Spiral Storage: Efficient Dynamic Hashing with Constant Performance

JAMES K. MULLIN
University of Western Ontario, London, Canada

We describe and analyse a dynamic hashing method called 'Spiral Storage'. Dynamic hashing methods extend the power of conventional hashing methods by avoiding the need to have good initial estimates of the storage demand. The file storage space will grow or shrink with demand. 'Spiral Storage' is the only known dynamic hashing method which provides constant average performance while the storage space changes in proportion to the storage demand. The performance of the method with link chained overflow is investigated. Results of analysis and of simulations confirm the utility of the method.

1. INTRODUCTION

We describe a method for hash storage organisation that copes well with changing demand for storage space. Hash-based direct access methods have been used for many years. They provide a mapping between record identifiers and the storage addresses for records in a file. Prior to 1980, one of the major disadvantages of such hashing methods was the need to estimate the storage demand accurately. An overestimate led to wasted space, and an underestimate led to poor performance or – in some cases – failure. If a poor estimate was given, it was necessary to completely reorganise the file into a new appropriately sized storage area. This could be a time-consuming operation for large files. Nevertheless, hashing is attractive. Records can be accessed in (on average) constant time – independent of the number of records in the file. The average is taken across all records in the file. Performance is better than that of competing multi-link tree structures, where the average time to access a record is logarithmic in the number of records. Multi-link tree structures do however provide the ability to expand and contract file space gracefully according to demand.

In about 1980, a variety of hash-based methods appeared which remove the requirement for accurate storage estimates. These methods offered the good performance of hashing coupled with the ability gracefully to expand or contract storage space in proportion to storage demand. Litwin's[1] 'linear hashing' takes its name from the linear relationship between storage used and demand. Other methods are Larson's[2] linear hashing with partial expansions and Martin's 'Spiral Storage'.[3] An excellent survey of dynamic hashing may be found in Scholl.[4]

In this paper we describe and analyze Spiral Storage. Spiral Storage is a dynamic hashing method which provides constant average performance while the storage space expands and contracts in proportion to storage demand. Its creator, Martin, originally presented the concept using properties of the exponential spiral – hence the name. Since Ref. 3 is not generally available, and also in the author's opinion the exposition is simpler without using the exponential spiral, the spiral concept will be abandoned in favour of a more basic approach, while retaining the original name. The basic features of this method will be described, followed by an analysis of an implementation with linked overflow.

2. THE METHOD

A hash function which returns a value in the range (0...1) is required. Performance is best when each value in (0...1) is equally probable. A pseudo-random number generator using the key as seed performed well in the system tested.

Spiral Storage can best be appreciated in two stages: as a mapping of keys to logical addresses and then as a mapping of logical to physical addresses.

key → hash(key) → logical address → physical address

The file is composed of buckets each capable of holding a fixed number of records. Some overflow handling method is required when a bucket overfills. Martin uses double hashing. The implementation described here uses simple linked overflow chains in a separate area. Overflow handling will – for the moment – be ignored.

Spiral Storage provides a systematic way to expand or contract the amount of storage in use – i.e., the size of the logical address space. Consider the case of expanding memory. The file space can be visualised as shrinking on the left and growing on the right. Figure 1 shows three

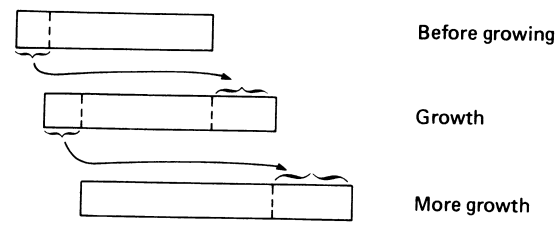

Figure 1. File growth with Spiral Storage. Notice that records move to a larger space.

stages in successive expansions of the virtual memory. When expanding storage, records in a bucket on the left are moved to a new larger space on the right. Thus both file boundaries move. Since many computer systems would have difficulty with a file where both boundaries move, a simple method is employed to re-use space freed on the left. This employs a logical to physical address mapping which will be described later.

3. LOGICAL ADDRESS MAPPING

The logical address mapping function is: logical = $\lfloor b^G \rfloor$. $\lfloor x \rfloor$ denotes the integer truncation of x and $\lceil x \rceil$ denotes

the smallest integer greater than or equal to x. b, called the growth factor, is some chosen constant greater than 1 such as 1.4 or 2. The larger the constant, the more rapidly space expands or contracts. The value of G is given by:

$$G = \lceil c\text{-hash(key)} \rceil + \text{hash(key)}$$

G has a sawtooth shape, as a function of hash(key), depending on the parameter c which determines the initial and final bucket locations. An increase in c will cause an expansion in net space. A decrease in c causes a contraction in space. The value of G ranges from c to $c+1$. Figure 2 shows the relationship between the hash

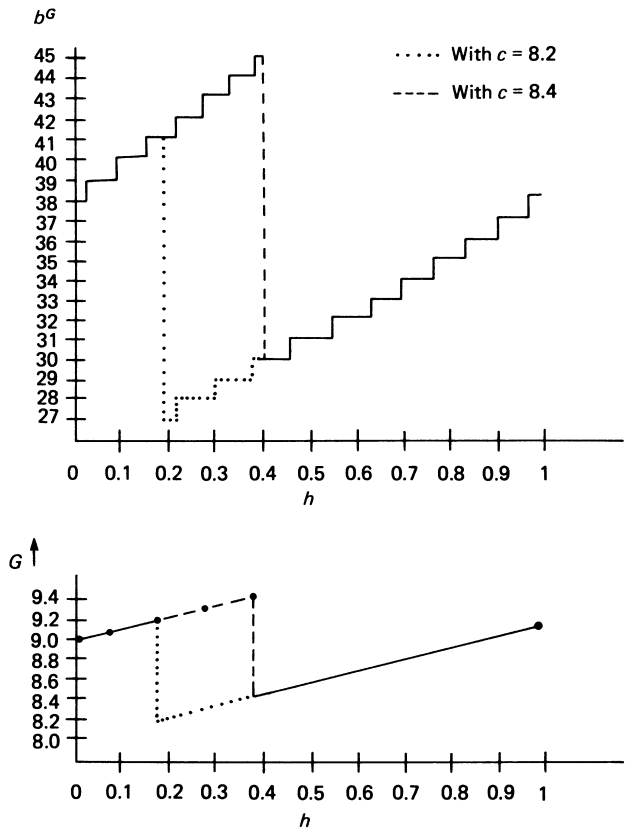

Figure 2. Logical address mapping Using the Sawtooth Function G

function hash(key), G and $\lfloor b^G \rfloor$. When c has the (arbitrarily chosen) value 8.2, follow the solid and the dotted curve. When c is increased to 8.4, follow the solid and the dashed curve.

When $c = 8.2$, the address space varies from 27 to 41. When c increases to 8.4, the addresses range from 30 to 45. Another important property of the function G is also evident in Figure 2. When c changes from $c1$ to $c2$, only the lower addresses from $\lfloor b^{c1} \rfloor$ to $\lfloor b^{c2} \rfloor$ need to change. Other addresses remain the same. It is this property that permits incremental changes in remapping addresses during file expansion. Without such a property, gradual expansion or contraction would not be possible.

Note that the minimum value of G is c and its maximum value is $c+1$. Thus the minimum logical address in use is $\lfloor b^c \rfloor$ and the maximum address is $\lfloor b^{c+1} \rfloor$. The amount of file space used is therefore $1 + \lfloor b^{c+1} \rfloor - \lfloor b^c \rfloor$. This can be approximated by $b^c(b-1)$ when the address space is large.

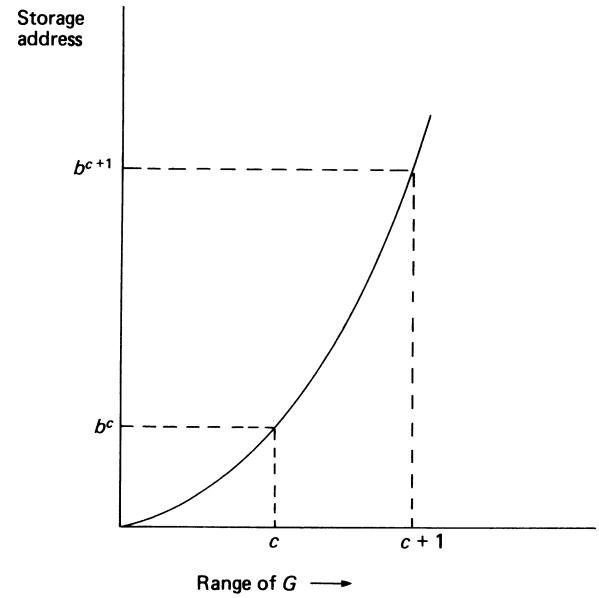

Figure 3. Logical address mapping

Another view of the address mapping can be seen in Figure 3. If the hash function is uniform, each equal-sized interval from c to $c+1$ has the same expected number of keys. In forming the mapping to addresses, note that the addresses near $\lfloor b^c \rfloor$ receive more keys than those near $\lfloor b^{c+1} \rfloor$. The expected number of keys mapping to an address is proportional to the fraction of hash values mapping to that address. This is inversely proportional to the slope of the exponential curve at that address. For an exponential, the slope of the curve is proportional to the address. There are thus unequal bucket expectations. The ratio of highest bucket expectation to lowest is the growth factor b.

4. FILE EXPANSION

With all known hashing methods, performance deteriorates as the storage actually used approaches the storage available. One commonly used measure of storage utilisation is the packing factor. The packing factor, defined as the number of records in the file divided by the product of number of buckets and capacity of a bucket, is monitored and whenever it exceeds (is below) some chosen threshold, we increase (decrease) the space by one bucket. Since performance is related to packing factor, one can control performance. It may be wise to have a threshold region of no change, so small fluctuations about the threshold do not result in excessive remapping.

When we desire to expand the space, a new value of c is chosen so as to completely eliminate the first bucket. This value c' is obtained from the inverse of the exponential function

$$c' = \log_b (first + 1).$$

First is the logical address of the leftmost bucket. Records in the old first bucket are now redistributed to a larger space at higher addresses near $b^{c'+1}$. The expected density of records in the first bucket is the highest in the file, so this is the most desirable bucket to remap. The current value of c must be recorded. The growth function b^G has a simply computed inverse. In addition, the average performance which is determined by the ratio of records

to buckets is kept constant during file growth. This occurs as long as new buckets are added in proportion to new records. Competing methods such as those of Litwin[1] and Larson[2] do not have this attractive property. Unfortunately, since lower addresses are more densely populated than higher addresses, performance does vary with the record accessed. This is also true of Refs 1 and 2.

5. PHYSICAL ADDRESS MAPPING

In the logical address mapping described above, both the left and right file boundaries move. This would cause difficulty in many operating systems such as TOPS-10 or Unix. We distinguish logical and physical addresses in order to keep the view of logical address mapping presented above and yet to map these logical addresses to a contiguous physical space where only the right boundary moves. Martin[3] shows such a method to re-use released buckets on the left of the file. In the implementation described here, whenever the packing factor exceeds a threshold, the leftmost bucket is totally removed by a suitable adjustment to c. The important idea is to immediately re-use this bucket for a new logical address and then to allocate new buckets as required.

Each time a logical bucket is accessed, one must determine its actual physical address. This requires a determination of how the given logical address was instantiated. There are three possible cases:

it was one of the original allocation;
it was a re-use of a freed bucket on the left;
it was a newly allocated bucket.

It is convenient to ignore the first case. We consider the file space to have been initially empty and all buckets to have been assigned on expansions from the null file.

Given a logical bucket address, we wish to determine whether it was put into a recycled or a newly allocated bucket. If the bucket was put into newly allocated space, its physical address is one plus the number of buckets existent just before its creation. If the bucket was put into recycled space, one determines its recycled ancestor and then recurs to find the first allocation for the ancestor bucket. This process involves finding the fractional bucket addresses, called the ancestor range, which when deallocated could map keys to the logical bucket under consideration. Figure 4 shows this process graphically. Consider bucket k. The range of keys mapping to bucket k is from the lower boundary at $\log_b k$ up to $\log_b(k+1)$. The range of keys which will be remapped into bucket k lies at boundaries one below from $\log_b k-1$ to $\log_b(k+1)-1$. This is because the active keyspace range is always one unit long from c to $c+1$. One can map from G back to the bucket ancestor addresses.

$$\text{lowest ancestor} \ldots \text{low} = b^{\log_b k-1} = k/b$$
$$\text{highest ancestor} \ldots \text{high} = b^{\log_b (k+1)-1} = (k+1)/b$$

The actual lower bucket is ⌊low⌋ and the highest is ⌊high⌋. On the graph, the lowest truncated boundary is 1 and the highest is 2. Since the lower ancestor boundary (at = 1.6) is not precisely at a bucket boundary, bucket 6 must have been instantiated from newly allocated space.

Note that if $b > 1$, the ancestor range mapping to a bucket is always smaller than one unit bucket. This is true since a deallocated logical bucket is always mapped to a new larger space. When a bucket on the left is deallocated, the entire key range in that bucket is mapped elsewhere. The new value of c is $\log_b(\text{leftmost}+1)$. When

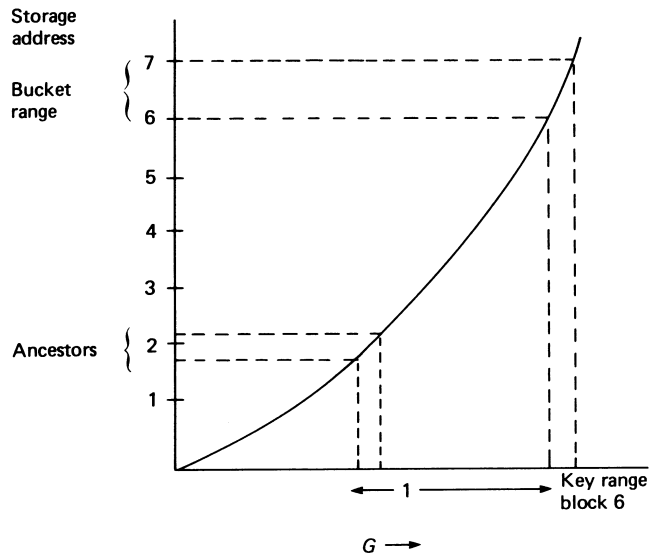

Figure 4. Physical address mapping determining ancestor blocks

a bucket is totally deallocated, that space can be (and is) immediately re-used. We need to determine whether the low ancestor address and high ancestor address are within the same bucket. If this is so then bucket k was instantiated when the bucket at ⌊low⌋ was released. Since buckets always expand to a larger space, high − low < 1. Thus if ⌊high⌋ > ⌊low⌋, then bucket k was instantiated from the recycled bucket ⌊low⌋. Otherwise, bucket k was instantiated from newly allocated space.

If bucket k (one with ⌊low⌋ = ⌊high⌋) was instantiated with a newly allocated bucket, one needs to know the number of active locations when it was instantiated. This is given by k-⌊low⌋, since k was then the last and low was the initial bucket at that time.

If the bucket was instantiated from a recycled bucket, the problem reduces to finding how this recycled bucket (at address low) was instantiated. The address is always reduced and the recursion completes.

Despite the tricky logic involved, the mapping algorithm is simple.

```
function physical(logical)
  high: = ⌊(1+logical)/b⌋
  low: = ⌊logical/b⌋
  if low < high
    then physical: = physical(low) (*recycled*)
    else physical: = logical−low+origin;
end physical;
```

'Origin' is the physical address of the initial bucket – usually zero. The recursion, which can be easily replaced with a whole loop, is limited to $\log_b(\text{buckets})+1$ cycles.

6. ANALYSIS OF PERFORMANCE

The Spiral Storage allocation method was analysed and simulated in order to study its behaviour. The overflow handling method used in this study is link chaining of overflow records to the prime bucket. Space is allocated in buckets each with a capacity of d records. Each bucket has a possibly empty overflow chain where unblocked records are attached to the bucket in a linked list. Many other conflict handling methods are possible. Martin[1] uses a double hashing method. As his method needs special additional provisions to handle unsuccessful

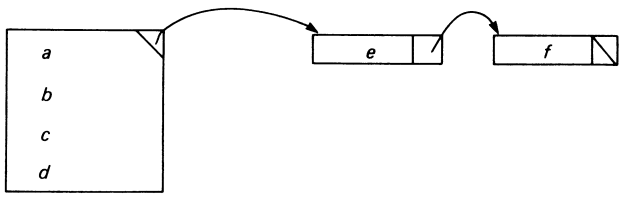

Figure 5. Storage layout of a bucket. The blocking factor *d* is 4.

Table 1. Analytic and simulation results, spiral storage with linked overflow chains

Parameters			Calculated			Simulated		
b	d	pf	Ss	Avcl	Su	Ss	Avcl	Su
1.4	10	0.7	1.05	0.30	0.68	1.05	0.16	0.69
1.4	10	0.8	1.09	0.55	0.76	1.09	0.36	0.77
1.4	10	0.9	1.16	0.88	0.83	1.18	0.80	0.83
2.0	10	0.7	1.06	0.35	0.68	1.09	0.28	0.68
2.0	10	0.8	1.11	0.57	0.76	1.14	0.50	0.76
2.0	10	0.9	1.17	0.81	0.83	1.29	1.01	0.81
3.0	10	0.8	1.15	0.69	0.75	1.35	0.90	0.73
2.0	20	0.8	1.08	0.55	0.78	1.11	0.48	0.78

searches, the simpler chaining method implemented is described here. Figure 5 shows the storage layout of a typical bucket. Table 1 summarises some results from the simulations. It is discussed later.

First, we analyse the method mathematically. We assume that the hash function used returns a real result in the range (0...1) and that each equal-sized interval in this range has the same expected key density. We shall calculate the expected number of access for a successful search and for an unsuccessful search. A search of the prime bucket and each overflow chain record read counts as one access. These calculations will be done in terms of a measure of space used. There are two measures commonly employed: packing factor and storage utilisation. Some definitions are now required.

Let pf = packing factor, su = storage utilization, nrec = number of records in the data space, d = blocking factor, avcl = average overflow chain length.

$$\text{pf} = \text{nrec}/(\text{prime\_buckets}*\text{d})$$
$$\text{su} = \text{nrec}/((\text{prime\_buckets}*(\text{d}+\text{avcl})))$$

The implementation expands space so as to maintain a constant packing factor. We thus choose to employ pf in the analysis. As the analysis calculates avcl, we can compute the storage utilisation if this is desired.

The expected number of records in a bucket varies across the data space. Looking at Figure 3, note that $G = \lceil c\text{-hash}(k)\rceil + \text{hash}(k)$ varies from c to $c+1$, with each equal-sized interval having an equal expected number of keys. Also note that the expected number of records in a bucket is inversely proportional to the slope of the exponential curve b^G at that bucket. Consider h to be a point in the range from c to $c+1$; h covers the active addresses. The expectation x in a bucket is given in terms of the maximum expectation in the leftmost bucket: max x.

$x = \max x/b^h$; h in 0...1; when $h = 0$, $x = \max x$; when $h = 1$, $x = \max x/b$.

Now, solving for pf in terms of b, d and max x,

$$\text{pf} = \text{nrec}/\text{prime\_buckets}*\text{d}.$$
$$\text{prime\_buckets} = \lfloor b^{c+1}\rfloor - \lfloor b^c\rfloor + 1 \approx b^c(b-1).$$

The approximation ignores the 1 and truncation effects. It will be valid with a large number of buckets.

The number of records, nrec, is the integral of the expectation x across all buckets. $\text{nrec} = \int_{b^c}^{b^{c+1}} x\,ds$; where s is the address space. Since $x = \max x\ b^c/s$, when s is the address b^c, $x = \max x$. Thus, $s = \max x\ b^c/x$. The differential $ds = -\max x\ b^c/x^2\ dx$. Now, nrec $= \int_{\max x}^{\max x/b} \max x\ b^c/x\ dx$. Integrating across expectation, nrec $= \max x\ \ln(b)\ b$, and hence pf $= \max x\ \ln b/(d(b-1))$. This permits the calculation of max x given pf, d and b: $\max x = \text{pf}\ d(b-1)/\ln b$.

We now wish to calculate the average number of accesses for searching. To do this, the probabilities of various overflow chain lengths are needed. If x occurrences of a random event are expected, the probability of actually finding k occurrences is given by Poisson's law:

$$P[nr = k] = x^k e^{-x}/k!$$

Let the probability of overflow chain lengths be $P[of = k]$.

$$P[of = k] = P[nr = d+k]; \quad k > 0$$
$$= x^{d+k}e^{-x}/(d+k)!; \quad k > 0.$$

The expectation x varies from bucket to bucket. To find the average, we must integrate across all buckets in the address space. This expectation was previously found to be $\max x/b^h$, when h ranges from 0 to 1.

Let $\bar{P}$ be the average of P over all buckets.

$$\bar{P}[of = k] = \int_0^1 x^{d+k}e^{-x}/(d+k)!\ dh; \quad k > 0.$$

Substituting for x and removing terms which do not vary with h from the integral, we find:

$$\bar{P}[of = k] = \frac{\max x^{d+k}e^{-\max x}}{(d+k)!} \int_0^1 (b^{-h})^{d+k}\ e^{-b^h}\ dh;\ k > 0.$$

This was evaluated numerically. Once the $\bar{P}$ are calculated, one can use them to find the expected search accesses which determine efficiency.

6.1 Failed Search: *Fs*

Here we calculate the expected number of accesses needed to determine that a key is not in the file. For overflow chains of length k, there are $k+1$ total accesses. The prime bucket and each of the k overflow blocks must be read. In the following formula each possible chain length is weighted by its probability of occurrence.

$$Fs = \sum_{k=0}^{\infty} \bar{P}[of = k]\ (k+1).$$

In reasonable operating ranges, ∞ can be safely approximated with 20. This was checked. $P[of = 0]$ need not be calculated as the sum of all probabilities must be one. Since Fs equals 1 + avcl, we can also find the average overflow chain length.

This result is very important in practice, since each insertion must effectively do an unsuccessful search to guard against duplicate entries in the file.

6.2 Successful Search: *Ss*

Here we calculate the expected number of accesses to locate an entry which is in the table. For each possible length of overflow chain, one weights the probability of that length of overflow chain with the expected number of accesses to find the record sought. This record is assumed to be equally probable in any position – both the d positions in the prime bucket and the k positions on the overflow chain.

$Ss = \bar{P}[of = 0] + \bar{P}[of = 1]$
$(d/(d+1) + 2/(d+1)) + \bar{P}[of = 2]$
$(d/(d+2) + 2/(d+2) + 3/(d+2)) + \ldots + \bar{P}[of = k]$
$(d+2+3\ldots+k+1)/(d+k)$.

Table 1 shows some results of the calculations and is compared to values from the simulations. Reasonable agreement is found.

7. DISCUSSION

The results show Spiral Storage to be an excellent dynamic hashing method. Its major advantage over competing methods (see Refs 1 and 2) is that performance does not vary cyclically during file growth or shrinkage.

One of the factors influencing performance is the growth factor b. A high growth factor increases search time. However, our results show that the effect of b on search time is not as great as might be anticipated. The effect can be seen in Table 1. The advantage of a high growth factor is less work during file expansion. With a growth factor of b, every time the space grows by $(b-1)$, every bucket in the file must be moved. If $b = 1.4$, growth by 40 per cent requires a complete readjustment of addresses. This is a severe overhead. When no good initial space estimate is available, a value for b of 2 or greater would be preferable.

Acknowledgments

The author wishes to thank P. Larson of the University of Waterloo for an interesting discussion of spiral storage, Julian Davies of the University of Western Ontario for a critical reading, and the Natural Science and Engineering Research Council of Canada for supporting this work with an operating grant.

REFERENCES

1. W. Litwin, Linear hashing: a new tool for file and table addressing. *Proceedings of the 6th International Conference on Very Large Databases, Montreal*, 1980, pp. 212–223.
2. P. Larson, Linear hashing with partial expansions. *Proceedings of the 6th International Conference on Very Large Databases, Montreal*, 1980, pp. 224–232.
3. G. N. Martin, *Spiral Storage: Incrementally Augmentable Hash Addressed Storage*. University of Warwick Technical Report 27, March 1979.
4. M. Scholl, New file organizations based on dynamic hashing. *ACM TODS*, **6** (3), 194–211 (1981).