A New Hashing Package for UNIX[1]

Margo Seltzer – University of California, Berkeley
Ozan Yigit – York University

ABSTRACT

UNIX support of disk oriented hashing was originally provided by *dbm* [ATT79] and subsequently improved upon in *ndbm* [BSD86]. In AT&T System V, in-memory hashed storage and access support was added in the *hsearch* library routines [ATT85]. The result is a system with two incompatible hashing schemes, each with its own set of shortcomings.

This paper presents the design and performance characteristics of a new hashing package providing a superset of the functionality provided by *dbm* and *hsearch*. The new package uses linear hashing to provide efficient support of both memory based and disk based hash tables with performance superior to both *dbm* and *hsearch* under most conditions.

Introduction

Current UNIX systems offer two forms of hashed data access. *Dbm* and its derivatives provide keyed access to disk resident data while *hsearch* provides access for memory resident data. These two access methods are incompatible in that memory resident hash tables may not be stored on disk and disk resident tables cannot be read into memory and accessed using the in-memory routines.

Dbm has several shortcomings. Since data is assumed to be disk resident, each access requires a system call, and almost certainly, a disk operation. For extremely large databases, where caching is unlikely to be effective, this is acceptable, however, when the database is small (i.e. the password file), performance improvements can be obtained through caching pages of the database in memory. In addition, *dbm* cannot store data items whose total key and data size exceed the page size of the hash table. Similarly, if two or more keys produce the same hash value and their total size exceeds the page size, the table cannot store all the colliding keys.

The in-memory *hsearch* routines have different shortcomings. First, the notion of a single hash table is embedded in the interface, preventing an application from accessing multiple tables concurrently. Secondly, the routine to create a hash table requires a parameter which declares the size of the hash table. If this size is set too low, performance degradation or the inability to add items to the table may result. In addition, *hsearch* requires that the application allocate memory for the key and data items. Lastly, the *hsearch* routines provide no interface to store hash tables on disk.

The goal of our work was to design and implement a new package that provides a superset of the functionality of both *dbm* and *hsearch*. The package had to overcome the interface shortcomings cited above and its implementation had to provide performance equal or superior to that of the existing implementations. In order to provide a compact disk representation, graceful table growth, and expected constant time performance, we selected Litwin's linear hashing algorithm [LAR88, LIT80]. We then enhanced the algorithm to handle page overflows and large key handling with a single mechanism, named buddy-in-waiting.

Existing UNIX Hashing Techniques

Over the last decade, several dynamic hashing schemes have been developed for the UNIX timesharing system, starting with the inclusion of *dbm*, a minimal database library written by Ken Thompson [THOM90], in the Seventh Edition UNIX system. Since then, an extended version of the same library, *ndbm*, and a public-domain clone of the latter, *sdbm*, have been developed. Another interface-compatible library *gdbm*, was recently made available as part of the Free Software Foundation's (FSF) software distribution.

All of these implementations are based on the idea of revealing just enough bits of a hash value to locate a page in a single access. While *dbm/ndbm* and *sdbm* map the hash value directly to a disk address, *gdbm* uses the hash value to index into a *directory* [ENB88] containing disk addresses.

The *hsearch* routines in System V are designed to provide memory-resident hash tables. Since data access does not require disk access, simple hashing schemes which may require multiple probes into the table are used. A more interesting version of *hsearch* is a public domain library, *dynahash*, that implements Larson's in-memory adaptation [LAR88] of linear hashing [LIT80].

[1] UNIX is a registered trademark of AT&T.

dbm and *ndbm*

The *dbm* and *ndbm* library implementations are based on the same algorithm by Ken Thompson [THOM90, TOR88, WAL84], but differ in their programmatic interfaces. The latter is a modified version of the former which adds support for multiple databases to be open concurrently. The discussion of the algorithm that follows is applicable to both *dbm* and *ndbm*.

The basic structure of *dbm* calls for fixed-sized disk blocks (buckets) and an *access* function that maps a key to a bucket. The interface routines use the *access* function to obtain the appropriate bucket in a single disk access.

Within the *access* function, a bit-randomizing hash function[2] is used to convert a key into a 32-bit hash value. Out of these 32 bits, only as many bits as necessary are used to determine the particular bucket on which a key resides. An in-memory bitmap is used to determine how many bits are required. Each bit indicates whether its associated bucket has been split yet (a 0 indicating that the bucket has not yet split). The use of the hash function and the bitmap is best described by stepping through database creation with multiple invocations of a *store* operation.

Initially, the hash table contains a single bucket (bucket 0), the bit map contains a single bit (bit 0 corresponding to bucket 0), and 0 bits of a hash value are examined to determine where a key is placed (in bucket 0). When bucket 0 is full, its bit in the bitmap (bit 0) is set, and its contents are split between buckets 0 and 1, by considering the 0^{th} bit (the lowest bit not previously examined) of the hash value for each key within the bucket. Given a well-designed hash function, approximately half of the keys will have hash values with the 0^{th} bit set. All such keys and associated data are moved to bucket 1, and the rest remain in bucket 0.

After this split, the file now contains two buckets, and the bitmap contains three bits: the 0^{th} bit is set to indicate a bucket 0 split when no bits of the hash value are considered, and two more unset bits for buckets 0 and 1. The placement of an incoming key now requires examination of the 0^{th} bit of the hash value, and the key is placed either in bucket 0 or bucket 1. If either bucket 0 or bucket 1 fills up, it is split as before, its bit is set in the bitmap, and a new set of unset bits are added to the bitmap.

Each time we consider a new bit (bit n), we add 2^{n+1} bits to the bitmap and obtain 2^{n+1} more addressable buckets in the file. As a result, the bitmap contains the previous $2^{n+1}-1$ bits ($1+2+4+...+2^n$) which trace the entire *split history* of the addressable buckets.

Given a key and the bitmap created by this algorithm, we first examine bit 0 of the bitmap (the bit to consult when 0 bits of the hash value are being examined). If it is set (indicating that the bucket split), we begin considering the bits of the 32-bit hash value. As bit n is revealed, a mask equal to $2^{n+1}-1$ will yield the current bucket address. Adding $2^{n+1}-1$ to the bucket address identifies which bit in the bitmap must be checked. We continue revealing bits of the hash value until all set bits in the bitmap are exhausted. The following algorithm, a simplification of the algorithm due to Ken Thompson [THOM90, TOR88], uses the hash value and the bitmap to calculate the bucket address as discussed above.

```
hash = calchash(key);
mask = 0;
while (isbitset((hash & mask) + mask))
        mask = (mask << 1) + 1;
bucket = hash & mask;
```

sdbm

The *sdbm* library is a public-domain clone of the *ndbm* library, developed by Ozan Yigit to provide *ndbm*'s functionality under some versions of UNIX that exclude it for licensing reasons [YIG89]. The programmer interface, and the basic structure of *sdbm* is identical to *ndbm* but internal details of the *access* function, such as the calculation of the bucket address, and the use of different hash functions make the two incompatible at the database level.

The *sdbm* library is based on a simplified implementation of Larson's 1978 *dynamic hashing* algorithm including the *refinements and variations* of section 5 [LAR78]. Larson's original algorithm calls for a forest of binary hash trees that are accessed by two hash functions. The first hash function selects a particular tree within the forest. The second hash function, which is required to be a boolean pseudo-random number generator that is seeded by the key, is used to traverse the tree until internal (split) nodes are exhausted and an external (non-split) node is reached. The bucket addresses are stored directly in the external nodes.

Larson's refinements are based on the observation that the nodes can be represented by a single bit that is set for internal nodes and not set for external nodes, resulting in a radix search trie. Figure 1 illustrates this. Nodes A and B are internal (split) nodes, thus having no bucket addresses associated with them. Instead, the external nodes (C, D, and E) each need to refer to a bucket address. These bucket addresses can be stored in the trie itself where the subtries would

[2] This bit-randomizing property is important to obtain radically different hash values for nearly identical keys, which in turn avoids clustering of such keys in a single bucket.

live if they existed [KNU68]. For example, if nodes F and G were the children of node C, the bucket address L00 could reside in the bits that will eventually be used to store nodes F and G and all their children.

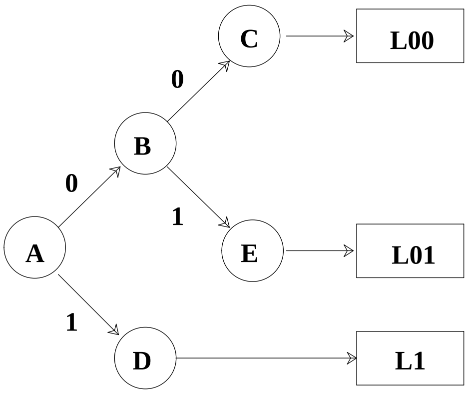

Figure 1: Radix search trie with internal nodes A and B, external nodes C, D, and E, and bucket addresses stored in the unused portion of the trie.

Further simplifications of the above [YIG89] are possible. Using a single radix trie to avoid the first hash function, replacing the pseudo-random number generator with a well designed, bit-randomizing hash function, and using the portion of the hash value exposed during the trie traversal as a direct bucket address results in an *access* function that works very similar to Thompson's algorithm above. The following algorithm uses the hash value to traverse a linearized radix trie[3] starting at the 0^{th} bit.

```
tbit = 0;       /* radix trie index */
hbit = 0;       /* hash bit index   */
mask = 0;
hash = calchash(key);

for (mask = 0;
     isbitset(tbit);
     mask = (mask << 1) + 1)
        if (hash & (1 << hbit++)))
            /* right son  */
            tbit = 2 * tbit + 2;
        else
            /* left son   */
            tbit = 2 * tbit + 1;

bucket = hash & mask;
```

[3] A linearized radix trie is merely an array representation of the radix search trie described above. The children of the node with index i can be found at the nodes indexed 2*i+1 and 2*i+2.

gdbm

The gdbm (GNU data base manager) library is a UNIX database manager written by Philip A. Nelson, and made available as a part of the FSF software distribution. The gdbm library provides the same functionality of the *dbm*/*ndbm* libraries [NEL90] but attempts to avoid some of their shortcomings. The gdbm library allows for arbitrary-length data, and its database is a singular, non-sparse[4] file. The gdbm library also includes *dbm* and *ndbm* compatible interfaces.

The gdbm library is based on *extensible hashing*, a dynamic hashing algorithm by Fagin et al [FAG79]. This algorithm differs from the previously discussed algorithms in that it uses a *directory* that is a collapsed representation [ENB88] of the radix search trie used by *sdbm*.

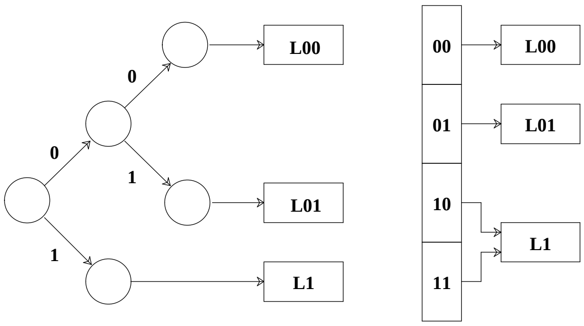

Figure 2: A radix search trie and a directory representing the trie.

In this algorithm, a directory consists of a search trie of depth n, containing 2^n bucket addresses (i.e. each element of the trie is a bucket address). To access the hash table, a 32-bit hash value is calculated and n bits of the value are used to index into the directory to obtain a bucket address. It is important to note that multiple entries of this directory may contain the same bucket address as a result of directory doubling during bucket splitting. Figure 2 illustrates the relationship between a typical (skewed) search trie and its directory representation. The formation of the directory shown in the figure is as follows.

[4] It does not contain holes.

Initially, there is one slot in the directory addressing a single bucket. The depth of the trie is 0 and 0 bits of each hash value are examined to determine in which bucket to place a key; all keys go in bucket 0. When this bucket is full, its contents are divided between L0 and L1 as was done in the previously discussed algorithms. After this split, the address of the second bucket must be stored in the directory. To accommodate the new address, the directory is split[5], by doubling it, thus increasing the depth of the directory by one.

After this split, a single bit of the hash value needs to be examined to decide whether the key belongs to L0 or L1. Once one of these buckets fills (L0 for example), it is split as before, and the directory is split again to make room for the address of the third bucket. This splitting causes the addresses of the non-splitting bucket (L1) to be duplicated. The directory now has four entries, a depth of 2, and indexes the buckets L00, L01 and L1, as shown in the Figure 2.

The crucial part of the algorithm is the observation that L1 is addressed twice in the directory. If this bucket were to split now, the directory already contains room to hold the address of the new bucket. In general, the relationship between the directory and the number of bucket addresses contained therein is used to decide when to split the directory. Each bucket has a depth, (n_b), associated with it and appears in the directory exactly 2^{n-n_b} times. When a bucket splits, its depth increases by one. The directory must split any time a bucket's depth exceeds the depth of the directory. The following code fragment helps to illustrate the extendible hashing algorithm [FAG79] for accessing individual buckets and maintaining the directory.

```
hash = calchash(key);
mask = maskvec[depth];

bucket = directory[hash & mask];

/* Key Insertion */
if (store(bucket, key, data) == FAIL) {
        newbl = getpage();
        bucket->depth++;
        newbl->depth = bucket->depth;
        if (bucket->depth > depth) {
                /* double directory */
                depth++;
                directory = double(directory);
        }
        splitbucket(bucket, newbl)
        ...
}
```

[5] This decision to split the directory is based on a comparison of the depth of the page being split and the depth of the trie. In Figure 2, the depths of both L00 and L01 are 2, whereas the depth of L1 is 1. Therefore, if L1 were to split, the directory would not need to split. In reality, a bucket is allocated for the directory at the time of file creation so although the directory splits logically, physical splits do not occur until the file becomes quite large.

hsearch

Since *hsearch* does not have to translate hash values into disk addresses, it can use much simpler algorithms than those defined above. System V's *hsearch* constructs a fixed-size hash table (specified by the user at table creation). By default, a multiplicative hash function based on that described in Knuth, Volume 3, section 6.4 [KNU68] is used to obtain a primary bucket address. If this bucket is full, a secondary multiplicative hash value is computed to define the probe interval. The probe interval is added to the original bucket address (modulo the table size) to obtain a new bucket address. This process repeats until an empty bucket is found. If no bucket is found, an insertion fails with a ‘‘table full’’ condition.

The basic algorithm may be modified by a number of compile time options available to those users with AT&T source code. First, the package provides two options for hash functions. Users may specify their own hash function by compiling with ‘‘USCR’’ defined and declaring and defining the variable *hcompar*, a function taking two string arguments and returning an integer. Users may also request that hash values be computed simply by taking the modulo of key (using division rather than multiplication for hash value calculation). If this technique is used, collisions are resolved by scanning sequentially from the selected bucket (linear probing). This option is available by defining the variable ‘‘DIV’’ at compile time.

A second option, based on an algorithm discovered by Richard P. Brent, rearranges the table at the time of insertion in order to speed up retrievals. The basic idea is to shorten long probe sequences by lengthening short probe sequences. Once the probe chain has exceeded some threshold (Brent suggests 2), we attempt to shuffle any colliding keys (keys which appeared in the probe sequence of the new key). The details of this key shuffling can be found in [KNU68] and [BRE73]. This algorithm may be obtained by defining the variable ‘‘BRENT’’ at compile time.

A third set of options, obtained by defining ‘‘CHAINED’’, use linked lists to resolve collisions. Either of the primary hash function described above may be used, but all collisions are resolved by building a linked list of entries from the primary bucket. By default, new entries will be added to a bucket at the beginning of the bucket chain. However, compile options ‘‘SORTUP’’ or ‘‘SORTDOWN’’ may be specified to order the hash chains within each bucket.

dynahash

The *dynahash* library, written by Esmond Pitt, implements Larson's linear hashing algorithm [LAR88] with an *hsearch* compatible interface. Intuitively, a hash table begins as a single bucket and grows in generations, where a generation corresponds to a doubling in the size of the hash table. The 0^{th} generation occurs as the table grows from one bucket to two. In the next generation the table grows from two to four. During each generation, every bucket that existed at the beginning of the generation is split.

The table starts as a single bucket (numbered 0), the current split bucket is set to bucket 0, and the maximum split point is set to twice the current split point (0). When it is time for a bucket to split, the keys in the current split bucket are divided between the current split bucket and a new bucket whose bucket number is equal to 1 + current split bucket + maximum split point. We can determine which keys move to the new bucket by examining the n^{th} bit of a key's hash value where n is the generation number. After the bucket at the maximum split point has been split, the generation number is incremented, the current split point is set back to zero, and the maximum split point is set to the number of the last bucket in the file (which is equal to twice the old maximum split point plus 1).

To facilitate locating keys, we maintain two masks. The low mask is equal to the maximum split bucket and the high mask is equal to the next maximum split bucket. To locate a specific key, we compute a 32-bit hash value using a bit-randomizing algorithm such as the one described in [LAR88]. This hash value is then masked with the high mask. If the resulting number is greater than the maximum bucket in the table (current split bucket + maximum split point), the hash value is masked with the low mask. In either case, the result of the mask is the bucket number for the given key. The algorithm below illustrates this process.

```
h = calchash(key);
bucket = h & high_mask;
if ( bucket > max_bucket )
        bucket = h & low_mask;
return(bucket);
```

In order to decide when to split a bucket, *dynahash* uses *controlled splitting*. A hash table has a fill factor which is expressed in terms of the average number of keys in each bucket. Each time the table's total number of keys divided by its number of buckets exceeds this fill factor, a bucket is split.

Since the *hsearch* create interface (*hcreate*) calls for an estimate of the final size of the hash table (*nelem*), *dynahash* uses this information to initialize the table. The initial number of buckets is set to *nelem* rounded to the next higher power of two. The current split point is set to 0 and the maximum bucket and maximum split point are set to this rounded value.

The New Implementation

Our implementation is also based on Larson's linear hashing [LAR88] algorithm as well as the *dynahash* implementation. The *dbm* family of algorithms decide dynamically which bucket to split and when to split it (when it overflows) while *dynahash* splits in a predefined order (linearly) and at a predefined time (when the table fill factor is exceeded). We use a hybrid of these techniques. Splits occur in the predefined order of linear hashing, but the time at which pages are split is determined both by page overflows (*uncontrolled splitting*) and by exceeding the fill factor (*controlled splitting*)

A hash table is parameterized by both its bucket size (*bsize*) and fill factor (*ffactor*). Whereas *dynahash's* buckets can be represented as a linked list of elements in memory, our package needs to support disk access, and must represent buckets in terms of pages. The *bsize* is the size (in bytes) of these pages. As in linear hashing, the number of buckets in the table is equal to the number of keys in the table divided by *ffactor*.[6] The controlled splitting occurs each time the number of keys in the table exceeds the fill factor multiplied by the number of buckets.

Inserting keys and splitting buckets is performed precisely as described previously for *dynahash*. However, since buckets are now comprised of pages, we must be prepared to handle cases where the size of the keys and data in a bucket exceed the bucket size.

Overflow Pages

There are two cases where a key may not fit in its designated bucket. In the first case, the total size of the key and data may exceed the bucket size. In the second, addition of a new key could cause an overflow, but the bucket in question is not yet scheduled to be split. In existing implementations, the second case never arises (since buckets are split when they overflow) and the first case is not handled at all. Although large key/data pair handling is difficult and expensive, it is essential. In a linear hashed implementation, overflow pages are required for buckets which overflow before they are split, so we can use the same mechanism for large key/data pairs that we use for overflow pages. Logically, we chain overflow

[6] This is not strictly true. The file does not contract when keys are deleted, so the number of buckets is actually equal to the maximum number of keys ever present in the table divided by the fill factor.

pages to the buckets (also called primary pages). In a memory based representation, overflow pages do not pose any special problems because we can chain overflow pages to primary pages using memory pointers. However, mapping these overflow pages into a disk file is more of a challenge, since we need to be able to address both bucket pages, whose numbers are growing linearly, and some indeterminate number of overflow pages without reorganizing the file.

One simple solution would be to allocate a separate file for overflow pages. The disadvantage with such a technique is that it requires an extra file descriptor, an extra system call on open and close, and logically associating two independent files. For these reasons, we wanted to map both primary pages and overflow pages into the same file space.

The buddy-in-waiting algorithm provides a mechanism to support multiple pages per logical bucket while retaining the simple split sequence of linear hashing. Overflow pages are preallocated between generations of primary pages. These overflow pages are used by any bucket containing more keys than fit on the primary page and are reclaimed, if possible, when the bucket later splits. Figure 3 depicts the layout of primary pages and overflow pages within the same file. Overflow page use information is recorded in bitmaps which are themselves stored on overflow pages. The addresses of the bitmap pages and the number of pages allocated at each split point are stored in the file header. Using this information, both overflow addresses and bucket addresses can be mapped to disk addresses by the following calculation:

```
int     bucket;     /* bucket address */
u_short oaddr;      /* OVERFLOW address */
int     nhdr_pages; /* npageseiı fi   header */
int     spares[32]; /* npages at each split */
int     log2();     /* ceil(log base 2) */

#DEFINE BUCKET_TO_PAGE(bucket) \
    bucket + nhdr_pages + \
    (bucket?spares[logs2(bucket + 1)-1]:0)

#DEFINE OADDR_TO_PAGE(oaddr) \
    BUCKET_TO_PAGE((1 << (oaddr>>11)) - 1) + \
    oaddr & 0x7ff;
```

An overflow page is addressed by its split point, identifying the generations between which the overflow page is allocated, and its page number, identifying the particular page within the split point. In this implementation, offsets within pages are 16 bits long (limiting the maximum page size to 32K), so we select an overflow page addressing algorithm that can be expressed in 16 bits and which allows quick retrieval. The top five bits indicate the split point and the lower eleven indicate the page number within the split point. Since five bits are reserved for the split point, files may split 32 times yielding a maximum file size of 2^{32} buckets and $32*2^{11}$ overflow pages. The maximum page size is 2^{15}, yielding a maximum file size greater than 131,000 GB (on file systems supporting files larger than 4GB).

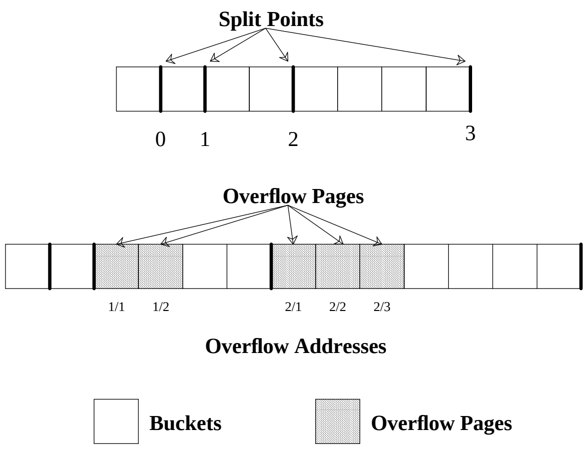

Figure 3: Split points occur between generations and are numbered from 0. In this figure there are two overflow pages allocated at split point 1 and three allocated at split point 2.

Buffer Management

The hash table is stored in memory as a logical array of bucket pointers. Physically, the array is arranged in segments of 256 pointers. Initially, there is space to allocate 256 segments. Reallocation occurs when the number of buckets exceeds 32K (256 * 256). Primary pages may be accessed directly through the array by bucket number and overflow pages are referenced logically by their overflow page address. For small hash tables, it is desirable to keep all pages in main memory while on larger tables, this is probably impossible. To satisfy both of these requirements, the package includes buffer management with LRU (least recently used) replacement.

By default, the package allocates up to 64K bytes of buffered pages. All pages in the buffer pool are linked in LRU order to facilitate fast replacement. Whereas efficient access to primary pages is provided by the bucket array, efficient access to overflow pages is provided by linking overflow page buffers to their predecessor page (either the primary page or another overflow page). This means that an overflow page cannot be present in the buffer pool if its primary page is not present. This does not impact performance or functionality, because an overflow page will be accessed only after its predecessor page has been accessed. Figure 4 depicts the data structures used to manage the buffer pool.

The in-memory bucket array contains pointers to buffer header structures which represent primary pages. Buffer headers contain modified bits, the page address of the buffer, a pointer to the actual buffer, and a pointer to the buffer header for an overflow page if it exists, in addition to the LRU links. If the buffer corresponding to a particular bucket is not in memory, its pointer is NULL. In effect, pages are linked in three ways. Using the buffer headers, they are linked physically through the LRU links and the overflow links. Using the pages themselves, they are linked logically through the overflow addresses on the page. Since overflow pages are accessed only after their predecessor pages, they are removed from the buffer pool when their primary is removed.

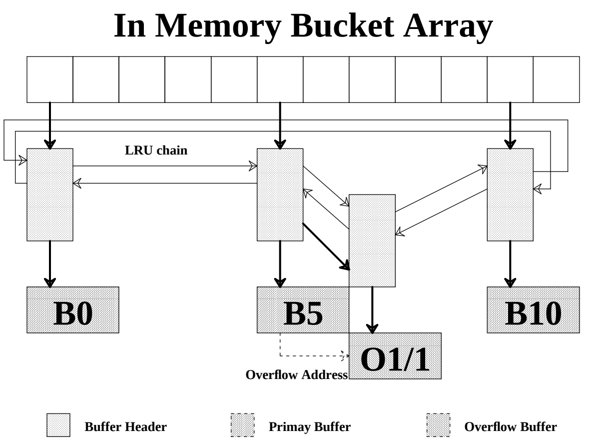

Figure 4: Three primary pages (B0, B5, B10) are accessed directly from the bucket array. The one overflow page (O1/1) is linked physically from its primary page's buffer header as well as logically from its predecessor page buffer (B5).

Table Parameterization

When a hash table is created, the bucket size, fill factor, initial number of elements, number of bytes of main memory used for caching, and a user-defined hash function may be specified. The bucket size (and page size for overflow pages) defaults to 256 bytes. For tables with large data items, it may be preferable to increase the page size, and, conversely, applications storing small items exclusively in memory may benefit from a smaller bucket size. A bucket size smaller than 64 bytes is not recommended.

The fill factor indicates a desired density within the hash table. It is an approximation of the number of keys allowed to accumulate in any one bucket, determining when the hash table grows. Its default is eight. If the user knows the average size of the key/data pairs being stored in the table, near optimal bucket sizes and fill factors may be selected by applying the equation:

```
(1)     ((average_pair_length + 4) *
         ffactor) >= bsize
```

For highly time critical applications, experimenting with different bucket sizes and fill factors is encouraged.

Figures 5a,b, and c illustrate the effects of varying page sizes and fill factors for the same data set. The data set consisted of 24474 keys taken from an online dictionary. The data value for each key was an ASCII string for an integer from 1 to 24474 inclusive. The test run consisted of creating a new hash table (where the ultimate size of the table was known in advance), entering each key/data pair into the table and then retrieving each key/data pair from the table. Each of the graphs shows the timings resulting from varying the pagesize from 128 bytes to 1M and the fill factor from 1 to 128. For each run, the buffer size was set at 1M. The tests were all run on an HP 9000/370 (33.3 Mhz MC68030), with 16M of memory, 64K physically addressed cache, and an HP7959S disk drive, running 4.3BSD-Reno single-user.

Both system time (Figure 5a) and elapsed time (Figure 5b) show that for all bucket sizes, the greatest performance gains are made by increasing the fill factor until equation 1 is satisfied. The user time shown in Figure 5c gives a more detailed picture of how performance varies. The smaller bucket sizes require fewer keys per page to satisfy equation 1 and therefore incur fewer collisions. However, when the buffer pool size is fixed, smaller pages imply more pages. An increased number of pages means more *malloc(3)* calls and more overhead in the hash package's buffer manager to manage the additional pages.

The tradeoff works out most favorably when the page size is 256 and the fill factor is 8. Similar conclusions were obtained if the test was run without knowing the final table size in advance. If the file was closed and written to disk, the conclusions were still the same. However, rereading the file from disk was slightly faster if a larger bucket size and fill factor were used (1K bucket size and 32 fill factor). This follows intuitively from the improved efficiency of performing 1K reads from the disk rather than 256 byte reads. In general, performance for disk based tables is best when the page size is approximately 1K.

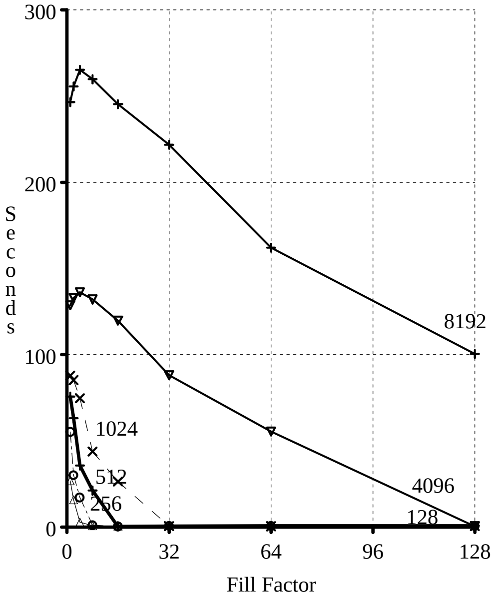

Figure 5a: System Time for dictionary data set with 1M of buffer space and varying bucket sizes and fill factors. Each line is labeled with its bucket size.

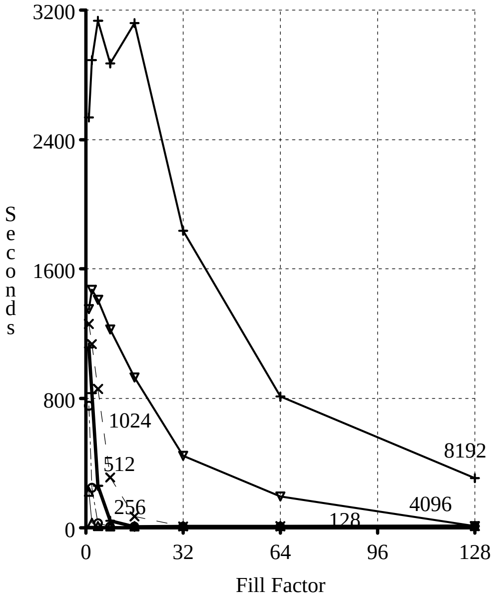

Figure 5b: Elapsed Time for dictionary data set with 1M of buffer space and varying bucket sizes and fill factors. Each line is labeled with its bucket size.

If an approximation of the number of elements ultimately to be stored in the hash table is known at the time of creation, the hash package takes this number as a parameter and uses it to hash entries into the full sized table rather than growing the table from a single bucket. If this number is not known, the hash table starts with a single bucket and gracefully expands as elements are added, although a slight performance degradation may be noticed. Figure 6 illustrates the difference in performance between storing keys in a file when the ultimate size is known (the left bars in each set), compared to building the file when the ultimate size is unknown (the right bars in each set). Once the fill factor is sufficiently high for the page size (8), growing the table dynamically does little to degrade performance.

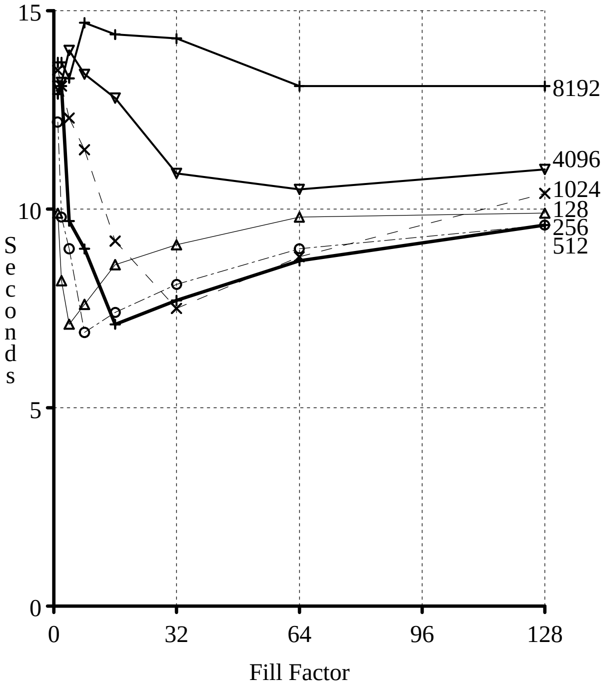

Figure 5c: User Time for dictionary data set with 1M of buffer space and varying bucket sizes and fill factors. Each line is labeled with its bucket size.

Since no known hash function performs equally well on all possible data, the user may find that the built-in hash function does poorly on a particular data set. In this case, a hash function, taking two arguments (a pointer to a byte string and a length) and returning an unsigned long to be used as the hash value, may be specified at hash table creation time. When an existing hash table is opened and a hash function is specified, the hash package will try to determine that the hash function supplied is the one with which the table was created. There are a variety of hash functions provided with the package. The default function for the package is the one which offered the best performance in terms of cycles executed per call (it did not produce the fewest collisions although it was within a small percentage of the function that produced the fewest collisions). Again, in time critical applications, users are encouraged to experiment with a variety of hash functions to achieve optimal performance.

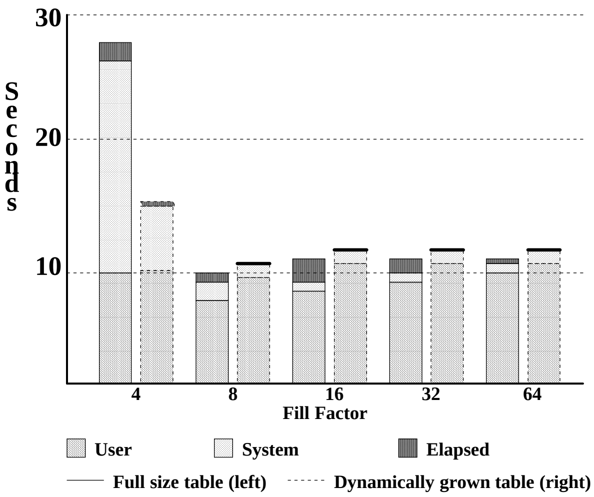

Figure 6: The total regions indicate the difference between the elapsed time and the sum of the system and user time. The left bar of each set depicts the timing of the test run when the number of entries is known in advance. The right bars depict the timing when the file is grown from a single bucket.

Since this hashing package provides buffer management, the amount of space allocated for the buffer pool may be specified by the user. Using the same data set and test procedure as used to derive the graphs in Figures 5a-c, Figure 7 shows the impact of varying the size of the buffer pool. The bucket size was set to 256 bytes and the fill factor was set to 16. The buffer pool size was varied from 0 (the minimum number of pages required to be buffered) to 1M. With 1M of buffer space, the package performed no I/O for this data set. As Figure 7 illustrates, increasing the buffer pool size can have a dramatic affect on resulting performance.[7]

[7] Some allocators are extremely inefficient at allocating memory. If you find that applications are running out of memory before you think they should, try varying the pagesize to get better utilization from the memory allocator.

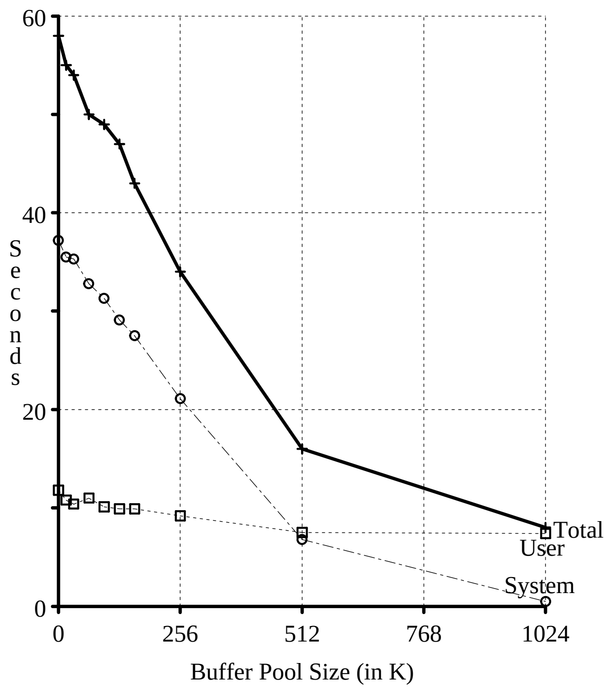

Figure 7: User time is virtually insensitive to the amount of buffer pool available, however, both system time and elapsed time are inversely proportional to the size of the buffer pool. Even for large data sets where one expects few collisions, specifying a large buffer pool dramatically improves performance.

Enhanced Functionality

This hashing package provides a set of compatibility routines to implement the *ndbm* interface. However, when the native interface is used, the following additional functionality is provided:

- Inserts never fail because too many keys hash to the same value.
- Inserts never fail because key and/or associated data is too large
- Hash functions may be user-specified.
- Multiple pages may be cached in main memory.

It also provides a set of compatibility routines to implement the *hsearch* interface. Again, the native interface offers enhanced functionality:

- Files may grow beyond *nelem* elements.
- Multiple hash tables may be accessed concurrently.
- Hash tables may be stored and accessed on disk.
- Hash functions may be user-specified at runtime.

Relative Performance of the New Implementation

The performance testing of the new package is divided into two test suites. The first suite of tests requires that the tables be read from and written to disk. In these tests, the basis for comparison is the 4.3BSD-Reno version of *ndbm*. Based on the designs of *sdbm* and *gdbm*, they are expected to perform similarly to *ndbm*, and we do not show their performance numbers. The second suite contains the memory resident test which does not require that the files ever be written to disk, only that hash tables may be manipulated in main memory. In this test, we compare the performance to that of the *hsearch* routines.

For both suites, two different databases were used. The first is the dictionary database described previously. The second was constructed from a password file with approximately 300 accounts. Two records were constructed for each account. The first used the logname as the key and the remainder of the password entry for the data. The second was keyed by uid and contained the entire password entry as its data field. The tests were all run on the HP 9000 with the same configuration previously described. Each test was run five times and the timing results of the runs were averaged. The variance across the 5 runs was approximately 1% of the average yielding 95% confidence intervals of approximately 2%.

Disk Based Tests

In these tests, we use a bucket size of 1024 and a fill factor of 32.

create test

The keys are entered into the hash table, and the file is flushed to disk.

read test

A lookup is performed for each key in the hash table.

verify test

A lookup is performed for each key in the hash table, and the data returned is compared against that originally stored in the hash table.

sequential retrieve

All keys are retrieved in sequential order from the hash table. The *ndbm* interface allows sequential retrieval of the keys from the database, but does not return the data associated with each key. Therefore, we compare the performance of the new package to two different runs of *ndbm*. In the first case, *ndbm* returns only the keys while in the second, *ndbm* returns both the keys and the data (requiring a second call to the library). There is a single run for the new library since it returns both the key and the data.

In-Memory Test

This test uses a bucket size of 256 and a fill factor of 8.

create/read test

In this test, a hash table is created by inserting all the key/data pairs. Then a keyed retrieval is performed for each pair, and the hash table is destroyed.

Performance Results

Figures 8a and 8b show the user time, system time, and elapsed time for each test for both the new implementation and the old implementation (*hsearch* or *ndbm*, whichever is appropriate) as well as the improvement. The improvement is expressed as a percentage of the old running time:

```
% = 100 * (old_time - new_time) / old_time
```

In nearly all cases, the new routines perform better than the old routines (both *hsearch* and *ndbm*). Although the **create** tests exhibit superior user time performance, the test time is dominated by the cost of writing the actual file to disk. For the large database (the dictionary), this completely overwhelmed the system time. However, for the small data base, we see that differences in both user and system time contribute to the superior performance of the new package.

The **read**, **verify**, and **sequential** results are deceptive for the small database since the entire test ran in under a second. However, on the larger database the **read** and **verify** tests benefit from the caching of buckets in the new package to improve performance by over 80%. Since the first **sequential** test does not require *ndbm* to return the data values, the user time is lower than for the new package. However when we require both packages to return data, the new package excels in all three timings.

The small database runs so quickly in the memory-resident case that the results are uninteresting. However, for the larger database the new package pays a small penalty in system time because it limits its main memory utilization and swaps pages out to temporary storage in the file system while the *hsearch* package requires that the application allocate enough space for all key/data pair. However, even with the system time penalty, the resulting elapsed time improves by over 50%.

	hash	*ndbm*	*%change*
CREATE			
user	6.4	12.2	48
sys	32.5	34.7	6
elapsed	**90.4**	**99.6**	**9**
READ			
user	3.4	6.1	44
sys	1.2	15.3	92
elapsed	**4.0**	**21.2**	**81**
VERIFY			
user	3.5	6.3	44
sys	1.2	15.3	92
elapsed	**4.0**	**21.2**	**81**
SEQUENTIAL			
user	2.7	1.9	-42
sys	0.7	3.9	82
elapsed	**3.0**	**5.0**	**40**
SEQUENTIAL (with data retrieval)			
user	2.7	8.2	67
sys	0.7	4.3	84
elapsed	**3.0**	**12.0**	**75**

	hash	*hsearch*	*%change*
CREATE/READ			
user	6.6	17.2	62
sys	1.1	0.3	-266
elapsed	**7.8**	**17.0**	**54**

Figure 8a: Timing results for the dictionary database.

Conclusion

This paper has presented the design, implementation and performance of a new hashing package for UNIX. The new package provides a superset of the functionality of existing hashing packages and incorporates additional features such as large key handling, user defined hash functions, multiple hash tables, variable sized pages, and linear hashing. In nearly all cases, the new package provides improved performance on the order of 50-80% for the workloads shown. Applications such as the loader, compiler, and mail, which currently implement their own hashing routines, should be modified to use the generic routines.

This hashing package is one access method which is part of a generic database access package being developed at the University of California, Berkeley. It will include a btree access method as well as fixed and variable length record access methods in addition to the hashed support presented here. All of the access methods are based on a key/data pair interface and appear identical to the application layer, allowing application implementations to be largely independent of the database type. The package is expected to be an integral part of the 4.4BSD system, with various standard applications such as more(1), sort(1) and vi(1) based on it. While the current design does not support multi-user access or transactions, they could be incorporated relatively easily.

	hash	*ndbm*	*%change*
CREATE			
user	0.2	0.4	50
sys	0.1	1.0	90
elapsed	**0**	**3.2**	**100**
READ			
user	0.1	0.1	0
sys	0.1	0.4	75
elapsed	**0.0**	**0.0**	**0**
VERIFY			
user	0.1	0.2	50
sys	0.1	0.3	67
elapsed	**0.0**	**0.0**	**0**
SEQUENTIAL			
user	0.1	0.0	-100
sys	0.1	0.1	0
elapsed	**0.0**	**0.0**	**0**
SEQUENTIAL (with data retrieval)			
user	0.1	0.1	0
sys	0.1	0.1	0
elapsed	**0.0**	**0.0**	**0**

	hash	*hsearch*	*%change*
CREATE/READ			
user	0.3	0.4	25
sys	0.0	0.0	0
elapsed	**0.0**	**0.0**	**0**

Figure 8b: Timing results for the password database.

References

[ATT79] AT&T, DBM(3X), *Unix Programmer's Manual, Seventh Edition, Volume 1*, January, 1979.

[ATT85] AT&T, HSEARCH(BA_LIB), *Unix System User's Manual, System V.3*, pp. 506-508, 1985.

[BRE73] Brent, Richard P., ''Reducing the Retrieval Time of Scatter Storage Techniques'', *Communications of the ACM*, Volume 16, No. 2, pp. 105-109, February, 1973.

[BSD86] NDBM(3), *4.3BSD Unix Programmer's Manual Reference Guide*, University of California, Berkeley, 1986.

[ENB88] Enbody, R. J., Du, H. C., ''Dynamic Hashing Schemes'', *ACM Computing Surveys*, Vol. 20, No. 2, pp. 85-113, June 1988.

[FAG79] Ronald Fagin, Jurg Nievergelt, Nicholas Pippenger, H. Raymond Strong, ‘‘Extendible Hashing -- A Fast Access Method for Dynamic Files’’, *ACM Transactions on Database Systems*, Volume 4, No. 3., September 1979, pp 315-34

[KNU68], Knuth, D.E., *The Art of Computer Programming Vol. 3: Sorting and Searching*, sections 6.3-6.4, pp 481-550.

[LAR78] Larson, Per-Ake, ‘‘Dynamic Hashing’’, *BIT*, Vol. 18, 1978, pp. 184-201.

[LAR88] Larson, Per-Ake, ‘‘Dynamic Hash Tables’’, *Communications of the ACM*, Volume 31, No. 4., April 1988, pp 446-457.

[LIT80] Witold, Litwin, ‘‘Linear Hashing: A New Tool for File and Table Addressing’’, *Proceedings of the 6th International Conference on Very Large Databases*, 1980.

[NEL90] Nelson, Philip A., *Gdbm 1.4 source distribution and README*, August 1990.

[THOM90] Ken Thompson, private communication, Nov. 1990.

[TOR87] Torek, C., ‘‘Re: dbm.a and ndbm.a archives’’, *USENET newsgroup comp.unix* 1987.

[TOR88] Torek, C., ‘‘Re: questions regarding databases created with dbm and ndbm routines’’ *USENET newsgroup comp.unix.questions*, June 1988.

[WAL84] Wales, R., ‘‘Discussion of "dbm" data base system’’, *USENET newsgroup unix.wizards*, January, 1984.

[YIG89] Ozan S. Yigit, ‘‘How to Roll Your Own Dbm/Ndbm’’, *unpublished manuscript*, Toronto, July, 1989

Margo I. Seltzer is a Ph.D. student in the Department of Electrical Engineering and Computer Sciences at the University of California, Berkeley. Her research interests include file systems, databases, and transaction processing systems. She spent several years working at startup companies designing and implementing file systems and transaction processing software and designing microprocessors. Ms. Seltzer received her AB in Applied Mathematics from Harvard/Radcliffe College in 1983.

In her spare time, Margo can usually be found preparing massive quantities of food for hungry hoards, studying Japanese, or playing soccer with an exciting Bay Area Women's Soccer team, the Berkeley Bruisers.

Ozan (**Oz**) Yigit is currently a software engineer with the Communications Research and Development group, Computing Services, York University. His formative years were also spent at York, where he held system programmer and administrator positions for various mixtures of of UNIX systems starting with Berkeley 4.1 in 1982, while at the same time obtaining a degree in Computer Science.

In his copious free time, Oz enjoys working on whatever software looks interesting, which often includes language interpreters, preprocessors, and lately, program generators and expert systems.

Oz has authored several public-domain software tools, including an nroff-like text formatter *proff* that is apparently still used in some basement PCs. His latest obsessions include the incredible programming language Scheme, and Chinese Brush painting.