An Algorithm for Dynamic Storage Allocation and Its Application to B$^+$-trees

Ricardo A. Baeza-Yates
Data Structuring Group
Department of Computer Science
University of Waterloo
Waterloo, Ontario
Canada N2L 3G1 *

July 10, 1987

Abstract

We derive a practical algorithm for dynamic storage allocation based on a theoretical result. We apply this algorithm to disk storage allocation of B$^+$-trees with partial expansions. Simulation results show that this algorithm works well for this case.

1 Introduction

The dynamic storage allocation problem consists in a storage of consecutive cells (memory) of size S and an allocation algorithm that answers requests for allocating new blocks or freeing blocks to the storage. Also a set of possible block sizes $\{b_1, b_2, \cdots, b_n\}$ is given, such that $1 \leq b_1 < b_2 < \cdots < b_n$.

The problem is to find a good strategy for the allocation algorithm to minimize the wasted space (*fragmentation*) in the storage. Given an upper limit M for the total number of occupied cells at any time, this problem can be stated as: What is the minimum number of cells $S(M, \{b_1, b_2, \cdots, b_n\})$ with which an allocation algorithm can answer any sequence of requests?

In [9] it was proved that the limit

$$\lim_{M \to \infty} \frac{S(M, \{b_1, b_2, \cdots, b_n\})}{M}$$

exists. We will call this limit $S(\{b_1, b_2, \cdots, b_n\})$. The case $b_j = 2^{j-1}$ was studied by Robson [10], who found that

$$S(\{1, 2, \cdots, 2^r\}) = 1 + \frac{r}{2}.$$

*This work was also supported by the University of Chile, Santiago, Chile

Krogdahl [6] extended this result for the case in which each of the ratios b_j/b_{j-1} is an integer, showing that

$$S(\{b_1, b_2, \cdots, b_n\}) = 1 + (1 - \frac{b_1}{b_2}) + \cdots + (1 - \frac{b_{n-1}}{b_n}).$$

In this paper we extend the latter result, showing that this result is also an upper bound for any set $\{b_1, b_2, \cdots, b_n\}$. From this we derive a practical algorithm for this problem. In the last section we apply this algorithm to the disk storage allocation of B$^+$-trees with partial expansions.

2 A General Upper Bound

The next theorem is based on [6].

THEOREM 2.1 *For any set of possible sizes $\{b_1, b_2, \cdots, b_n\}$ such that each b_i is an integer and $1 \leq b_1 < b_2 < \cdots < b_n$ then*

$$S(\{b_1, b_2, \cdots, b_n\}) \leq 1 + \sum_{i=2}^{n}(1 - \frac{b_{i-1}}{b_i}).$$

Proof: Let M be a multiple of $m = MCM(b_1, b_2^2, \cdots, b_n^2)$, where $MCM(a, b)$ means the minimum common multiple of a and b. We shall show that there is an algorithm that can meet any sequence of requests with a storage of at least size $M(1 + \sum_{i=2}^{n}(1 - \frac{b_{i-1}}{b_i}))$. Since we know that the limit exists, this will show that the statement of the theorem is true.

The storage is divided into n pieces, P_i, such that the size of each piece s_i is

$$s_i = \begin{cases} M & i = 1 \\ (1 - \frac{b_{i-1}}{b_i})M & 2 \leq i \leq n \end{cases}$$

Note that each s_i is a multiple of any b_j, for any i and j. A request for a b_j block is always placed in any one of the pieces $P_1, \cdots, P_j$ and always an integer multiple of its own length away from the boundary of the selected piece. This implies that the smallest block ever allocated in piece P_j is of size b_j. Assume that a b_k block cannot be allocated. Then each of the P_i $(i = 1, \cdots, j)$ pieces is occupied by at least b_i blocks. This then means $b_i P_i / b_k$ occupied cells in each P_i. Hence, the total number of occupied cells is

$$\frac{b_1}{b_k}M + \frac{(1 - \frac{b_1}{b_2})b_2}{b_k}M + \cdots + \frac{(1 - \frac{b_{k-1}}{b_k})b_k}{b_k}M = M.$$

This means that the request is illegal. ■

We were not able to prove that this upper bound is the exact result. However, taking in account that the problem is more difficult for a general set $\{b_1, b_2, \cdots, b_n\}$, we conjecture that this upper bound is in fact an equality.

Note that this upper bound in the amount of space gives a lower bound in the storage utilization (that is the reciprocal of S/M). As a special case, we have that if any integer size between 1 and n is allowed, then

$$S(1, 2, \cdots, n) \leq H_n = \ln n + O(1),$$

where $H_n = \sum_{i=1}^{n} 1/i$ are the harmonic numbers. This result proves half of the conjecture raised in [10].

3 The Allocation Algorithm

In practice, it is possible to implement the algorithm as presented in the previous section. However, the implementation could be difficult and expensive. In addition, there are some aspects of the algorithm not completely defined. For example, if there exist more than one place for a new block, which one we must use.

The first observation is that for the purposes of the proof, M is in general, a multiple of a very large number. This is not necessary, since each piece P_i is also a multiple of $B = MCM(b_1, b_2, \cdots, b_n)$. Then, instead of having n pieces of different sizes, we can divide the storage into many pieces of equal size B, and the allocation algorithm will be the same for each piece. The amount of initial storage available must be a multiple of B, and in general M will be much bigger than B. Note that we are not using the condition that a block of size b_i must be allocated in only some part of the storage. The reason for this is that without this condition the algorithm is simpler and on average the behaviour will be similar.

A request for a b_j block is placed in any of these pieces and always an integer multiple of its own length away from the boundary of the selected piece. In other words, the blocks are boundary aligned. A similar idea is used in the ***best fit aligned*** allocation method [4].

To find free space efficiently and to know if a request to free a block is valid, we need additional information. For this, we associate with each piece some status information that we will call the *state* of the piece. The possible number of states is the number of possible different allocation patterns in one piece. This number is less that 2^B, so we need less than B bits to represent the state of each piece, and therefore we need less than 1 bit for each cell of the storage. For example, if each cell is a 4-byte word, we need 1 bit for each 4 bytes.

There are two obvious ways to describe the states of all the pieces. The first is to have a bit map, storing sequentially the state of each piece. To find a free space we search linearly in the bit map for a piece in a state that has space for the block requested. To improve the worst case, it is possible to maintain n pointers to the first piece that has space for a block of size b_j. The search will be intrinsically linear, and the space needed will be $O(M)$, but with a small constant much less than 1.

The second way is to have a list for each state with the addresses of all the pieces in that state. To find free space we search in the lists of states having space for the requested block. In the worst case this search can be linear in the number of states, but usually will take $O(1)$. For this we need 2 words per piece, that is space $O(M/B)$.

Whatever we use for the description of the piece, the selection of the first adequate piece will be like a first fit algorithm. If we have more than one choice, we can have a best fit algorithm. In this case best fit will mean *best state*. For example we could choose the state more filled, or the state that has more blocks of the same size, etc. After choosing which piece, we need to decide where in that piece we put the new block if we have more than one choice. For example, the best fit aligned position. These *policies* for allocating a new block will depend on the application.

Having selected the location of the new block, we need to compute the new state of the piece. The same is true if we are freeing a block. There are 2 possibilities: we compute the new state or we have a transformation table which, given a state and an action, yields the new state.

The last problem, concerns to what happens if we cannot answer a request. We have two solutions. The first one is compaction. However, for compaction we need to know all the pointers to the pieces. This means that we need an additional data structure for translating the pointers of the application to our pointers. In that way we can move our pointers without change the pointers of the application. In general, compaction is expensive, and the results are often not worthwhile [4]. For this reason we recommend the next solution.

The second solution is to obtain more free space. This is very easy. We need only to request a piece of size B contiguous to our storage of size M. This means that our storage can grow incrementally, starting from one piece to many as we need. This also provides a way to use this algorithm over other storage environment. We have a linked list with all the current used pieces, and each time that we need more space we allocate a new block of size B. This is particularly suitable if we want to have a better allocation algorithm than the one provided by the environment; moreover we only need an storage environment that provides a good allocation strategy for blocks of a unique size. With the same structure, compaction is also possible, because the linked list is the structure that transforms the pointers of the application to our pointers.

This algorithm is a general setting for the problem of dynamic storage allocation with many possible sizes. We think that in many cases it is a good choice, and we demonstrate this in the next section for a particular case.

4 An Application

4.1 B$^+$-trees with Partial Expansions

A B-tree is a balanced multiway search tree. We define a B-tree of order m as follows:

1. The root has between two and $2m+1$ descendents.

2. Any other internal node has between $m+1$ and $2m+1$ descendents.

3. All the leaves are at the same level.

In a B$^+$-tree all data records are stored at the lowest level (*buckets*), and the upper levels are a B-tree index to the data buckets [5]. All the buckets are of the same size. File growth is handled by bucket splitting, that is, when a bucket overflows, an additional bucket is allocated and half of the records from the overflowing bucket are moved to the new bucket. The same method is applied to index nodes. The average storage utilization of a B^+-tree is asymptotically 69% [1].

To improve the storage utilization of the B^+-tree some overflow techniques have been proposed (see [5] for a summary). One of the best known are B^*-trees. In this case, if we have an overflow, first we check at its adjacent brothers for free space. If there is space, some records are shifted and a split is avoided. If not, it means that at least one other bucket is in overflow. Then, an additional bucket is allocated, and the records of the two overflowing buckets are distributed in three buckets. With this the minimum storage utilization is 67%. However, the insertion time is greater than the simple case.

In a recent paper [8], Lomet proposed the use of *elastic buckets*. The idea is simply to increase the size of an overflowing bucket instead of splitting it. Each expansion step is called a *partial expansion* and a bucket is expanded until it reaches some predetermined maximum size. When a bucket of maximum size overflows, it is split into two buckets (of minimum size) in the same way as for a standard B^+-tree. This process of a bucket gradually growing from its minimum size until it splits, is called a *full expansion*. A full expansion increases the number of buckets by one. The idea of partial expansions has previously been applied to Linear Hashing [7].

Let r be the number of expansion steps required to grow a bucket from its minimum size up to and including the final split. For simplicity we assume that records are of fixed length and measure the bucket size (capacity) in number of records. Let the r different bucket sizes be $s_0, s_1, ..., s_{r-1}$, where $s_0 < s_1 < ... < s_{r-1}$. We call this the bucket *growth sequence*. In principle, any strictly increasing sequence is a valid growth sequence, provided that $2s_0 \geq s_{r-1} + 1$. However, due to hardware and software limitations, the following growth sequence seems most practical [7]. Let a *page* be the smallest unit of transfer between disk and main memory, and a *page block* some fixed number of consecutive pages. Assume that the capacity of a page block is b records. For simplicity, we assume that b is integer. Then choose the growth sequence rb, $(r+1)b$, $\cdots$, $(2r-1)b$. In other words, the minimum bucket size is rb records and each partial expansion increases the bucket size by one page block, up to the maximum bucket size of $(2r - 1)b$. The minimum possible storage utilization is $r/(r + 1)$ (in the first expansion).
Example: For page blocks of size 5, and 3 partial expansions, the growth sequence is 15, 20, 25. When inserting the 26th record into a bucket, it splits into two buckets of size 15, each one containing 13 records. The minimum storage utilization for the various bucket sizes is $13/15 = 0.866...$, $16/20 = 0.8$, $21/25 = 0.84$.

For a complete analytical and empirical analysis of this trees see [2]. All the theoretical results mentioned in this section are from the previous reference. All the results are asymptotical in the number of records inserted in the tree (no deletions). For example, the asymptotic average storage utilization for large buckets for B^+-trees with r partial expansions is [2]

$$U = \frac{\ln 2}{H_{2r-1} - H_{r-1}}$$

and one advantage is that the insertion time is only a little more expensive than that of simple B^{+}-trees. However, the disadvantage is that disk space management is more difficult.

4.2 Disk Storage Management

For the disk space management we apply our algorithm. The basic cell unit will be a page block, and if we are using r partial expansions, the set of allowed blocks is $\{r, ..., 2r-1\}$. Hence, the piece size B will be the minimum common multiple of all these numbers. Clearly, $B \leq r(r+1)\cdots(2r-1)$. Now we are interested in the total storage utilization in the used pieces. Applying the lower bound of the first section, and using the minimum storage utilization in a bucket, we obtain a lower bound for the total storage utilization. This lower bound is the worst case of the original algorithm. Similarly, the average total storage utilization is bounded from above by the storage utilization in the B^{+}-tree. Hence, if U_t is the average total storage utilization, we have

$$\frac{r}{(r+1)(1+H_{2r-1}-H_r)} \leq U_t \leq U$$

For large r, this becomes

$$\frac{1}{1+\ln 2} + O(1/r) \leq U_t \leq 1 - O(1/r)$$

The value of the lower bound is 59%.

Which value of r is the best? From the point of view of internal storage utilization, r must be as large as possible. However, we have some drawbacks. First the size of B grows with r and then each new piece allocated significantly changes the storage utilization. In other words, the transient phase will be longer and unstable. Second, the external fragmentation grows as we have more sizes. We have selected $r = 2$ for testing the algorithm, as a compromise between storage utilization and the dynamic storage management.

With two partial expansions, we divide the disk storage in pieces of size $B = 6$ page blocks. When more space is needed for the file, an additional piece is requested from the operating system. A piece stores either one, two or three buckets. Denoting the relative bucket sizes by 2 and 3, the following classification of the states of a piece is sufficient for our purpose:

- empty, 2+0, 0+2+0, 0+2, 3+0, 0+3, 2+2+0, 2+0+2, 0+2+2, 2+3, 3+2, 2+2+2, 3+3

The notation 2+0 means that a bucket of size 2 occupies the two leftmost page blocks and the rest are free. 2+0+2 means that there is free space in the middle of the piece. An in-core table is used for keeping track of the state of each piece of the file. We must be able to distinguish between the states listed above. Hence this scheme requires 4-bit entries in the table (*bit map*). From this table we can find out not only whether a piece has room for a new bucket, but also the exact size and location of the free space within the piece. We can therefore write out a new bucket directly without first having to read in the piece. Even for a large file, the bit table is small enough to be kept in main memory. For example, if we have 50 million records, we will need approximately 335K pieces if a page block has a capacity of 30 records (small bucket size 60). This means a bit map of approximately 170K. The fact that the search for free space is linear is not important given the difference in access time between primary and secondary memory.

Space may now be wasted because of internal and external fragmentation. Internal fragmentation refers to the space wasted within buckets. External fragmentation refers to the space wasted because pieces are not completely filled.

Now, we must define the allocation policy for this particular case. The algorithm searches for pieces with free space in a certain order. To speed up the search we maintain a pointer to the first piece in each state. The order used was (depending on the size requested)

Size 2: 0+2+0; 2+2+0 or 2+0+2 or 0+2+2; 2+0+0 or 0+0+2; 0+3 or 3+0

Size 3: 0+3 or 3+0; 0+0+2 or 2+0+0

That is, when space is needed for a bucket of size 2, we first try to find a piece in state 0+2+0. If no such pieces exist, we try to find a piece in one of the states 2+2+0, 2+0+2, or 0+2+2, and so on. When we find space we can answer the request immediately, and we can update the pointers in parallel with the system.

4.3 Analysis

In general, the analysis of dynamic storage allocation algorithms is very difficult (see [3]). However, in this case we have more information. We only have two types of events. If a block of size 2 is freed, then a block of size 3 will be requested. If a block of size 3 is freed, then two blocks of size 2 will be requested.

With this, it is possible to obtain a good lower bound for the average total storage utilization. For this, we follow the same allocation schema, with the restriction that all the requests are resolved within the same piece. If there is no more space in a piece, then a new piece is allocated. The result will be a lower bound, because we not use the free space available in other pieces.

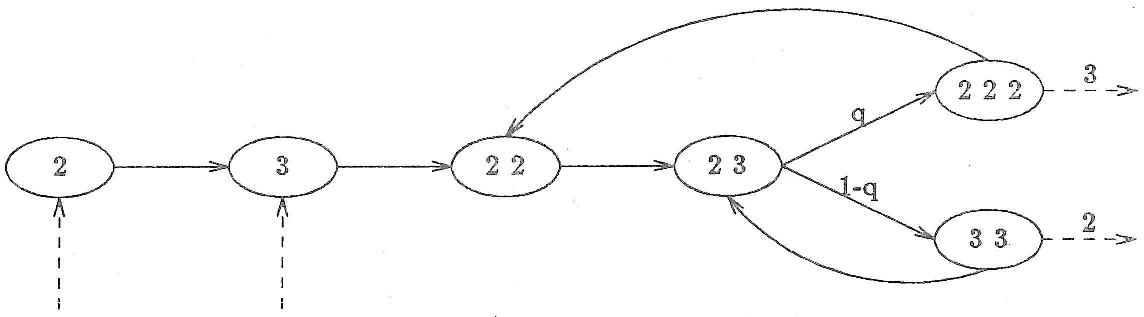

Figure 1: Transition Diagram of the Allocation Process in a piece

Figure 1 shows the transition diagram that represents the possible states of a piece and the allocation process. The dashed arrows represent the creation of new pieces. The probability q represents how frequently a block of size 3 is transformed into two of size 2 (split). Hence, the probability of a partial expansion is $1-q$. We will reach a steady state, asymptotically in the number of records inserted. If p_i is the probability of a piece being in state i we have

$$p_2 = 0, \quad p_3 = p_2, \quad p_{22} = p_3 + p_{222}, \quad p_{23} = p_{22} + p_{33}, \quad p_{33} = (1-q)p_{23}, \quad p_{222} = q\, p_{23}$$

and, of course,

$$p_2 + p_3 + p_{22} + p_{23} + p_{33} + p_{222} = 1.$$

We need another equation to obtain the value of q. For this, we use the ratio between buckets of size 2 and buckets of size 3. Let $f_2(b)$ and $f_3(b)$ be the asymptotic fraction of buckets of size 2 and 3, respectively, when the page block size is b. Then

$$\frac{p_2 + 2p_{22} + p_{23} + 3p_{222}}{p_3 + p_{23} + 2p_{33}} = \frac{f_2(b)}{f_3(b)} = g(b).$$

From [2] we have

$$f_2(b) = \frac{b+1}{2b+1} \quad \text{and} \quad f_3(b) = \frac{b}{2b+1}.$$

So $g(b) = \frac{b}{b+1}$. Solving the system we obtain

$$p_2 = 0, \; p_3 = 0, \; p_{22} = \frac{2b+3}{16b+7}, \; p_{23} = \frac{7b+2}{16b+7}, \; p_{33} = \frac{5b-1}{16b+7}, \; p_{222} = \frac{2b+3}{16b+7}$$

and $q = \frac{2b+3}{7b+2}$. For large bucket sizes (or page block sizes) q converges to $2/7$.

Let $u_2(b)$ and $u_3(b)$ be the asymptotic storage utilization on buckets of size 2 and 3 respectively. Then, the total storage utilization in one piece is bounded by

$$\begin{aligned} U_t &\geq (p_2 + p_{22} + \frac{p_{23}}{3} + p_{222})u_2(b) + (p_3 + \frac{p23}{2} + p_{33})u_3(b) \\ &= \frac{(38b+40)u_2(b) + 51b u_3(b)}{6(16b+7)} \end{aligned}$$

As already mentioned, an upper bound on the total storage utilization is the internal storage utilization of the buckets. For $r = 2$ [2] this value for a page block size b is

$$U_t \leq \frac{(2b+1)(3b+1)}{b(5b+2)}\alpha(b)$$

with

$$\alpha(b) = \begin{cases} H_{3b+1} - H_{(3b+1)/2} & b \textit{ odd} \\ H_{3b} - H_{3b/2} & b \textit{ even} \end{cases}.$$

From [2] we know that

$$u_2(b) = \begin{cases} \frac{(2b+1)(3b+1)}{2b(b+1)}(H_{2b+1} - H_{(3b+1)/2}) & b \textit{ odd} \\ \frac{(2b+1)(3b+1)}{2b(b+1)}(H_{2b+1} - H_{3b/2} - \frac{1}{3b+1}) & b \textit{ even} \end{cases}$$

and

$$u_3(b) = \frac{(3b+1)(2b+1)}{2b^2}(H_{3b+1} - H_{2b+1}).$$

Hence, for large bucket sizes we have

$$\frac{21}{16}\ln 2 - \frac{1}{8}\ln 3 + O(1/b) \le U_t \le \frac{6}{5}\ln 2 + O(1/b)$$

or

$$.7724 + O(1/b) \le U_t \le .8318 + O(1/b).$$

4.4 Simulation Results

In this section, simulation results are presented. Figure 2 shows the development of the average total storage utilization for smallest bucket size 60. The solid line represent averages from 100 simulated file loadings. The top line represents the internal storage utilization (upper bound) of the B$^+$-tree with 2 partial expansions. The bottom line represents the theoretically expected storage utilization for a standard B$^+$-tree with bucket size 60. Initially the file consisted of completely filled smallest buckets. This initial state was deliberately chosen to see how the schemes perform when there is heavy expansion and splitting activity, and when there is a significant excess of buckets of one type.

It is apparent from figure 2 that the fragmentation is very little, and that the total storage utilization out performs the simple B$^+$-tree. For this algorithm, the lowest storage utilization occurs when there has been an excess of bucket of size 3 and they start splitting. When there is an excess of buckets of size 3, there will be many buckets in state 3+3. When these buckets start splitting, a significant fraction of buckets in state 2+3 (or 3+2) will be created, wasting 1/6 of a piece. This is further exacerbated by the low internal storage utilization in the newly created buckets of size 2.

Table 1 shows simulation results and the theoretical upper bounds for large files and two different smallest bucket sizes. Note that the upper bound cannot be achieved without relocation of buckets. In the steady state there is still expansion and splitting activity going on. When a bucket is expanded or split, a "hole" may be created which then will persist for some time until it is eventually filled. At any given point in time, a few such "holes" may exist. To fill a "hole" immediately would require relocation of some existing bucket.

Smallest bucket size	Lower Bound	Simulation results (100 runs)	Upper bound
12 (20000 records)	0.7970	0.82800 ± 0.00071	0.85696
60 (150000 records)	0.7776	0.81020 ± 0.00056	0.83706

Table 1: Total storage utilization for large files (95% confidence interval)

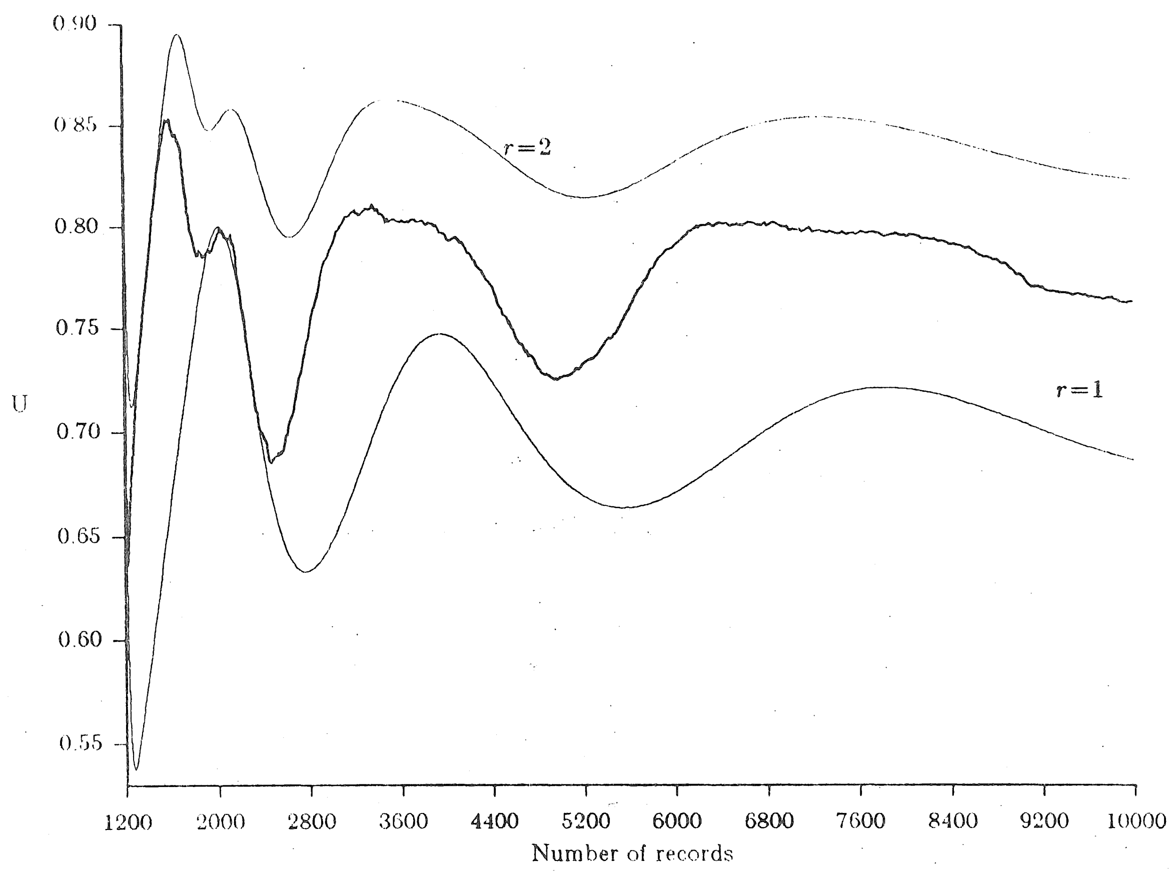

Figure 2: Average total storage utilization for smallest bucket size 60

The total storage utilization is over 80%, which is more than 10 percentage points higher than the storage utilization of a standard B$^+$-tree. The results are close to the upper bound, which indicates that there is little to be gained by bucket relocation.

5 Conclusions

We have presented a general algorithm for the dynamic storage allocation problem. This algorithm can be adapted for many different problems, and can be used in an environment that provides storage allocation for blocks of a unique size.

We have applied this algorithm to a particular case: B$^+$-trees with partial expansions. Simulation results confirm the fact that the storage utilization obtained with this algorithm is almost optimal. As a further conclusion, B$^+$-trees with partial expansions using this allocation algorithm for secondary storage is the better implementation of B$^+$-trees, to the best of our knowledge. The main reason being that it provides a storage utilization of around 81% without increasing the insertion time [2]. On the other hand, B^*-trees provide the same storage utilization, but double the insertion time [1].

Acknowledgements

I wish to thank Jorge Olivos for starting my interest in this problem, Per-Åke Larson for suggesting the application, Jose Pino for encouraging me to write this paper, and Tony Gahlinger for his helpful comments.

References

[1] Baeza-Yates, R., The Expected Behaviour of B^+-trees, Department of Computer Science, University of Chile, Santiago, Chile, 1985. Also available as Technical Report CS-86-68, Dept. of Computer Science, University of Waterloo, 1986.

[2] Baeza-Yates, R. and Larson, P-Å. Analysis of B^+-trees with Partial Expansions, Research Report CS-87-04, Dept. of Computer Science, University of Waterloo, Waterloo, Canada.

[3] Coffman, E. "An Introduction to Combinatorial Models of Dynamic Storage Allocation", *SIAM Review* 25, 3 (1983), 311-325.

[4] Coffman, E. and Leighton, F. "A Provably Efficient Algorithm for Dynamic Storage Allocation", Proceedings of the 18^th annual ACM Symposium on Theory of Computing, Berkeley, California (1986), 77-90.

[5] Comer, D., The Ubiquitous B-Tree, *Computing Surveys* **11**, 2 (1979), 121-137.

[6] Krogdahl, S. "A Dynamic Storage Allocation Problem", *Information Processing Letters* **2** (1973) 96-99.

[7] Larson, P-Å. Linear Hashing with Partial Expansions, Proc. 6th Int. Conf. on Very Large Data Bases, (Montreal, Canada, 1980), 224-232.

[8] Lomet, D., Partial Expansions for File Organizations with an Index, *ACM Trans. on Database Systems* **12** (1987), 65-84.

[9] Robson, J. "An Estimate of the Store Size Necessary for Dynamic Storage Allocation", *Journal of the ACM* **18** (1971) 416-423.

[10] Robson, J. "Bounds for Some Functions Concerning Dynamic Storage Allocation", *Journal of the ACM* **21** (1974) 491-499.