External Perfect Hashing

Per-Åke Larson and M V Ramakrishna

Data Structuring Group
Department of Computer Science
University of Waterloo
Waterloo, Ontario, Canada

ABSTRACT

A hashing function is perfect if it does not create any overflow records The use of perfect hashing functions has previously been studied only for small static sets stored in main memory In this paper we describe a perfect hashing scheme for large external files which we are currently investigating The scheme guarantees retrieval of any record in a single disk access This is achieved at the cost of a small in-core table and increased cost of insertions We also suggest a policy for limiting the cost of insertions and we study the tradeoff between expected storage utilization, size of the internal table and cost of insertions under this policy The results obtained so far are very promising They indicate that it may indeed by possible to design practical perfect hashing schemes for external files based on the suggested approach

Electronic mail

uucp {decvax,allegra,ihnp4}
!watmath!watdaisy!{palarson,mvramakrishn}
csnet {palarson,mvramakrishn}% watdaisy@ waterloo csnet

1. Introduction

Hashing is a popular technique for organizing internal tables as well as external files It is simple and retrieval is very fast on average However, the worst case may still be extremely poor, that is, the retrieval cost for any given record may still be high This may not be acceptable for certain real-time applications where there are strict bounds on the time available for retrieval of a record

The variation in retrieval speed is caused by overflow records If there were no overflow records the problem would disappear A perfect hashing function is a hashing function that does not create overflow records. The problem of finding a perfect hashing function for a given set of keys has previously been studied only for small static sets, stored in internal memory We are currently investigating whether, and how, this idea can be successfully extended to large dynamic external files The approach taken and preliminary performance results are summarized in this paper

The goal is to develop a simple, practical file structure that guarantees retrieval of any record in one access The proposed method requires an in-core directory to achieve this The results indicate that the directory can be kept quite small and that an overall load factor of 75% or higher can be maintained A method with the same goal, but based on a different approach was proposed by Gonnet and Larson [GNT, LRSN]

2. Background

Consider a hash table (or hash file) consisting of m buckets (pages), each with a capacity of b records. A set S of n records, $n \leq mb$, are to be stored in the table. A hashing function h, operating on the key of the record, assigns each record an address in the range 1,2, . . , m. The hashing function h is said to be perfect if no address receives more than b records. It is a minimal perfect hashing function if the table is of minimal size, that is, $m = \lceil n/b \rceil$.

Given a class of hashing functions, a hash table T and a set S of keys, the problem is to find a member of the class which is perfect for the set S and the table T. For practical reasons the hashing functions considered are normally fairly simple functions with a few unknown parameters. The problem of finding a perfect hashing function then boils down to determining the values of the parameters. This can be done in a systematic way or on a trial-and-error basis.

The problem of finding a perfect hashing function for a given set S was first studied systematically by Sprugnoli [SPRG]. He considered only internal hash tables and the following two classes of functions

$$h_1(x) = \left\lfloor \frac{x + s}{N} \right\rfloor$$

$$h_2(x) = \left\lfloor \frac{(xq + d) \bmod M}{N} \right\rfloor$$

where s, N and q, d, M, N are the unknown parameters and x denotes a key. He gave a systematic procedure for finding the parameters. The procedure is complicated and its time complexity is $O(n^3)$, with a large constant. It is guaranteed to find a perfect hashing function, if one exists within the given class, but the resulting load factor may be low. The cost of finding a perfect hashing function using his method grows very rapidly with the size of the set. To handle large sets, Sprugnoli suggested segmentation. An ordinary hashing function is first used to divide the given set into a number of small subsets. The problem of finding a perfect hashing function is then solved separately for each subset. The size and location of the hash tables for each subset, together with the parameters of each perfect hashing function, are stored in a separate table, which is addressed by the segmentation hashing function.

Jaeschke [JSCH] designed a simpler, systematic method for finding minimal perfect hashing functions. His method is called reciprocal hashing and utilizes a function of the form

$$h(x) = \left\lfloor \frac{C}{Dx + E} \right\rfloor \bmod m$$

Even though its time complexity is exponential, it is slightly faster than Sprugnoli's method for small sets. Another systematic method, was proposed by Chang [CHNG]. It has the advantage of giving order-preserving hashing functions. However, it does not appear practical because the calculations involve very large numbers.

Several researchers have studied trial-and-error methods. Cichelli [CCHL] considered alphanumeric keys and hash functions of the form

$$h(x) = \text{length } (x) + g \text{ (first character of } x) + g \text{ (last character of } x)$$

where g is defined by a table mapping characters into integers. The mapping is chosen by guided trial-and-error search so as to give a perfect hashing function. This idea has been extended to include other characters of the key by Cercone, Krause and Boates [CRCN]. In practice, the method appears to perform well, even for sets of a few hundred keys.

Fredman, Komlos and Szemeredi [FRDM] described a method, again based on trial-and-error, which has constant retrieval time and uses $n + o(n)$ storage. Their data structure implements Sprugnoli's segmentation idea. The class of hashing functions considered is defined by

$$h(x) = (kx \bmod p) \bmod s$$

where s is the size of the hash table (for the subset in question), p is a prime number greater than any of the keys and k is a parameter, $k < p$. For a given set of keys the complete data structure and the hashing functions can be constructed in $O(n)$ random expected time.

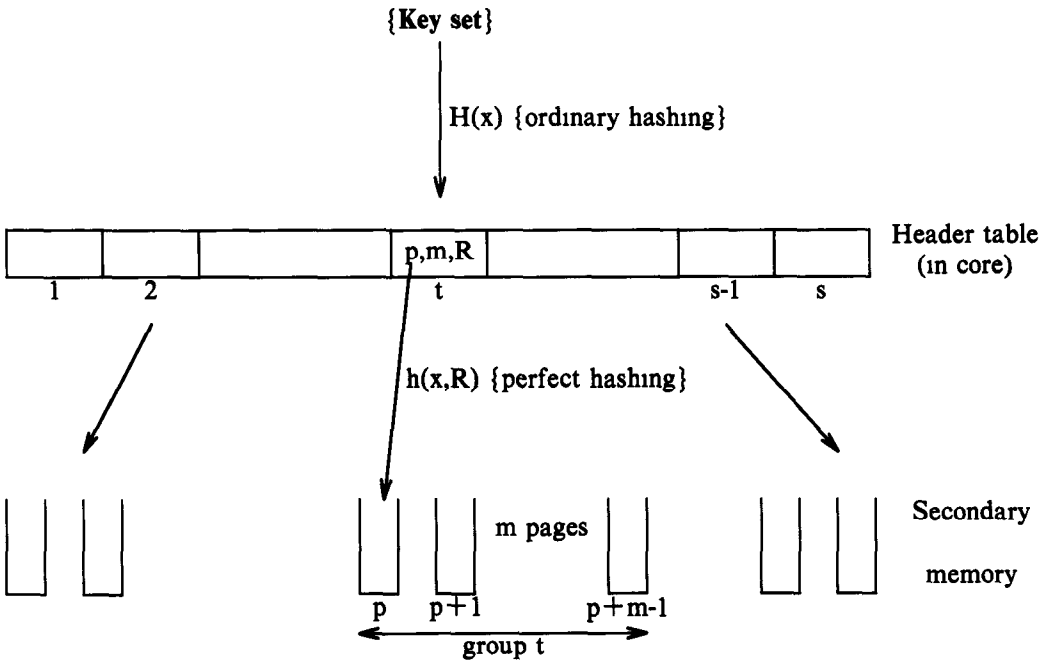

Figure 1: Illustration of external perfect hashing

Based on the ideas of Fredman et al, Cormack, Horspool and Kaiserswerth [CRMC] designed a more practical scheme, which is not limited to static sets Insertion of a new element may involve finding a new perfect hashing function for the subset into which the new record is inserted The study reported in this paper was largely motivated by their results

3. Perfect hashing for external files

The results in [FRDM], [CRMC] and [CRCN] indicate that, in practice, it is not difficult to find perfect hashing functions for small sets using a trial-and-error approach For external files records are normally stored on fairly large pages There are good reasons to believe that this should make it easier to find a perfect hashing function These observations led us to consider possible extensions of perfect hashing to large external files Segmentation is obviously necessary and for finding perfect hashing functions the trial-and-error approach appears more promising than the systematic methods known so far The two level data structure suggested by Cormack et al [CRMC] can immediately be extended to external files, as illustrated in Figure 1

A header table of length s is stored in main memory An ordinary hash function H maps keys into the header table Let **key group** t denote the set of keys hashing to address t, $1 \leq t \leq s$, of the header table Each entry in the table is of the form (p, m, R) where p is a pointer to a group of m contiguous pages on secondary storage and R is the set of parameters defining the perfect hashing function used for that key group Let (p_t, m_t, R_t) be the header table entry for key group t Let **page group** t denote the pages on which the records of key group t are stored, that is, pages $p_t, p_t + 1, \ldots, p_t + m_t - 1$ The address of a record in key group t is then given by $p_t + h(x, R_t)$, where $h(x, R_t)$ is the perfect hashing function having the parameter values R_t

Retrieval is extremely simple Given the search key x, compute $t = H(x)$ and extract (p_t, m_t, R_t) from the header table The page address is then given by $p_t + h(x, R_t)$ Read in that page and search the page for the desired record

To insert a record, first compute the page address as above and read in the page If the page is not full, insert the record If the page is full, a new perfect hashing function must be found, possibly increasing the number of pages in the group Rehash all the records using the new perfect hashing function and update the header table entry

Deletion of a record does not, as such, necessitate construction of a new hashing function However, we may want to prevent the load factor of the group from decreasing below a certain level If so, we construct a new perfect hashing function for the group (probably having fewer pages), rehash the records of the group and update the header table entry

The above description outlines a possible scheme but its usefulness depends on several factors How difficult is it to find a perfect hashing function when using large pages? What load factor is achievable? Can we find a practical class of hashing functions from which to choose a perfect hashing function? These and other questions are addressed in the subsequent sections

4. Randomly chosen functions

There are a total of m^n different mappings, or functions, of n keys into m addresses If we randomly choose one of the m^n functions, what is the probability that it will be perfect? That is, what is the probability that none of the addresses will receive more than b records? Let $P(n, m, b)$ denote this probability Let $F(n, m, b)$ denote the number of ways in which n keys can be distributed among m pages, of size b records each, without any one of the pages overflowing Clearly we have $P(n, m, b) = F(n, m, b) / m^n$ For the case $b = 1$, $F(n, m, 1)$ is extremely simple $F(n, m, 1) = m! / (m - n)!$ However, for $b > 1$ the function is much more complex [DVS]

$$F(n, m, b) = \sum_{0 \le f_i \le b} n! / \prod_{i=1}^{m} f_i!$$

where the summation is over all possible combinations of f_i such that $\sum_{i=1}^{m} f_i = n$. Davis and Barton point out that there is no closed form expression for $F(n, m, b)$ simpler than the one above [DVS] $F(n, m, b)$ can also be computed as the coefficient of $x^n / n!$ in the generating function $(1 + \frac{x}{1!} + \frac{x^2}{2!} + \quad + \frac{x^b}{b!})^m$ For computational purposes both formulas are essentially useless They are computationally slow and involve very large numbers However, we were able to derive a recurrence relation that gives a fast and easy way of computing $P(n, m, b)$

Consider a situation where n keys have been distributed among m pages without overflow The next record, the $(n + 1)$st, is now randomly hashed to a page Let p_0 denote the conditional probability that it will not cause overflow This probability can be expressed as

$$p_0 = \frac{P(n + 1, m, b)}{P(n, m, b)}$$

The $(n + 1)$st record will cause overflow if and only if it hits a page that already contains b keys The probability of this event is precisely $1 - p_0$ It can be expressed as

$$1 - p_0 = \binom{n}{b} \frac{P(n - b, m - 1, b)\,(m - 1)^{n-b}}{P(n, m, b)\,m^n}$$

By combining these two equations and rearranging, we obtain the following recurrence relation

$$P(n + 1, m, b) = P(n, m, b) - \binom{n}{b} P(n - b, m - 1, b)\,(m - 1)^{n-b} / m^n$$

Using this recurrence relation, we can easily compute a table of $P(n, m, b)$ for various values of m and n (n varies up to mb) The computation can be organized so that computing each new value in the table requires only five floating point operations The numbers involved are well scaled and round-off errors do not cause problems In Figure 2 the probability of a randomly chosen function being perfect is plotted for the case $b = 20$ and several values of m These results were computed using the above recurrence relation

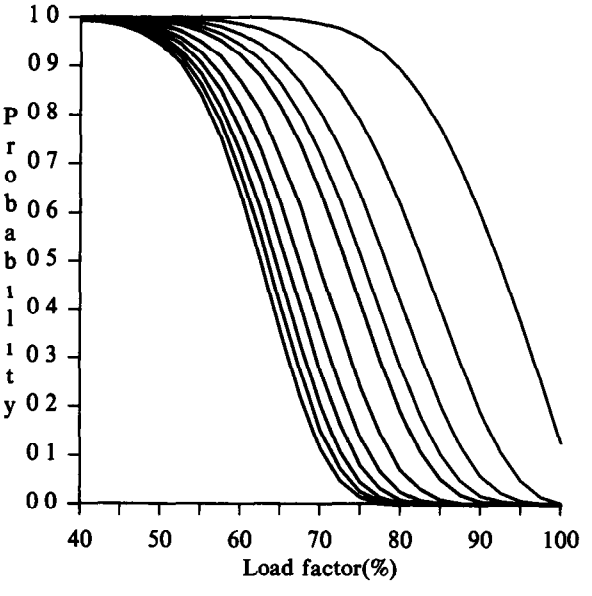

Figure 2: Probability of a randomly chosen function being perfect
$b = 20$, $m = 2,4,6,8,10,15, \quad ,40$

The graph indicates that the probability of finding a perfect hashing function drops very rapidly from almost 1 to almost 0 within a narrow load factor range This critical range moves slowly towards zero as the value of m increases The above results are quite encouraging For example, if we want to hash 150 records into 10 pages of size 20 records (load factor 75%) we can expect to find a perfect hashing function in 2 trials If we are willing to make 15 trials on average, we can hash 240 records into 15 pages,

corresponding to a load factor of 80%

5. Limiting the cost of finding a perfect hashing function

There is a tradeoff between the load factor and the cost of finding a perfect hashing function the higher the load on a group of pages, the higher the cost of finding a perfect hashing function will be. Furthermore, the larger the number of records (and pages), the higher will the cost be, even when the load factor is kept constant

As records are inserted and the number of records per group increases, we need some policy for how to increase the number of pages per group A straightforward solution is to keep the load factor constant (or within a narrow range) However, this approach has some disadvantages The cost of finding a perfect hashing function will steadily increase The cost of inserting a record will also increase because the hashing function must be recomputed more frequently The variation in the number of trials needed to find a perfect hashing function is also very high

The above disadvantages led us to consider a policy that attempts to balance the cost of finding a perfect hashing function and the load factor The idea is to place a bound on the number of trials and then distribute these trials over an acceptable load factor range so as to maximize the expected load factor Given n, we restrict the number of pages in a group to an interval $m_0 \leq m \leq m_1$ The lower bound is simply $m_0 = \lceil n / b \rceil$ The upper bound is chosen so that $n / m_1 b \approx 0.5$ The probability of a trial succeeding is very high when the load factor is as low as 0 5 Let $r = m_1 - m_0 + 1$ and let t be the maximum number of trials we are willing to make to find a perfect hashing function Partition t into $(t_1, t_2, \ldots, t_r)$ such that $t_1 + t_2 + \ldots + t_r = t$ Any such partitioning defines a policy for finding a perfect hashing function try up to t_1 randomly chosen hashing functions with m_0 pages, if there is no success, try up to t_2 functions with $m_0 + 1$ pages, etc Every such policy has an associated expected load factor and an overall probability of success P, that is, probability of having found a perfect hashing function after at most t trials We specify a lower bound on the probability of overall success, say $P \geq 0.99$ or $P \geq 0.98$ Given these restrictions, we want to find the policy $(t_1, t_2, \ldots, t_r)$ that gives the highest expected load factor There is a small probability, $(1 - P)$, of not succeeding in t trials If so, we continue trying with $m = m_1$, until a perfect hashing function is eventually found Hence, the actual number of trials may occasionally exceed t by a small amount

Mathematically the above approach leads to a nonlinear (integer) optimization problem with linear constraints, see the Appendix Solving this problem exactly is too time consuming, but a very close approximation may be obtained by dynamic programming The example below illustrates the discussion For this particular case the optimal policy is at most 4 trials with 11 pages, 4 with 12 pages, 1 with 13 pages and 1 with 15 pages If we have not succeeded in 10 trials, we continue trying with 17 pages until a perfect hashing function is found The probability of this event is only 0 0073 The expected load factor resulting from this policy is 75 53%

Parameters
$n = 180, b = 20, t = 10, P = 0.99$
Bounds $m_0 = 180/20 = 9, m_1 = 17$

Pages (m)	$P(n,m,b)$	Optimal Policy	Pr of final group size being m
9	000	0	000
10	005	0	000
11	100	4	342
12	331	4	526
13	578	1	076
14	758	0	000
15	867	1	048
16	930	0	000
17	963	0	007

Expected number of pages 11 92
Expected load factor 75 53%

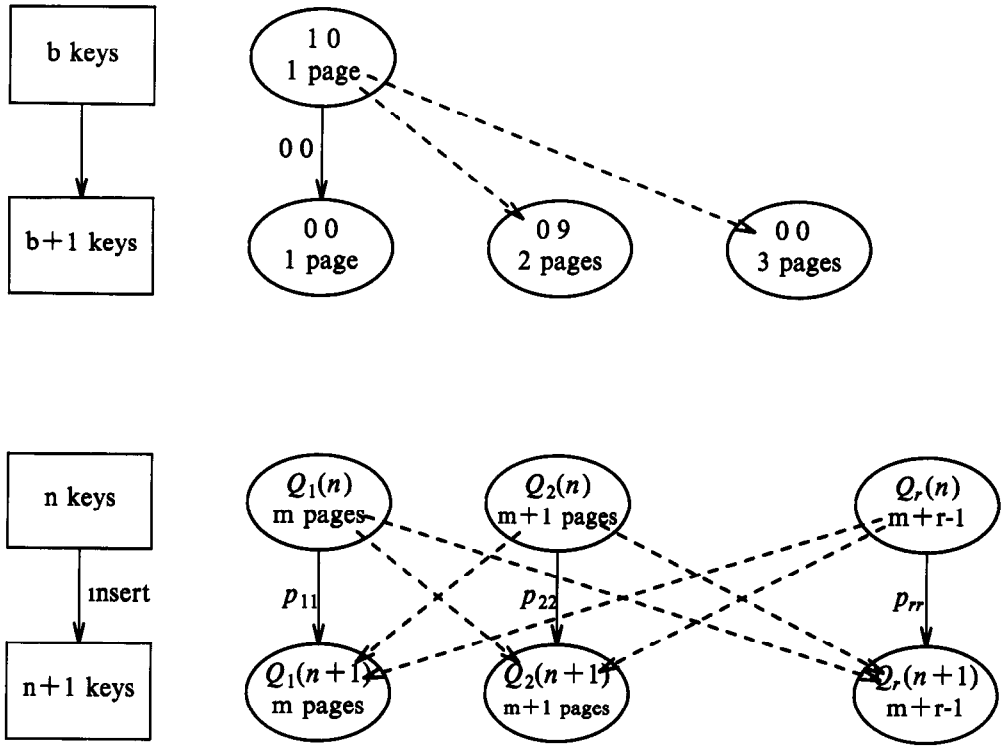

Figure 3: State transitions when inserting a record

$Q_i(n)$ is the probability that the group consists of $(m + i - 1)$ pages when there are n records

p_{ii} is the probability that the current perfect hashing function remains perfect for $(n + 1)$ keys

6. Load factor under the optimal policy

Consider a file being built incrementally by insertion, following the optimal policy outlined in the previous section whenever a rehash is necessary What is the expected load factor? Before being able to answer this question we must first compute the expected load of a group built incrementally

Consider an arbitrary but fixed group When the number of records is less than or equal to b, the group needs only one page When the $(b + 1)$st record arrives, a rehash is necessary The group size will grow to two or three pages, depending on the total number of trials allowed and the actual number of trials required In general, when inserting the $(n + 1)$st record one of two events will occur the record fits into its assigned page or the assigned page overflows In the latter case, a new perfect hashing function must be found and all the $n + 1$ records must be rehashed The optimal policy for finding a hashing function is computed and followed in the search The state transitions that the $(n + 1)$st record may cause are illustrated in Figure 3

Given the probabilities $Q_i(n)$ and the probabilities $P(n, m, b)$, the probabilities $Q_i(n+1)$ can be computed, see the Appendix This involves finding the optimal policy by solving the optimization problem discussed in the previous section Once these probabilities have been found, we can compute the expected load factor This is illustrated in the following table for the case $n = 179, b = 20, t = 10, P = 0\,99$

i	Pages m_0+i-1	Q_i (n)	p_{ii}	Optimal Policy	Q_i $(n+1)$
1	9	000	000	0	000
2	10	047	750	0	035
3	11	573	886	4	540
4	12	264	943	4	298
5	13	057	970	1	062
6	14	028	984	0	028
7	15	022	991	1	026
8	16	008	995	0	008
9	17	002	997	0	003

Expected load factor (n = 180) 77 39%
Probability of rehashing 0 094

Figure 4 shows the expected load factor of a group as a function of the number of records in the group The page size is 50 and t = 5, 10, 20

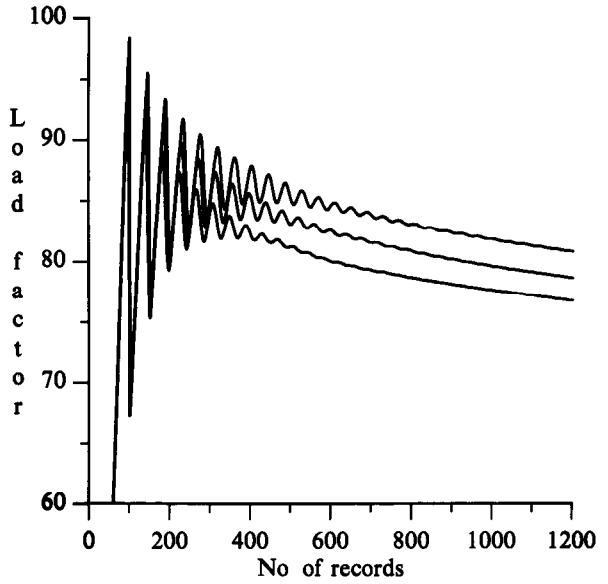

Figure 4: Expected load factor of an incrementally built group with b = 50 and t = 5,10,20

The oscillations in the beginning are expected When there are exactly b records, one page is needed and the load factor is 100% On the other hand, when there are b + 1 records, at least two pages are required and the load factor drops to around 50% The oscillations gradually die out as the number of records increases As the number of records grows the expected load factor decreases, but very slowly This is basically a result of limiting the number of trials allowed for finding a perfect hashing function and the fact that it becomes more and more difficult to find one

The distribution of the number of records in a group can be closely approximated by a Poisson distribution By combining this with the above results we can compute the expected load factor of a file built incrementally under the optimal policy, see the Appendix Figure 5 plots the expected load factor as a function of the expected number of records per group for a few different page sizes The limit on the number of trials is t = 20 in all cases The page size significantly affects the load factor To achieve an expected load factor of 70% or higher, the page size should be 20 or higher

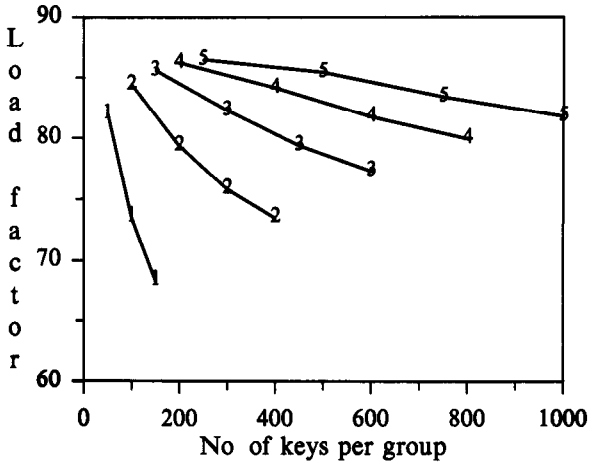

Figure 5: Expected load factor of a file when using the optimal strategy t = 20 and b = 10,20,30,40,50 (plot symbols 1,2,3,4,5)

The cost of an insertion is largely determined by the frequency of rehashing The probability of an insertion causing a rehash of a page group is plotted in Figure 6 The page size is 50 The upper curve corresponds to t = 20, the lower one to t = 10

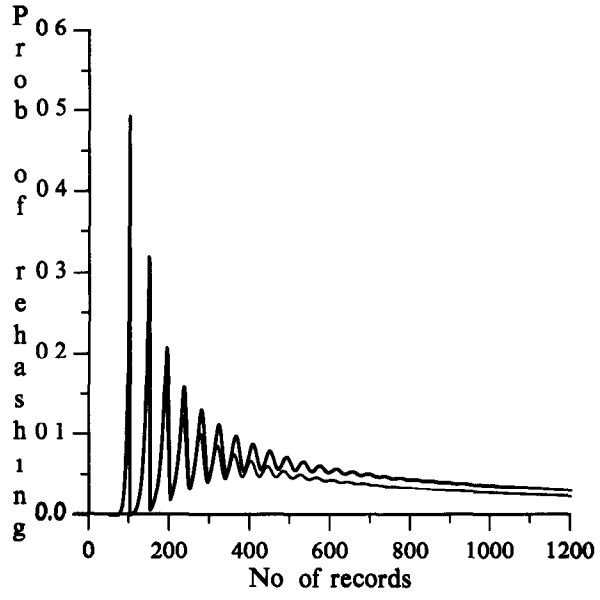

Figure 6: Prob of rehashing a group with $b = 50$ and $t = 10,20$

As far as the achievable load factor and the frequency of rehashing are concerned, the above results strongly suggest that perfect hashing could be of practical use for external files For a file with $b = 50$ and $t = 20$ we can achieve an expected load factor of 82% by having groups of 1,000 records (approximately 25 pages) on average The probability of an insertion causing a rehash is then as low as 0 04, that is, on the average every 25 insertions will cause a rehash

It is clearly impractical to have to solve a dynamic programming problem to find the optimal policy every time a rehash is required Some less expensive way of determining a policy is desirable We have made same preliminary tests of a fairly simple heuristic Its performance, in terms of expected load factor, is quite close to that of the optimal policy Further study of simple heuristics is needed and we will not go into details here

7. Internal storage requirements

To achieve one-access retrieval the header table must be small enough to be stored in internal memory The page size has a significant effect on the size of the header table The larger the page size is, the fewer groups and entries in the header table are needed Furthermore, each group can consist of more pages and/or a higher load factor can be achieved Figure 7 plots the expected load factor (under the optimal policy) as a function of the number of entries in the header table for different page sizes The number of records in the file is 10^6 A page size of 10 records yields a very low load factor even for a large header table For a given page size, the improvements in the load factor become insignificant when the header table size is increased beyond a certain range Consider the case when $b = 50$ A header table of 2,000 entries gives a load factor of almost 85% Doubling the table size to 4,000 entries increases the load factor to 86% only The size of an entry in the header table depends on the number of parameters of the hashing functions used In the next section a class of functions with two parameters is studied and found to perform well Using this class of functions, 12 bytes per entry is sufficient (pointer 3, size 1, parameters 2×4) A header table of 1,000 entries, (yielding a load factor of 81%), then corresponds to 12,000 bytes Further work on the structure of the header table is in progress, and we are confident that the amount of storage needed can be significantly reduced

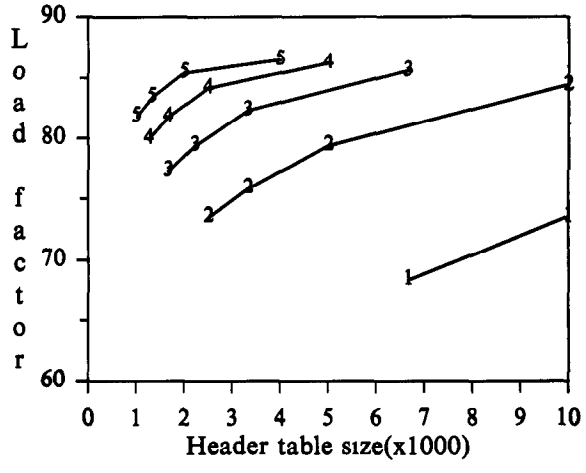

Figure 7: Expected load factor of a file of 10^6 records as a function of header table size $t = 20$, $b = 10,20,30,40,50$ (plot symbols 1,2,3,4,5)

8. A practical class of hashing functions

The analytical results in the previous sections were based on the assumption that the hashing functions are chosen from the set of all functions from n keys to m addresses This is obviously impractical, $n \log m$ bits would be needed to be able to represent any one of the m^n functions A practical hashing function must be simple to evaluate (constant time, no extremely large numbers involved) and require only a small, fixed amount of storage

To gain some experience of the difficulty of finding perfect hashing functions we performed a series of experiments using two real life test files The hashing functions used in the experiments were of the form

$$h(x, c, d) = ((cx + d) \bmod p) \bmod m$$

where c and d are (integer) parameters defining the function, p, is a fixed prime greater than the highest key and m is the number of pages The parameters c and d must both be less than p This class of hashing function was shown to be universal$_2$ by Carter and Wegman [CRTR]

Test data for the experiments were obtained from two files Test file A contained 6,100 words randomly drawn from a 24,000 word dictionary used for spelling checking Test file B consisted of approximately 12,000 userids (max 8 characters) from a large time-sharing installation For simplicity, all parameters and keys were restricted to 15 bit unsigned integers The keys were converted from strings of ascii characters to 15 bit integers by concatenating the 5 least significant bits of three successive characters and taking the exclusive OR of the resulting bit strings (of length 15)

A set of 400 hashing functions was created as follows The maximum key value is 32,767, so p was chosen to be 32,801 (a prime) A hashing function is then completely defined by specifying the parameters c and d A set of 400 (c, d) pairs was randomly generated The standard random number generator RANDOM ( ) supplied with UNIX was used (seed = 314159) The 15 least significant bits of each random number were extracted and used as the value for a parameter

The keys in each test file were divided into 11 groups by the hashing function $h(x) = ((29422x + 25858) \bmod 32801) \bmod 11$

Each experiment was carried out as follows The page size b, the number of pages m, the load factor α and the key group were specified The number of records was then computed as $n = \alpha mb$ and the first n keys from the key group read in The n keys were hashed by each one of the 400 hashing functions and the number of perfect hashing functions was recorded The same was repeated for various values of α, b and m Figure 8 plots the observed relative frequency of perfect functions against the load factor for different values of m The page size is 20 and the test data is from key group 5 of file B The solid lines represent the experimental results and the dotted lines represent the corresponding theoretical results Similar results were obtained for other groups of file B, and also for file A

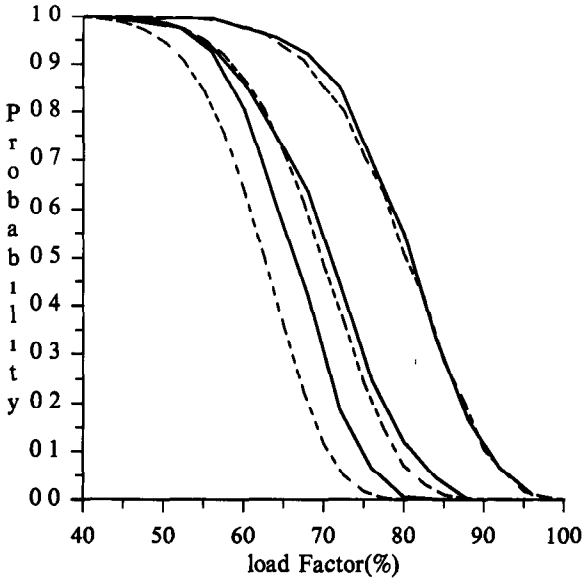

Figure 8: Observed and computed probability of a randomly chosen function being perfect $b = 20$, $m = 5,15,40$
Keys are from group 5 of test file B

The experimental results are close to, or better, than the theoretical results The class of hashing functions considered here is extremely simple and it appears to perform as well as the set of all functions, perhaps even better There are probably many other classes of hashing functions with a similar behaviour We find these first results very encouraging, finding simple classes of hashing functions that perform well in practice does not appear overly difficult Needless to say, further experiments and analysis are needed to substantiate this claim

9. Conclusion

A perfect hashing scheme for large external files has been proposed and analysed The scheme is based on the idea of segmentation an ordinary hashing function is used to divide the records into small groups and a perfect hashing function is defined for each group separately The size, location and the parameters of the hashing function of each group are kept in an internally stored table

We proposed finding perfect hashing functions by repeated random selection from a suitable class of functions The probability that a function randomly selected from the set of all functions is perfect was derived A method for handling the

trade-off between the amount of external storage used and the cost of finding a perfect hashing function was also suggested The resulting expected load factor and rehashing frequency were analysed Finally, we performed a number of experiments with real data and a simple class of hashing functions to find out whether the predicted results can be achieved in practice

In summary, we find the results obtained so far very encouraging It appears possible to design practical perfect hashing schemes for external files based on the suggested approach However, further work is clearly needed to gain a better understanding of the possibilities and limitations of the method

Acknowledgement

We want to thank G V Cormack for many fruitful discussions This work was supported by the Natural Sciences and Engineering Research Council of Canada under grant A2460 and a Commonwealth Scholarship

References

[CCHL]
Cichelli, R J *Minimal perfect hash functions made simple* CACM, 23, 1(1980), 17-19

[CHNG]
Chang, C C *The study of an ordered minimal perfect hashing scheme* CACM 27, 4(1984), 384-387

[CRCN]
Cercone, N , Krause, M and Boates, J *Minimal and almost minimal perfect hash function search with application to natural language lexicon design* Comp & Maths with Appl 9, 1(1983), 215-231

[CRMC]
Cormack, G V , Horspool, R N S and Kaiserswerth, M *Practical perfect hashing* The Computer Journal, 28, 1(1985)

[CRTR]
Carter, L J and Wegman, M L *Universal classes of hash functions* Ninth ACM Symposium on Theory of Computing, (1977), 106-112

[DVS] Davis, F N and Barton, D E *Combinatorial Chance* London Griffin, 1962

[FRDM]
Fredman, M L , Komlos, J and Szemeredi, E *Storing a sparse table with 0(1) worst case access time* Proc of 23rd Symposium on Foundations of Computer Science, IEEE Computer Society (1982), 165-168

[GNT]
Gonnet, G H and Larson, P -A *External hashing with limited internal storage* Proc ACM Symposium on Principles of Database Syst , (Los Angeles, 1982), ACM, N Y , 256-261

[JSCH]
Jaeschke, G *Reciprocal hashing A method for generating minimal perfect hashing functions* CACM, 24, 12(1981), 829-833

[LRSN]
Larson, P -A and Kajla, A *File organization - implementation of a method guaranteeing retrieval in one access* CACM, 27,7 (1984), 670-677

[SPRG]
Sprugnoli, R J *Perfect hashing functions A single probe retrieving method for static sets* CACM, 20, 11(1977), 841-850

Appendix

This appendix summarizes the formulas used for the computations discussed in sections 5-8

Let q_i denote the probability that a randomly chosen hashing function is **not** perfect when there are n keys and $m_0 + i - 1$ pages We have

$$q_i = 1 - P(n, m_0 + i - 1, b)$$

where $P(n, m_0 + i - 1, b)$ can be computed using the recurrence derived in section 4 Let $E(n)$ denote the expected number of pages resulting from a partitioning policy when there are n records The optimization problem then becomes

Minimize $E(n)$,

$$E(n) = m_0(1-q_1^{t_1})+(m_0+1)q_1^{t_1}(1-q_2^{t_2})+$$
$$+ m_1q_1^{t_1}q_2^{t_2} \quad q_{r-1}^{t_{r-1}}(1-q_r^{t_r})+m_1q_1^{t_1}q_2^{t_2} \quad q_r^{t_r}$$

such that

$$t_1 + t_2 + \quad + t_r = t$$
$$t_1\log q_1 + t_2\log q_2 +$$
$$+ t_r\log q_r \geq -\log(1 - P)$$

Given a policy $(t_1, t_2, \ , t_r)$ the probability that a group will consist of $m_0 + i - 1$ pages when there are $n + 1$ pages, $Q_i(n + 1)$, can be computed as

$$Q_i(n + 1) = p_{ii}Q_i(n) +$$
$$R_i(n+1)(1 - \sum_{k=1}^{r} Q_k(n)p_{kk}), \ i = 1,2, \ , r-1$$

and $$Q_r(n + 1) = 1 - \sum_{k=1}^{r-1} Q_k(n)$$

where

$$R_i(n+1) = q_1^{t_1}q_2^{t_2} \quad q_{i-1}^{t_{i-1}}(1 - q_i^{t_i}),$$
$$p_{kk} = \frac{P(n + 1, m_0 + k - 1, b)}{P(n , m_0 + k - 1, b)}$$

Once the probabilities $Q_i(n + 1)$ have been computed we can compute the expected number of pages in a group, and the load factor, as follows

$$GP(n + 1) = \sum_{i=1}^{r} Q_i(n + 1)(m_0 + i - 1),$$
$$GLF(n + 1) = \frac{(n + 1)}{b\ GP(n + 1)}$$

The probability of rehashing while inserting the $(n + 1)$st record into a group is given as

$$RH(n + 1) = 1 - \sum_{k=1}^{r} Q_k(n)\, p_{kk}$$

Let λ be the expected number of records per group ($\lambda =$ total number of records in the file/number of groups) Then the expected number of pages per group, and the overall load factor, can be computed as

$$EP(\lambda) = \sum_{n=1}^{\infty} \frac{e^{-\lambda}\lambda^n}{n!} GP(n),$$
$$FLF(\lambda) = \frac{\lambda}{b\ EP(\lambda)}$$