Not All Keys Can be Hashed in Constant Time (Preliminary Version)

Joseph Gil *† Friedhelm Meyer auf der Heide ‡§ Avi Wigderson *

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 [6], communication, parallel and distributed computation, efficient algorithm design and even complexity theory [8,9].

A multitude of models and algorithms for hashing have been suggested and analyzed. However, almost all of them are specific in their assumptions and results. We present a simple new 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 respectively to the largest probe sequences for insertion (parallel insertion time) and search of a key.

*Dept. of Comp. Sc. Hebrew University, Jerusalem 91904, Israel.

†Research supported in part by the Leibnitz Center for Research in Computer Science.

‡Fachbereich 17, Mathematik/Informatik, Universität-GH Paderborn, D-4790 Paderborn, Fed. Rep. of Germany

§Research supported in part by DFG grants Me 872/1-3, and the Leibnitz Center for Research in Computer Science.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

© 1990 ACM 089791-361-2/90/0005/0244 $1.50

A fundamental result of [4] shows that n elements from any universe can be hashed to a linear size table in linear time, allowing for constant search time. It was observed, however, that although average insertion time per element is constant, parallel application of this (and other) algorithms cannot work in constant time. The reason is that while the average is constant, some elements will have to be hashed non-constant number of times.

Our main results exhibit tight trade-offs between space and parallel time, in the basic model and three variants, which capture standard hashing practice.

2 The Basic Model

The process of inserting n elements $A = \{a_1, a_2, \ldots, a_n\}$ taken from some universe U into a hash table can be thought of as a process of refining partitions, and is simply depicted by a tree. Originally, all elements reside in a single node (the root). The algorithm chooses a range m, picks a function $f : U \mapsto [m]$ according to some distribution, and partitions A into sets $A_1, A_2, \ldots, A_m$ (some empty) s.t. $x_j \in A_i$ iff $f(x_j) = i$. The function f is stored at the root, and the sets A_i move respectively to m children of the root. The process repeats for each A_i that contains at least two elements. This refining process halts when every leaf contains at most one element from A.

A search for an element $b \in U$ in the hash table is easily performed by following a path from the root that is determined by applying the hash functions at internal nodes to the element b, and when reach-

ing a leaf comparing b to the element residing there (if there is one).

This approach leads to an alternate view of hashing algorithm as an element distinctness proof generator. The input is a set of distinct keys taken from 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.

2.1 Comments

1. 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. The existence of algorithms for the dynamic case that achieve both constant amortized time and minimal worst time remains open.

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 storing elements in the internal nodes of the tree as well as in leaves. We shall consider these generalizations later.

3. Yao's cell probe model [10], which is the standard general model for hashing can also be described as a tree in a similar way. Our model differs from it in a way that a decision tree differs from a Turing machine. He allows each cell a limited number of bits, (depending on U), but these can encode arbitrary objects and be computed for free. 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, but less general.

4. The two types of strategic decisions made by a specific algorithm are the choice of range (number of children) for the hash function, and the choice of distribution on functions to this range. Simple convexity considerations show that the second choice is optimal (for the relevant method) when choosing a random function, i.e. sending each element of U independently to each possible child with uniform distribution. Hence, analyzing the hashing process is reduced to analyzing a natural process of successively throwing identical balls into boxes until all the balls reside in distinct boxes.

2.2 Resources

The stochastic process determined by a hashing algorithm H given $A \subset U$ of size n, is described by a random tree. The main resources of H operating on A are natural parameters of this tree.

Space The space required, or hash table size, denoted by $S_H(A)$, is simply the total number of nodes in the tree.

Insertion Time We denote by $T_H(A)$ the total insertion time. This is the total depth of leaves containing an element, i.e. the number of applications of hash functions to elements. Each application counts as one time unit. Later we discuss the number of arithmetic operations.

Parallel insertion time Denoted by $D_H(A)$ is simply the depth of the tree. (Allowing to refine all leaves in parallel, or equivalently, to apply one hash function to each element in parallel.) 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 algorithms, so we devote a different notation for it.

Maximum search time This is the largest number of function applications to find out if $b \in U$ is in A, using the tree generated by H on A. This parameter will be denoted by $F_H(A)$.

Let P be a generic parameter (e.g. S,T,D,F). Calligraphic case letters will denote the expectation of our random variables, with respect to the random choices made by the algorithm H. So $\mathcal{P}_H(A) = E[P_H(A)]$. We denote by

$$\mathcal{P}_H(n) = \max_{|A|=n} P_H(A),$$

the performance of H on a worst case set A, and by

$$\mathcal{P}(n) = \min_H \mathcal{P}_H(n)$$

the performance of the best algorithm on its worst case set A.

2.3 Variants of the Model

Finally, we consider more powerful algorithms than described above, in the following variants. Most hashing algorithms deviate from our basic model by allowing one or both of the following:

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 fell into one or very few boxes. this is an unlikely event, that causes a big waste of space. The algorithm is allowed to consider this (or other more likely events) "bad", and try again. To maintain the meaning of depth (the parameters $D, \mathcal{D}$), we create one single child to v, and move all the balls there. We attach the superscript r to measures for this model, e.g. $S^r_H(A)$ and $\mathcal{D}^r(n)$ etc. Note that here F^r may be much smaller than D^r, since when searching no function application is needed at a node that has only one child.

Chaining Here we allow the algorithm to store elements in internal nodes too. Specifically, when m keys reach a node v, only m-1 proceed to v's children, and one of them is stored v. The word chaining is termed here since this variants models hashing techniques where a chain of keys can be stored in hashing array positions. We add the superscript c to the complexity measures. Clearly F^c and D^c are the same again, since even if there is no branching at node v, the element we search for should be compared to the one residing at v.

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* functions. Superscript p is added in this variant to the complexity measures. This variant can be combined we the two previous ones; if retries are permitted then the algorithm may choose not to use any of the the hash functions that were tried; if exploiting internal nodes is possible then the algorithm would leave one element at v no matter which (if any) hash function is selected for the node. Despite the name this variant does not lead directly to a *PRAM* algorithm. The major difficulty is assignment of idle processors to tasks.

3 Results

The most interesting algorithms are those that achieve $\mathcal{S}_H(n) = O(n)$, i.e. linear space. In his seminal paper "Should Tables be Sorted?", Yao asked if one can simultaneously achieve $\mathcal{S}(n) = O(n)$ and $\mathcal{F}(n) = O(1)$. In our *basic* model, this is impossible.

Theorem 1 *If* $\mathcal{S}_H(n) = O(n)$ *then*

$$\mathcal{D}_H(n) = \mathcal{F}_H(n) = \Omega(\log\log n)$$

However, allowing retries, Yao gave an algorithm Y achieving $\mathcal{S}^r_Y(n) = O(n)$ and $\mathcal{F}^r_Y(n) = O(1)$, for large enough universes U. Fredman, Komlós and Szemerédi [4] closed the gap by an algorithm *FKS* achieving $\mathcal{S}^r_{FKS}(n) = O(n)$ and $\mathcal{F}^r_{FKS}(n) = O(1)$ for *any* universe U (and any set A). Analyzing their algorithm, we find that while insertion time $\mathcal{T}^r_{FKS}(n) = O(n)$, (i.e. on the average we apply only a constant number of functions to each element), $\mathcal{D}^r_{FKS}(n) = \Omega(\log n)$, so some element will be hashed $\Omega(\log n)$ times, and this is a lower bound on hashing the elements in parallel using the *FKS* scheme. A natural question that rises here is whether this parameter decrease to $O(1)$? We answer this question by giving:

Theorem 2 *If a hashing scheme is restricted to* $O(n)$ *size memory then* $\mathcal{D}^r(n) = \Omega(\mathcal{D}(n)) = \Omega(\log\log n)$.

With the help of retries, the lower bound of theorem 1 can be achieved.

Theorem 3 *There is an algorithm* H *which uses retries and linear space (* $\mathcal{S}^r_H(n) = O(n)$*), that gives* $\mathcal{T}^r_H(n) = O(n)$*, as well as* $\mathcal{D}^r_H(n) = O(\log\log n)$ *and* $\mathcal{F}^r_H(n) = O(1)$.

This algorithm is a variant to the FKS algorithm. The improvement in $\mathcal{D}(n)$ is achieved by using a different, more adaptive memory allocation scheme while doing the retries. This algorithm is optimal with respect to all parameters, even if we count arithmetic operations and limit word size to $O(\log U)$. Moreover, if we restrict the algorithm to the basic model, all the parameters, except for F_H that will be the same as D_H will remain optimal.

The general trade-off between space and depth is given by

Theorem 4 *If* $\mathcal{S}^r_H(n) = n^{1+1/\lambda}$ *then* $\mathcal{D}^r_H(n) = \Omega(D^r_H(n)) = \Omega(\log \lambda)$.

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 5 *Both* $\mathcal{S}^c(n) = O(n)$ *and* $\mathcal{D}^c(n) = O(\log\log n/\log\log\log n)$ *can be achieved simultaneously.*

The algorithm behind this theorem uses truly random hash functions (or, equivalently, high degree polynomials), and these bounds hold only in our structured model. We do not know if the above bounds are achievable if we charge for arithmetic operations and limit word size. What we can show is that this meager improvement of $\log\log\log n$ is best possible.

Theorem 6 *If* $\mathcal{S}^c_H(n) = O(n)$ *then* $\mathcal{D}^c_H(n) = \Omega(\log\log n/\log\log\log n)$.

The general trade-off is given by:

Theorem 7 *If* $\mathcal{S}^c_H(n) = n^{1+1/\lambda}$ *then* $\mathcal{D}^c_H(n) = \Omega\left(\frac{\log\lambda}{\log\log\lambda}\right)$.

As before, adding the power of retries to this variant of the model cannot improve $\mathcal{D}(n)$.

Theorem 8 *Let m be the total memory used by the algorithm, If* $m = n^{1+1/\lambda}$ *then* $\mathcal{D}^{cr}(n) = \Omega(\mathcal{D}^c(n))$.

"Multiple retries" which are permitted in the parallel variant, perhaps at the cost of memory, do help because they allow folding many iterations into one.

Theorem 9 *If memory usage is restricted to* $O(n)$ *then* $\mathcal{D}^{crp}(n) = \mathcal{D}^{cp}(n)$, $\mathcal{D}^{rp}(n) = \mathcal{D}^{p}(n) = \Theta(\log^* n)$ *and* $\mathcal{F}^{rp}(n) = \Theta(1)$

The algorithm here is not directly applicable to *PRAM*, where the time for assigning tasks to processors has to be counted.

4 Proofs of lower bounds

We view hashing algorithms from a parallel perspective. Each parallel iteration is an attempt to separate *all* keys that were not separated in the previous iteration. Thus successive iterations correspond to successive tree levels.

For a certain hashing problem we need to examine H^*, the best possible algorithm for that problem. To proof the lower bounds we analyze the expected number of iterations of H^*. We let H^* 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 H^* and it will be allowed to use m memory cells in *each* iteration.

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

Restricted set size In the t^{th} iteration H^* will need to deal only with (nodes containing) sets of s_t keys. Smaller and bigger sets can be completely ignored. The exact value of s_t will be specified later.

Early termination H^* need not concern the case where there are fewer than $\log n$ sets of size s_t [1]. As soon as the number of sets drops below that bound, H^* can terminate immediately.

[1] Base 2 is implicit in all logarithms

Higher success probability While analyzing H^* we will assume that failure probability is determined by $s = s_1$ although s_t keys are actually mapped. It will be shown that this may only decreases the failure probability.

The rest of this section is outlined as follows. We first compute $k(s)$, the initial number of sets of size s. Next we estimate $k_t = k(s_t)$, the number of those sets in the $t+1$ iteration as a function of $k_{t-1}(s)$. Then we derive an explicit formula for $k_t(s)$. Proofs are then completed by picking suitable values of s for the different cases, and obtaining the lower bound from that. This process is done for the basic model and the chaining variant together, and then it is repeated for the parallel variant. We end with a remark explaining why all lower bound proofs are applicable to the retries variants of the model.

4.1 Root node hashing

The root node will be called iteration 0. In it n keys are hashed into m memory cells. The set $A = \{a_1, a_2, \ldots, a_n\}$ is separated using a random function $f : U \mapsto [m]$ into subsets $A_1, A_2, \ldots, A_m$. Let $k = k(s)$ be the number of subsets that have exactly s elements in them and let $\kappa = \kappa(s) = k(s)/m$.

Lemma 1 *Let $\mu = n/m$, if $\mu = \mu(n) < C$ for some constant C and $s = o(n)$ then*

$$\begin{aligned} \lim_{n\to\infty} E(k(s)) &= me^{-\mu}\frac{\mu^s}{s!} \\ \sigma(k(s) &\leq E(k)^{1/2} \end{aligned}$$

Proof Let $x_i = x_i(s)$ be the random variable defined by

$$x_i = \begin{cases} 1 & \text{if } |A_i| = s \\ 0 & \text{otherwise} \end{cases}$$

since the mapping done by f is uniform

$$E(x_i) = Prob(x_i = 1) = \binom{n}{s}\left(\frac{1}{m}\right)^s\left(1-\frac{1}{m}\right)^{n-s}$$

When $s = o(n)$ this can be approximated by the Poisson distribution

$$\lim_{n\to\infty} E(x_i) = e^{-\mu}\frac{\mu^s}{s!}$$

The proof of the second part of the lemma is omitted.

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

Corollary 2 *If $E(k(s)) = \Omega(n^\epsilon)$ for some $\epsilon > 0$ then the event $k(s) < E(k(s))/4$ happens with probability $O(n^{-\epsilon})$ and hence it is negligible.*

Negligible events will be ignored in the following discussion, since even if they could be treated without any resource investment, the expected value of the algorithm performance measures will not benefit significantly.

4.2 The basic model and the chaining variant

In any iteration H^* must handle $k_t = k_t(s)$ sets of s_t keys each using m memory cells. The algorithm should allocate memory to cells in a way that will minimize the number of failures k_{t+1}. The following lemma reveals the memory allocation scheme used by H^*.

Lemma 3 *H^* uses a balanced memory allocation scheme, that is each of the k_t sets is hashed into m/k_t cells.*

Proof If s_t keys are mapped by a random function into ω memory cells, then the *complete failure* probability is ω^{1-s_t}. This probability is getting smaller as ω increases, and therefore a memory allocation cannot be optimal if some of the m cells are not utilized. Let $m = m_1 + m_2 + \cdots + m_{k_t}$ be a memory allocation of the m cells to the k_t sets. The expected value of k_{t+1} is given by

$$E(k_{t+1}) = \sum_{i=1}^{k_t} m_i^{1-s_t}$$

Clearly this is minimized when all m_i are equal ∎

H^* uses a complete failure probability derived from $s = s_1$, the initial size of the sets, so actually

$$E(k_{t+1}) = \sum_{i=1}^{k_t} m_i^{1-s}$$

and by lemma 3

$$E(k_{t+1}) = k_t \left(\frac{m}{k_t}\right)^{1-s}$$

The probability that k_{t+1} will be much smaller than its expected value is estimated by the following lemma

Lemma 4 *Prob* $\left(k_{t+1} < \frac{E(k_{t+1})}{4}\right) < e^{-\frac{E(k_t)}{4}}$

Proof Note that k_{t+1} is the sum of k_t independent binary random variables, with equal distribution. The lemma is obtained from application of Chernoff inequality. ∎

If k_t is $\Omega(\log n)$ and $t = O(\log\log n)$ then the event $k_i < E(k_i)/4$ for $i \leq t$ is negligible. Since the union of $O(\log\log n)$ negligible events is also a negligible event we can assume that $k_i < E(k_i)/4$ will never happen. For simplicity we permit H^* have

$$k_{t+1} = \frac{k_t}{4}\left(\frac{m}{k_t}\right)^{1-s}$$

Defining $\kappa_t = \frac{k_t}{m}$, will put the above in a simpler form

$$\kappa_{t+1} = \frac{\kappa_t^s}{4}$$

This representation demonstrates the fact that the fraction of sets of a given size decreases "only" double-exponentially, which gives rise to the double logarithmic lower and upper bounds. The solution of the above recurrence is given by

$$\kappa_t = \frac{\kappa_0^{s^t}}{4^{\frac{s^t-1}{s-1}}}$$

Again we benefit H^* by setting

$$\kappa_t = \left(\frac{\kappa_0}{4}\right)^{s^t}$$

The following lemma is the core of all lower bound proofs

Lemma 5 *H^* tree has at least*

$$\Omega\left(\frac{\log - \frac{\log m}{\log \kappa_0(s)-2}}{\log s}\right)$$

levels.

Proof The initial fraction of sets of size s is $\kappa_0(s)$. To count the iterations we set the following equation for r

$$m\kappa_r(s) = m\left(\frac{\kappa_0}{4}\right)^{s^r} = \log n$$

Before H^* performs r iterations it will still have more than $\log n$ sets, and it will not be able to terminate. The explicit value of r is given by

$$r = \frac{\log \frac{\log\log n - \log m}{\log \kappa_0(s)-2}}{\log s}$$

which is (asymptotically) the bound as stated in the lemma. ∎

When H^* cannot use internal nodes, we follow only sets of size exactly 2 i.e. $s_t = 2$ for $t \geq 1$. When usage of intermediate nodes is possible, sets of fixed size s can no longer be tracked, since the number of iterations is dependent on n, and even complete failure to hash a set will decrease its size by 1. Instead define $s_0 = s = s(n)$ and $s_{t+1} = s_t - 1$. Note that in this variant s_0 must be greater than the desired lower bound for the number of iterations.

The following lemma will estimate $E(k)$ for those certain pairs of s and m we are interested in. Note that in all of those cases $E(k)$ is at least $\Omega(n^\epsilon)$ for some $\epsilon > 0$ and hence the event $k < E(k)/4$ is negligible.

Lemma 6 *The initial $k = k_0(s)$ (after the root node hashing) is given by*

1. *If $s = 2$ and $m = O(n)$ then*
$$E(k) = \Omega(n)$$
2. *If $s = 2$ and $m = n^{1+1/\lambda}$ for some fixed λ then*
$$E(k) = \Omega(n^{1-1/\lambda})$$
3. *If $s = \frac{\log\log n}{\log\log\log n}$ and $m = O(n)$ then*
$$E(k) = \Omega\left(n^{1-\frac{\log\log n}{\log\log\log n}+o\left(\frac{\log\log n}{\log\log\log n}\right)}\right)$$
4. *If $s = \frac{\log\lambda}{\log\log\lambda}$ and $m = n^{1+1/\lambda}$ for some fixed λ then*
$$E(k) = \Omega(n^{1+1/\lambda-s/\lambda+o(1)})$$

Applying lemma 5 on the above estimates will yield the proofs for the lower bounds set by theorems 1, 4, 6 and 7.

4.3 The parallel variant

The memory allocation scheme as used by H^* is a little different here. Many hash functions can be applied in parallel to the same set. Let $\omega_{t,1}, \omega_{t,2}, \cdots$ be the cardinalities of ranges of those functions in step t. Let $\omega_t = \omega_1 + \omega_2 + \cdots$ be the total memory allocation for a certain set of size s_t. The probability that all those hash functions will be a *complete failure* is

$$\prod_j \omega_{t,j}^{1-s_t}$$

This probability is minimized when all the $\omega_{t,j} = 2$ and in this case the failure probability is

$$2^{\frac{\omega_t(1-s_t)}{2}}$$

Let $m = m_1 + m_2 + \cdots + m_{k_t}$ be a memory allocation of the m cells to the k_t sets. The expected value of k_{t+1} is given in the parallel hashing variant

$$E(k_{t+1}) = \sum_{i=1}^{k_t} 2^{\frac{m_i(1-s_t)}{2}}$$

And again this is minimized when all the m_i are equal. We concluded in

Lemma 7 *In the parallel variant all hash functions ever used by H^* are to a range of size 2. In the tth iteration $m/2k_t$ functions with a total range of size m/k_t are applied to each of the k_t sets of size s_t.*

We get that

$$E(k_{t+1}) = k_t 2^{\frac{m(1-s)}{2k_t}}$$

We set $s = \Theta(\log^* n)$ (which is big enough for the chaining parallel variant too). Note that lemma 4 holds in this case too. Let $\omega_t = m/k_t$. We will be interested only in values of ω_t which are much smaller than $\log n$. For those values the probability that k_{t+1} will be significantly small than its expected value is exponentially small. We get that

$$\omega_{t+1} \leq 4\omega_t 2^{\frac{s-1}{2}\omega_t}$$

In this variant H^* is required to iterate only until $\omega_t > \log\log\log n$, but this will not happen before $O(\log^* n)$ steps. (which is sufficient for both the parallel and the chaining parallel variants).

4.4 The retries variants

To complete this section we need to proof theorems 2 and 8. Note that the retries notion is useful if a certain application of a hash function was not satisfactory according to some criteria, then instead of learning to leave with it, the algorithm may draw polls again. However, our simplifications allow H^* a dichotomous classification of poll results. If there was a *complete failure* in a hash of certain internal node, then doing a retry is identical to what H^* will do in the next iteration. On the other hand, if this internal node was not a *complete failure*, we allowed H^* to ignore it, and since all nodes are independent doing a retry can only lead to a deeper tree.

Retries cannot help in the root node either. From lemma 1, even $O(\log n)$ retries in the root node will not yield a significantly better value for $k_0(s)$.

The above arguments could be summarized by the observation that separation information obtained from tree node, can only improve the insertion time. The equivalence of an algorithm without retries to an algorithm without retries was possible here because the H^* could use all its total memory allowance in each and every iteration. In general, using this technique to transform H an algorithm that uses retries into H' an algorithm that avoids retries, could increase memory utilization by a factor of up to $D_H(n)$.

5 Proofs of upper bounds

The algorithm presented by [4] can be sketched in the following way. In the root node a function $h(x)$ is used to map all keys to a table of size n. $h(x)$ is defined as $(ax \bmod p) \bmod n$ where p is a suitable fixed large prime, and a is picked at random in the range $[1, p-1]$. This function is (almost) 2-wise independent, and with high probability it will

satisfy

$$\sum_{1=1}^{n} 2|A_i|^2 \leq 5n$$

Thus $2|A_i|^2$ memory cells can be assigned to the set A_i without exceeding the linear memory bound. A sequence of similar functions $h_i(x)$ are then tried on the set A_i, until a complete separation function is found.

This functions are defined by

$$h_i(x) = (a'x \bmod p) \bmod (2|A_i|^2))$$

where p is as before and a' is picked again at random from the interval $[1, p-1]$.

The probability that $h_i(x)$ will not achieve complete separation is $\Omega(1)$, and since the number of the sets is $\Omega(n)$, the number of iterations will be $\Omega(\log n)$.

5.1 The basic model and the retries variant

Before we can improve on that we need the following lemma which is an easy consequence of the basic lemma of [4].

Lemma 8 *Let $h_i(x)$ be a function of the type $(ax \bmod p) \bmod (g|A_i|^2))$ that maps the bucket A_i into the range $[1..g|A_i|^2]$. If h_i is picked at random from all the functions of its kind then the probability that h is a one to one function is at least $1 - 1/g$.*

Our improvement will be based on a different approach to memory allocation. The idea is to classify all sets according to the log of their size. Each class uses a portion of the memory which is proportional to the initial number of keys in the set. The total memory used by a class in an iteration decreases geometrically. That memory is equally divided to all living sets in the class. The details are given in the following algorithm; As before let $k(s)$ be the initial count of sets of size exactly s, and let $k_t(s)$ be the count of "living" sets by iteration t.

Algorithm *QuickHash*
begin
  Find h, an initial FKS function for
  which $\sum_{1 \leq n} |A_i|^2 \leq 5n$
  Mark all sets A_i as generated by h
  Classify all sets having s' elements
  as having $s = 2^{\lceil \log s' \rceil}$ elements.
Loop1: **foreach** s s.t. there are marked sets of size s **pardo**
    $t \leftarrow 0$
    $g_t \leftarrow 16$
  Loop2: **while** there are still marked sets of size s **do**
      $k_t(s) \leftarrow$ #(marked sets of size s)
      allocate a memory block of size gs^2 to
      each set of size s
    Loop3: **repeat**
        foreach living set X of size s **pardo**
          Pick a random number a in
          the range $[1..p]$
          Apply $h(x) = (ax \bmod p) \bmod (g_t s^2)$ to
          all members of X
          if h gives a one to one function for X **then**
            Unmark X
          fi
        od
      until #(marked sets of size s) $\leq 2k_t(s)/g_t$
    od
    $g_{t+1} \leftarrow (g_t^2/4)$
    $t \leftarrow t + 1$
  od
end *QuickHash*

Time analysis: To analyze the body of loop3 we modify it in the following way: if by the end of loop the count of marked sets is not below the required level then all the loop effects are cleared. In this modification every iteration of loop3 can be modeled as a random variable (a count of sets that were unmarked) sampling. The expected value of this random variable is at most $k_t(s)/g_t$, and therefore the probability that the loop will fail is $1/2$ at the most.

In analyzing loop2 note that handling sets of any size clearly terminates before $g_t > n$. It is rather straightforward to verify that g indeed increases in every assignment of the type $g_{t+1} \leftarrow g_t^2/4$, and that $g_{t+1} \geq g_t^{1.5}$. We conclude that loop2 is executed $O(\log \log n)$ times. Incorporating the estimates for loop3, we get that the expected number of executions of loop3 in loop2 is $O(\log \log n)$ too.

To analyze the outer loop1, one must take into account the fact that many copies of loop2 are executed simultaneously in it. The probability that

one of the copies of loop2 will be executed more than its expected value can be estimated using Chernoff bounds. The probability that one instance of loop2 will take more than C times its expected value is $o(1/\log n)$, and since there are at most $O(\log n)$ such instances, the probability that the slowest of them will terminate in time longer than $O(\log\log n)$ is $o(1)$.

Note that the initial phase can be done in $O(1)$ expected time too, but it may require (with a low probability) a retry. We do not allow retries in the root node, and if the initial hashing was not satisfactory then the algorithm will fail. This of course cannot change the expected behavior of the algorithm.

Memory Analysis: Total memory used is $O(n)$: In each iteration of loop2 $k_s g s^2$ cells are allocated, since $k_{t+1}(s) \leq 2k_t(s)/g_t$ and $g_{t+1} = g_t^2/4$ we get that the memory usage in the $t+1$ iteration is at most $1/4$ of that of the tth iteration, thus showing that the total memory usage is dominated by the initial memory allocation. Extra memory beyond the FKS requirements is needed in the initial phase, because sets sizes can almost be doubled, when increased from s' to s. Another initial memory demand comes from the first phase allocation of $16s^2$ cells to every set instead of $2s^2$. It is easy to see that the initial memory usage is still $O(n)$.

If retries are to be avoided, then our algorithm can be modified to allocate a new memory block in each iteration, in this case memory usage will still be linear, but it could not be predetermined.

5.2 The parallel variant

The above algorithm is not applicable directly to *PRAM* machines, but its ideas can be used to supply an actual *PRAM* algorithm [5]. In contrast to that the following algorithm seems much harder to implement. In each iteration of it—more and more processors are drafted to the hashing of fewer and fewer keys. Locating those in need and organizing help for them require (probably) time, and the algorithm ignores that.

The algorithm for the parallel variant is almost identical to the above. The main difference is that the $g_t s^2$ memory block allocated to a set of size s in a the tth iteration is divided further to sub blocks of size $2s^2$, into which the parallel hashing is done. The failure probability in this case is $2^{-g_t/2}$ so loop2 can be repeated until there are $2^{1-g_t/2}k_t(s)$ "live sets" of size s. The new g_{t+1} factor can be set to $2^{g_t/2-3}$ which gives rise to the $O(\log^* n)$ behavior. Further details of the proof are similar to the previous one.

5.3 Chaining variant

Here we would like to use the power of chaining in order to bring the hash time down to $O(\log\log n/\log\log\log n)$. Let

$$r = C\frac{\log\log n}{\log\log\log n}$$

for some constant C. The algorithm will start with an FKS like function that will ensure that $\sum_{1\leq n}|A_i|^2 \leq 5n$, and then hashing will be used only on sets of size $s > r$.

The goal of this hashing will be set breaking: sets will be divided into subsets of size no more than r elements, instead of the usual attempt to achieve a complete separation.

When all sets are of size $\leq r$ they can be handled just by the ability of storing 1 elements in every internal nodes, adding at most r levels to the tree.

Let h' be a mapping from a fixed set of size s into memory of size gs^2. We say that h' is *bad* if one or more of the subsets (buckets) it defines is of size $> r$. We say that a class H is a *good breaker* if the probability that h picked at random from H is *bad* is at most g^{1-r}. This property is achieved by true random functions, as can be seen from lemma 1.

The outline of algorithm *QuickHash* is used here too for the set breaking. In the tth iteration of loop2, loop3 is

executed until at most $2k_t(s)g_t^{1-r}$ are unmarked. Thus

$$k_{t+1}(s) \leq 2k_{t+1}(s)g_t^{1-r}$$

If we require

$$k_{t+1}(s)g_{t+1} \leq k_t(s)g_t/2$$

we will insure that the total memory requirement is linear. We get $g_{t+1} = g_t^r/4$. Replacing the value for r and solving the recurrence we get that after $O(\log\log n/\log\log\log n)$ iterations, g_t will be

$O(n)$ and all the sets will be broken. As we said random functions are *good breakers*. However, true random functions are not useful for hashing as they require huge space for representation. If instead H will be the class of $r-1$ degree polynomials then H is a *good breaker* and it can be represented efficiently. However in this case each application of a hash function takes $O(r)$, and hence the total run time will not be reduced below $O(\log\log n)$ time (although the number of hash function application is still $O(\log\log n/\log\log\log n)$).

Recently, Dietzfelbinger and Meyer auf der Heide suggested in [1,2] a class DM of hash function that is a *good breaker*, which can be evaluated in $O(1)$ time and and represented in $O(n)$ memory. Implementation of their function, is not included in our basic model or its variants, since merging of tree nodes is used there. In general, merging of nodes in the hash tree can only increase separation time. Merging in [1] is only used to *increase randomness*, which is implicitly assumed in our model.

References

[1] Dietzfelbinger and Meyer auf der Heide. How to distribute a dictionary on a complete network. In *Proceedings of 22*nd *Annual ACM Symposium on Theory of Computing*, 1990.

[2] Dietzfelbinger and Meyer auf der Heide. A new class of universal hash functions and dynamic hashing in real time. to appear in ICALP 1990.

[3] Dietzfelbinger, Karlin, Mehlhorn, Meyer auf der Heide, Rohnert, and Tarjan. Dynamic perfect hashing: upper an lower bounds. In *29th Annual Symposium on Foundations of Computer Science*, pages 524–531, Oct 1988.

[4] Fredman, Komlós, and Szemerédi. Storing a sparse table with $O(1)$ worst case access time. *Journal of The Association for Computing Machinery*, 31(3):538–544, July 1984.

[5] Joseph Gil and Yossi Matias. $o(\log\log n)$ time hashing on pram. Mar 1990. In preparation.

[6] Anna Karlin and Eli Upfal. Parallel hashing—an efficient implementation of shared memory. In *18th Annual ACM Symposium on Theory of Computing*, pages 160–168, May 1983.

[7] Kurt Mehlhorn. *Data Structures and Algorithms 1: Sorting and Searching.* Springer-Verlag, Berlin Heidelberg, 1984.

[8] Michael Sipser. A complexity approach to randomness. In *Proceedings of 15*th *Annual ACM Symposium on Theory of Compu ting*, pages 330–335, April 1983.

[9] Larry Stockmeyer. A complexity approach to randomness. In *Proceeding of 15*th *Annual ACM Symposium on Theory of Computi ng*, pages 118–126, April 1983.

[10] Andrew Chi-Chih Yao. Should tables be sorted? *Journal of The Association for Computing Machinery*, 28(3):615–628, July 1981.