, so we devote a different notation to it:

Maximum Search Time This is the largest number of function applications needed to find out if $x \in U$ is in S using the tree generated by Alg on S . This parameter will be denoted by $\text{SEARCH}(\text{Alg}, S)$. In the basic model this definition coincides with that of $\text{DEPTH}(\text{Alg}, S)$. However, this will no longer be true when the model is elaborated, although we will still have $\text{SEARCH}(\text{Alg}, S) \leq \text{DEPTH}(\text{Alg}, S)$.

Let PARAM be a generic parameter (SPACE , TIME , DEPTH or SEARCH), then $\overline{\text{PARAM}}(\text{Alg}, S)$ will denote the expectation of PARAM with respect to the random choices made by the algorithm Alg , so $\overline{\text{PARAM}}(\text{Alg}, S) = \mathbf{E}(\text{PARAM}(\text{Alg}, S))$. We denote by

$$\overline{\text{PARAM}}(\text{Alg}, n) = \max_{|S|=n} \overline{\text{PARAM}}(\text{Alg}, S)$$

m , picks a hash function $h : U \mapsto [m]$ according to some distribution, and partitions S into subsets S_1, S_2, \dots, S_m (some empty) such that $S_i = h^{-1}(i) \cap S$. Following the literature, these subsets are sometimes called *buckets*. The function h is stored at the root of the tree. The root has m children and the subsets S_i move to the different children of the root.

The process is repeated for each S_i that contains at least two elements. The refining halts when every leaf contains at most one element from S .

A search for an element $x \in U$ in the hash table is easily performed by following the path from the root which is determined by applying the hash functions at internal nodes to the element x , and when reaching a leaf, comparing x to the element residing there if such an element exists.

This model leads to an alternate view of a hashing algorithm as an element distinctness proof generator. The input is a set of distinct keys taken from a universe with no order relation defined on it. The output is a proof that all elements are distinct. The proof components are functions from the universe to a bounded range.

2.2 Comments

1. The two types of strategic decisions made by a specific algorithm are the choice of range (i.e., number of children) for the hash function, and the choice of distribution used in selecting a particular function to this range. We *assume* that the function used is a truly random function, i.e., all possible functions are given equal probability.

This assumption follows the tradition in the analysis of hashing algorithms. In almost all such analysis (see e.g., [18, pages 514–517]) and [22, pages 120–124] for text book examples as well as [17, 29, 4, 5, 6, 24, 25]) it is assumed that the hash functions used are random or alternately, that the functions is fixed, but the input set is selected at random. The reason for this assumption is that random functions were intuitively perceived as the best for hashing [19]. This intuition was proved correct under quite general conditions by Ajtai, Komlós and Szemerédi [1]. (It should be noted however that random functions are not always the best. For example, if $|U| = O(n)$, then the identity function might produce much fewer collisions than a truly random function.)

With our random functions assumption, the hashing process that occurs in a node can be described as the act of independently sending each element of U to each possible child with uniform distribution. Analyzing the full hashing algorithm is reduced to analyzing a natural process of successively throwing identical balls into boxes until all the balls reside in distinct boxes.

2. Most common algorithms are stronger than the process we described—they use retries when the chosen hash function is extremely bad (e.g., all elements were mapped to one cell), and allow the storage of elements in the internal nodes of the tree as well as in the leaves. These generalizations and others will be considered later by introducing variants to the basic model.

1 Introduction

Hashing is one of the most important concepts in Computer Science. Its applications touch almost every aspect of this field—operating systems, file structure organization [16], communication, parallel and distributed computation, efficient algorithm design and even complexity theory [26, 27]. Nevertheless, the most common use of hashing is for the very fundamental question of efficient storage of sparse tables (see [22] and [18] for a systematic study).

A fundamental result of Fredman, Komlós and Szemerédi [10] shows that n elements (keys) from a universe of any size can be hashed to a linear size table in linear expected time, allowing for constant search time. It was observed, however, that although the average insertion time per element is constant, parallel application of this algorithm cannot work in constant time. The reason is that while the average is constant, some elements will have to be hashed a non-constant number of times.

In this paper we study the question of whether parallel hashing can be done in constant time. We present a simple new general model that captures many natural (sequential and parallel) hashing algorithms. In a game against nature, the algorithm and coin-tosses cause the evolution of a random tree, whose size corresponds to space (hash table size), and two notions of depth correspond to the longest probe sequences for insertion (parallel insertion time) and search of a key respectively.

We study these parameters of hashing schemes by analyzing the underlying stochastic process, and derive tight lower and upper bounds on the relation between the amount of memory allocated to the hashing execution and the worst case insertion time. In particular, we show that except for extremely unlikely events, every input set of size n will have members for which $\Omega(\lg \lg n)$ applications of a hash function are required. From a parallel perspective we obtain that if n processors are each given a key drawn from a large universe and if the input keys cannot be exchanged among the processors, then $\Omega(\lg \lg n)$ expected time is required to hash the input keys into $O(n)$ space. This is despite the existence of serial algorithms which achieve constant amortized time for insertion, as well as constant worst case search time [8].

Three variants of the basic model, which represent common hashing practice, are defined and tight bounds are presented for them too. The most striking conclusion that can be drawn from the bounds is that, under all combinations of model variants, not all keys may be hashed in constant time.

2 The Tree Model for Hashing

2.1 The Basic Model

The process of inserting a set S of n elements taken from some universe U into a hash table can be thought of as a process of refining partitions, and is depicted simply by a tree. Originally, all elements reside in a single node (the root). The algorithm chooses a range

The Tree Model for Hashing: Lower and Upper Bounds *

Joseph Gil

Technion – Israel Institute of Technology

Friedhelm Meyer auf Der Heide
Universität Paderborn

Avi Wigderson
The Hebrew University of Jerusalem

Abstract

We define a new simple and general model for hashing. The basic model, together with several variants capture many natural (sequential and parallel) hashing algorithms and represent common hashing practice. Our main results exhibit tight tradeoffs between hash table size and the number of applications of a hash function on a single key.

*This work included in the first author Ph.D. dissertation [11] and appeared in a preliminary form in [14].
Research supported in part by the Leibniz Center for Research in Computer Science