Main–Memory Linear Hashing – Some Enhancements of Larson's Algorithm

Mikael Pettersson*
Department of Computer and Information Science
University of Linköping, S–581 83 Linköping, Sweden
email: mpe@ida.liu.se

December 8, 1993

Abstract

Linear Hashing has been proposed as a simple and efficient technique for storing and retrieving key sets whose cardinality is not known in advance. Its main advantage is that it allows tables to grow and shrink *gracefully* without major reorganizations. In this paper, several modifications of the basic scheme are presented, together with performance measurements. It is also shown that the seemingly–popular `hashpjw` hashing function presented in "the dragon book" can perform fairly poorly if the so–called *randomizing mod* operation is omitted. A simple and fast hashing function is presented that has performed well in this experiment. When combined, the algorithm changes presented here are shown to yield a performance improvement of roughly a factor of 4.5. The final version is shown to be about 16% slower than a standard hash table using table doubling on overflow. A sample implementation is provided in the appendix.

1 Introduction

Consider a standard hash table having B buckets with indices in $[0, B-1]$, where each bucket is a pointer to a collision list. Furthermore, let N denote the number of items in the table. The *load* of the table is defined as N/B, i.e. as the average length of the collision lists [1]. The time taken for basic operations like *lookup*, *delete*, and *enter*, is obviously proportional to the load of the table, if uniform distribution of keys and hash values is assumed.

In conventional *open hashing* [1], an overfull table is reorganized by allocating a new, larger table (for instance twice as large), and rehashing all elements in the old table to the new table. Afterwards, the old table can be deleted. Since these insertions in the new table take constant time (there is no need to check for duplicates), this process will take $O(N)$ time. As N grows, these delays may be undesirable or even unacceptable in interactive or real–time applications.

Linear Hashing [3], a technique originally developed for databases [4], avoids these $O(N)$ delays by amortizing the work over a longer period of time [2]. The idea is that the table is

*Partially supported by the Swedish National Board for Industrial and Technical Development (NUTEK).

[1] This definition differs slightly from the one commonly used in the context of data bases, since there the buckets typically are of fixed capacity, while here dynamically allocated pointer structures are used.

[2] Do not confuse linear *hashing* with linear *probing*, a technique used for dealing with collisions in *closed* hash tables.

expanded by one *bucket* at a time. Each expansion causes the elements of the left–most (i.e. lowest indexed) unsplit bucket of the initial table to be rehashed and split between this and the new bucket. Eventually, all buckets in the original part of the table have been split, and the table has doubled in size. At this point, the expansion process is restarted from index 0 again.

1.1 Overview of the paper

In section 2, the basic implementation of linear hashing is shown. Then, in section 3, a number of modifications of the basic scheme are presented and evaluated. Section 4 concludes the evaluation by comparing with a conventional hash table. Then the experimental setup is described (section 5), followed by the summary (section 6). The appendix contains the final version of the linear hashing code.

2 Implementation of Linear Hashing

Initially, the table contains $B = B_0$ empty buckets. It also maintains two indices, P (initially 0) and $MAXP$ (initially B_0). Whenever an insertion causes the load to be too high, i.e. $N/B > MAXLOADFACTOR$, an expansion is performed: a new bucket is appended at the (current) end of the table, B and P are both incremented by one, and the elements of bucket $P-1$ are rehashed and split between bucket $P-1$ and the new bucket $B-1$. Eventually, P reaches $MAXP$; at this time, the table has doubled its size from B_0 to $2B_0$ buckets, so P is reset to 0, $MAXP$ is doubled (the first time to $2B_0$, then to $4B_0$, and so on), before the splitting process continues.

A central issue is how to map hash codes to table indices. In a conventional hash table, $H \ mod \ B$ (where H is the hash code), is used as the index. Here, the table really contains $MAXP + P = B$ buckets, and each bucket with index $i < P$ has been split and had some of its elements moved to bucket $MAXP + i$. The buckets $[0, P[\ \cup \ [MAXP, MAXP + P[$ essentially belong to the "next" larger table which has $2MAXP$ buckets. The hash code to table index mapping is as follows:

$$hindex(H) = \begin{cases} H \ mod \ 2MAXP & \text{if } H \ mod \ MAXP \in [0, P[\quad \text{(split range)} \\ H \ mod \ MAXP & \text{if } H \ mod \ MAXP \in [P, MAXP[\quad \text{(unsplit range)} \end{cases}$$

Note that $H \ mod \ 2MAXP \in \{H \ mod \ MAXP, \ (H \ mod \ MAXP) + MAXP\}$.

To implement expandable arrays, a two–level structure is used. The array is partitioned into fixed–sized *segments* of buckets. The segments are referenced via a *directory*: an array of pointers to segments. Given a virtual index $index$, the corresponding collision list is located as follows (in Pseudo–C):

$$\begin{aligned} segmentptr &= directory[index \ div \ segmentsize] \\ list &= (*segmentptr)[index \ mod \ segmentsize] \end{aligned}$$

To expand the virtual array, it is first checked whether there is any room left in the last segment. If not, a new segment is allocated and its address is appended in the directory. Should the directory be filled, then a new, larger directory is allocated and the segment pointers are copied from the old to the new one.

If a deletion causes the load to fall below some threshold, then the inverse of the expansion process is invoked. P is first decremented by one; if it became less than zero, $MAXP$ is halved, and P is set to $MAXP - 1$. Then the elements of bucket $MAXP + P$ are merged into bucket P. Finally, if $(MAXP + P)$ *mod segmentsize* $= 0$, then the previous last segment is no longer accessible and can be freed.

3 Algorithm Modifications

In this section, several modifications of Larson's original code are presented and evaluated. The base version of the code uses Larson's algorithm for table expansion, and the `hashpjw` function [2, page 436] for computing hash values (since this hashing function is reputed to perform well). This is referred to as the *VANILLA* algorithm. Table 1 summarizes the timings and relative performance of all the versions. Furthermore, when testing some algorithm change, it is understood that all previous changes that lead to a time reduction are incorporated.

3.1 Caching the Hash Value

During an expansion, each element of the chosen bucket is rehashed in order to check whether its new address is the old or newly added bucket. By caching the hash value in the element, only the index calculation and comparison needs to be performed.

In the experiment, this optimization (alg. *SAVEHASH*) gave a 24.42% reduction in time, compared to *VANILLA*.

3.2 Cheaper Address Calculation

The address calculation in *hindex* uses two modulo operations: first `h % maxp` to compute the initial index, and then `h % (2*maxp)` if the initial index referred to an already split bucket. Now, it is easy to see that `maxp` is a power of 2 times the initial number of buckets B_0. If B_0, in turn, is a power of two, then so is `maxp`. Then these two modulo operations can be replaced by simpler bit–masking operations, i.e. `h & (maxp - 1)` and `h & (2*maxp - 1)` respectively.

In the experiment, this optimization (alg. *POW2*) reduced the time by 41.14% (compared to *SAVEHASH*), bringing the total savings to 55.51%.

It is also possible to maintain two variables having the values `maxp-1` and `2*maxp-1` respectively, as done in [6]. This "optimization" turned out to be of dubious value. Caching `maxp-1` (alg. *POW2LOMASK*) gave another 0.50% time reduction, while caching `2*maxp-1` (alg. *POW2HIMASK*) actually *worsened* the timings by 0.91%. The reason for this is probably that on typical RISC machines, it is often faster to load a single word from memory (e.g. `maxp` or `maxp_minus_1`) and do some simple arithmetic using only registers, than to load two words separately. Consequently, the *POW2HIMASK* version was not adopted.

3.3 Inlining the Index Calculation

The *hindex* function is called from two places: in the `enter` procedure to find the bucket to search/update, and in the table expansion routine `expandtable` when checking whether an element should be moved or not.

Inlining this very simple function (alg. *INLINEHIX*) in these two places gave a 2.78% reduction in time (compared to *POW2LOMASK*), bringing the total savings to 56.97%.

3.4 Cheaper Expansion Checking

Recall that the hashtable should be expanded whenever an insertion causes the load factor to exceed the desired maximum, i.e. `keycount / currentsize > maxloadfctr`. This test, which requires an integer division after each insertion, can be simplified based on the following transformations:

1. Introduce the invariant: `slack == currentsize * maxloadfctr - keycount`. Intuitively, `slack` indicates how many more entries can be added to the hash table before the load exceeds the desired maximum load factor.
2. Since `keycount` initially is zero, `slack` is initialized to B_0`*maxloadfctr`.
3. In `expandtable`, `currentsize` is incremented by one to reflect the addition of another bucket to the table. Therefore, `slack` is incremented by `maxloadfctr`.
4. In `enter`, `keycount` is incremented by one after a new symbol is added to the table. Therefore, `slack` is decremented by one.
5. Finally, the expansion check in `enter` reduces to `slack < 0`.

After the `slack` variable has been introduced, `currentsize` and `keycount` can be eliminated. They can be recomputed when needed as `maxp+p` and `(maxp+p)*maxloadfctr-slack` respectively.

In the experiment, this transformation (alg. *SLACK*) gave a 14.38% time reduction (compared to *INLINEHIX*), bringing the total savings to 63.15%.

3.5 Omitting the Randomizing Mod

As shown by the *POW2* and *SLACK* optimizations, integer divide/modulo operations are expensive (even when supported by hardware, they are often much slower than additions, shifts or bitwise operations). It is therefore tempting to also eliminate the *randomizing mod* that traditionally has been viewed as the normal last step of hashing functions. The impact of this is unclear. On the one hand, the hash computation should be faster. On the other hand, however, it is expected that the key distribution is worsened and consequently some of the buckets longer, which in turn means that insertions and lookups may take longer time.

In the experiment, this optimization (alg. *NOSCRAMBLE*) gave a 4.16% time reduction (compared to *SLACK*), bringing the total savings to 64.69%. As figures 3 and 4 show, the simplified hashing function tended to create a small number of very long buckets. However, these buckets were few enough that the speedup of the hashing function dominated.

3.6 A Better Hashing Function

As described in the previous paragraph, eliminating the randomizing mod can cause some buckets to become exceedingly long. Upon inspection, it was found that the keys in these very long buckets showed great similarities. In each long bucket, most keys shared a common

3– or 4–letter suffix, and furthermore were between 3 and 7 characters long. The reason for this clustering effect is that although the `hash_pjw` function (shown in figure 1) does have a feed–back component, it is never triggered for keys shorter than 7 characters [3]. The result is that the low 12–16 bits of the hash code are completely determined by the common suffix. Some of these common suffixes were `buf`, `ptr`, `line` and `time`.

Several different hashing functions were tried, all of which utilized a scheme were the intermediate hash value was multiplied by an odd constant (thus also adding it to itself) before adding the next character of the key. The one that gave the best overall performance is shown in figure 2 [4].

```
unsigned hash_pjw(str, len)
    unsigned char *str;
    int len;
{
    unsigned long h = 0, g;

    while( --len >= 0 ) {
        h = (h << 4) + *str++;
        /* assume 32-bit integers */
        if( (g = h & 0xf0000000) != 0 ) {
            h ^= g >> 24;
            h ^= g;
        }
    }
    /* note: no scrambling */
    return (unsigned int)h;
}
```

Figure 1: the `hash_pjw` hashing function adapted from [2, page 436]

In the experiment, this modification (alg. *HASHX33*) gave a 28.06% time reduction (compared to *NOSCRAMBLE*), bringing the total savings to 74.60%. As shown in figures 4 and 5, the new hash function exhibits a *much* improved key distribution compared to the original hash function without the randomizing mod. The distribution is even better than the one exhibited when `hash_pjw` *does* use a randomizing mod (figure 3).

It should be emphasized that achieving near–optimal distribution properties in hashing functions is *not* is goal in itself (at least not in this context). Instead, the overall goal is to speed up the *combination* of hashing and collision processing. Spending M cycles in the hashing function can only be justified by the corresponding saving of at least M cycles in the collision processing. If, as seems to be the case here, collision processing is reasonably cheap, then the hashing function must also be cheap.

Finding the right tradeoffs can be difficult. However, as demonstrated here, conventional

[3] Applying `hash_pjw` to a 6–element string of 7–bit characters generates a value with at most 28 bits.

[4] The "times 33" idea seems to have been used for a number of years; the originator *may* be Dan Bernstein, although some sources (e.g. the code in [6]) attributes it to Chris Torek. See [5] for a thorough analysis of hashing functions.

```
unsigned hash_x33(str, len)
    unsigned char *str;
    int len;
{
    unsigned h = 0;
    while( --len >= 0 )
        h = (h << 5) + h + *str++;
    return h;
}
```

Figure 2: a better hashing function

wisdom ("scambling" is good, `haspjw()` is good, ...) can be false, and should therefore be reevaluated when appropriate rather than blindly adopted "on faith."

3.7 Unrolling the hash function

In order to further reduce the overhead of the hashing function, the test was re–run with the body of the inner loop was unrolled between 2 and 8 times. The best improvement came when the loop was unrolled 4 times. The worst results came when the loop was unrolled 7 times; in this case the performance dropped by more than a factor of two!

Unrolling the loop four times (alg. *UNROLL4*) gave a 2.41% time reduction (compared to *HASHX33*), bringing the total savings to 75.21%.

3.8 Cheaper Split Test

When the table is expanded, all the elements of the bucket to be split are rehashed and have their indices recomputed in order to determine which elements should stay in the old bucket, and which are to be moved to the new one.

Let $MAXP_0$ and P_0 denote the values of `maxp` and `p` respectively on entry to the table expansion routine. The bucket to be split is P_0 and $newaddress = MAXP_0 + P_0$. Since $P_0 \in [P_0, MAXP_0[$, we have that $hindex(hash(x)) = hash(x) \ mod \ MAXP_0 = P_0$ for every element x in bucket P_0.

The update code for the state variables is:

```
p = p + 1;
if( p == maxp ) {
    p = 0;
    maxp = maxp * 2;
}
```

Let $MAXP_1$ and P_1 denote the values of `maxp` and `p` respectively after the update has been performed. Now consider some element x in bucket P_0, and let h be its hash value. x should be moved only if $hindex(h) = newaddress$. There are two cases:

1. $P_0 \in [0, MAXP_0 - 1[$. This entails that $P_1 = P_0 + 1$ and $MAXP_1 = MAXP_0$. Substituting these values in the definition of *hindex* gives:

$$hindex(h) = \begin{cases} h \ mod \ 2MAXP_0 & \text{if } h \ mod \ MAXP_0 \in [0, P_0 + 1[\\ h \ mod \ MAXP_0 & \text{if } h \ mod \ MAXP_0 \in [P_0 + 1, MAXP_0[\end{cases}$$

But $h \ mod \ MAXP_0 = P_0$, so this reduces to:

$$hindex(h) = h \ mod \ 2MAXP_0$$

2. $P_0 = MAXP_0 - 1$. This entails that $P_1 = 0$ and $MAXP_1 = 2MAXP_0$. Substituting these values in the definition of *hindex* gives:

$$hindex(h) = \begin{cases} h \ mod \ 4MAXP_0 & \text{if } h \ mod \ 2MAXP_0 \in [0, 0[\\ h \ mod \ 2MAXP_0 & \text{if } h \ mod \ 2MAXP_0 \in [0, MAXP_0[\end{cases}$$

But the first case can never apply, so this reduces to:

$$hindex(h) = h \ mod \ 2MAXP_0$$

Thus, the element x should be moved iff $h \ mod \ 2MAXP_0 = MAXP_0 + P_0$.

Since `maxp0` is a power of two, this can be expressed as `(h & (2*maxp0-1)) == maxp0+p0`, which is true iff `(h & maxp0) == maxp0`, or equivalently `(h & maxp0) != 0`.

In the experiment, this optimization (alg. *SPLITBIT*) gave a 6.34% time reduction (compared to *UNROLL4*), bringing the total savings to 76.78%.

3.9 Cheaper collision list search

Searching a bucket (collision list) is done for each string entered to or looked up from the hash table. Upto this point, this search has used a simple string comparison: first compare the lengths and then the strings themselves. Since alg. *SAVEHASH*, the hash code is stored with the strings in the hash table. This can be used to augment the comparison operation: first compare the hash values, and only if they were equal compare the strings.

In the experiment, this optimization (alg. *CMPHASH*) gave a 4.66% time reduction (compared to *SPLITBIT*), bringing the total savings to 77.86%.

3.10 Increasing the Load Factor

Upto this point, the algorithms had been run with a loading factor of 2. To measure the impact of an increased loading factor, the *CMPHASH* algorithm was re–run with loading factors of 4, 6, 8 and 10 (algorithms *LOAD4* to *LOAD10*). As indicated by table 1, the time increase was faily modest and apparently linear. Larson's conclusion [3] that the loading factor can be a high as 5 without sacrificing performance much seems to be true here as well.

4 Comparison with a standard hash table

Linear hashing, while offering incremental behaviour during table expansion or reduction, does impose some overhead in its slightly more complicated address calculations and segmented table structure.

algorithm version	machine cycles	cycle percentages normalized	relative	
VANILLA	48542670	100.00	n/a	
SAVEHASH	36688147	75.58	-24.42	
POW2	21595335	44.49	-41.14	
POW2LOMASK	21487824	44.27	-0.50	
POW2HIMASK	21684030	44.67	+0.91	(*not used*)
INLINEHIX	20890131	43.03	-2.78	
SLACK	17886721	36.85	-14.38	
NOSCRAMBLE	17142615	35.31	-4.16	
HASHX33	12331932	25.40	-28.06	
UNROLL4	12034553	24.79	-2.41	
SPLITBIT	11271313	23.22	-6.34	
CMPHASH	10746067	22.14	-4.66	
LOAD4	11821701	24.35	+10.01	
LOAD6	12716198	26.20	+18.33	
LOAD8	13499894	27.81	+25.63	
LOAD10	14437314	29.74	+34.35	
DBLHASH	9242910	19.04	-13.99	

Table 1: algorithm timings

In order to estimate this overhead, a conventional hash table was implemented. When overfull, it used the table doubling technique described in the introduction to reduce the load. All the optimizations described here for linear hashing were used for the conventional table as well.

As shown in table 1, the conventional table (alg. *DBLHASH*) was 13.99% faster than alg. *CMPHASH*. This means that *CMPHASH* is about 16% slower than *DBLHASH*.

5 Experimental Setup

The experiments were conducted on a Sun SPARCstation ELC under SunOS 4.1.3.

First, unique symbol occurrences were gathered from several sources: a portion of the Berkeley 4.3BSD–NET2 distribution, a portion of the local GNU sources, the author's collection of Scheme code (much of which was machine–generated), and the entire `/usr/dict/words`. A total of 51395 unique symbols were collected into a file in random order.

The hash table's segment and directory sizes were both 256, as was the initial number of buckets. The initial loading factor was 2. The hash table code itself was a direct translation to C of the Pascal code given in [3], using `hashpjw` as the hashing function. Sun's ordinary, bundled C compiler was used to compile the code, at maximum optimization level (O4). The test program was set up to first read the symbol file into primary memory, and then iterate through the symbols, while inserting each occurrence in the hash table.

All memory needed for the different data structures was pre–allocated. No system or library calls (except calls to `memcmp` to compare strings) were made by the hash table code.

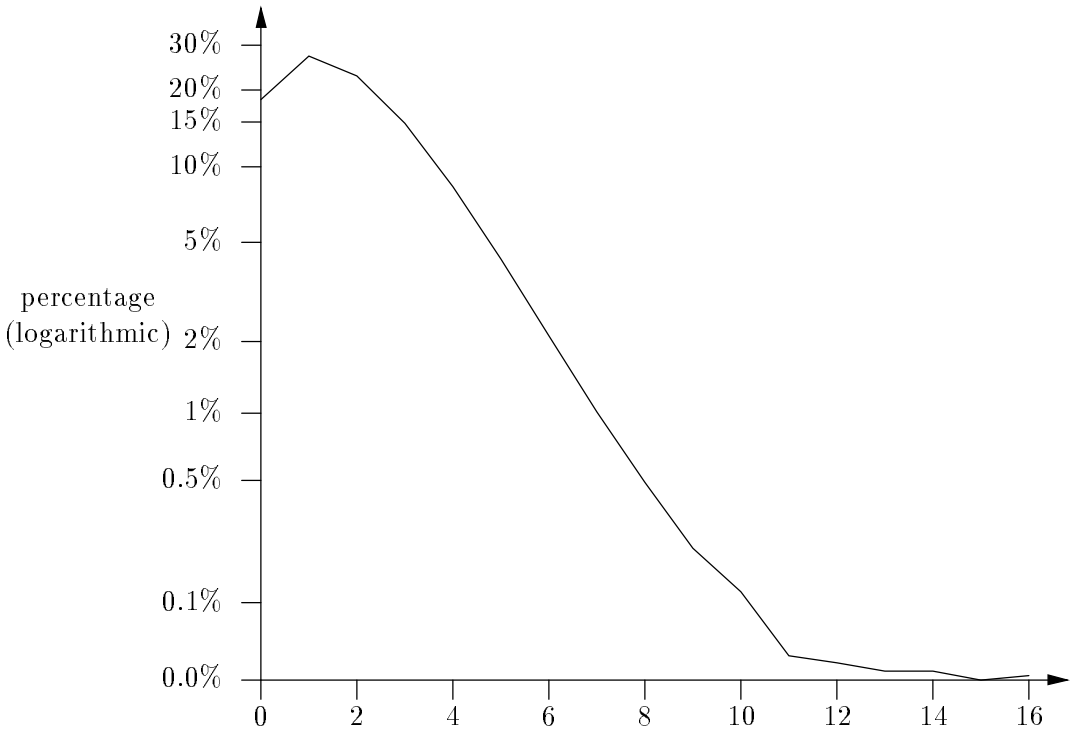

Figure 3: bucket length distributions: algorithm VANILLA

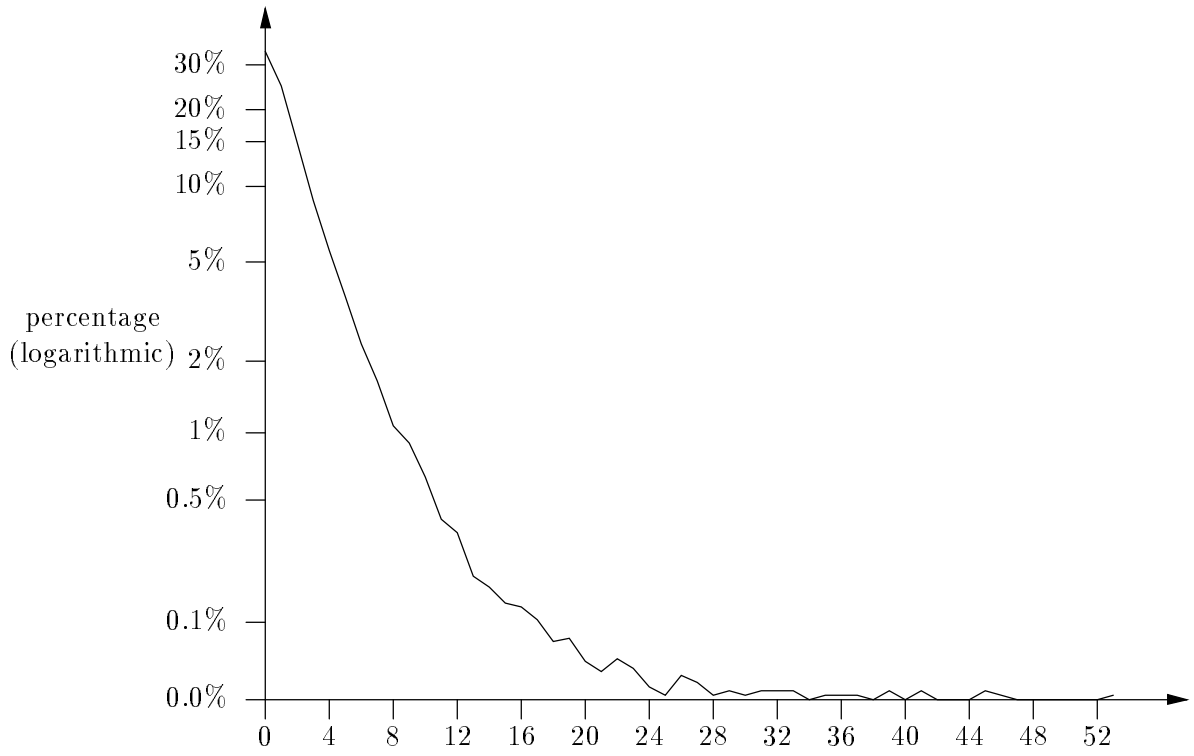

Figure 4: bucket length distributions: algorithm NOSCRAMBLE

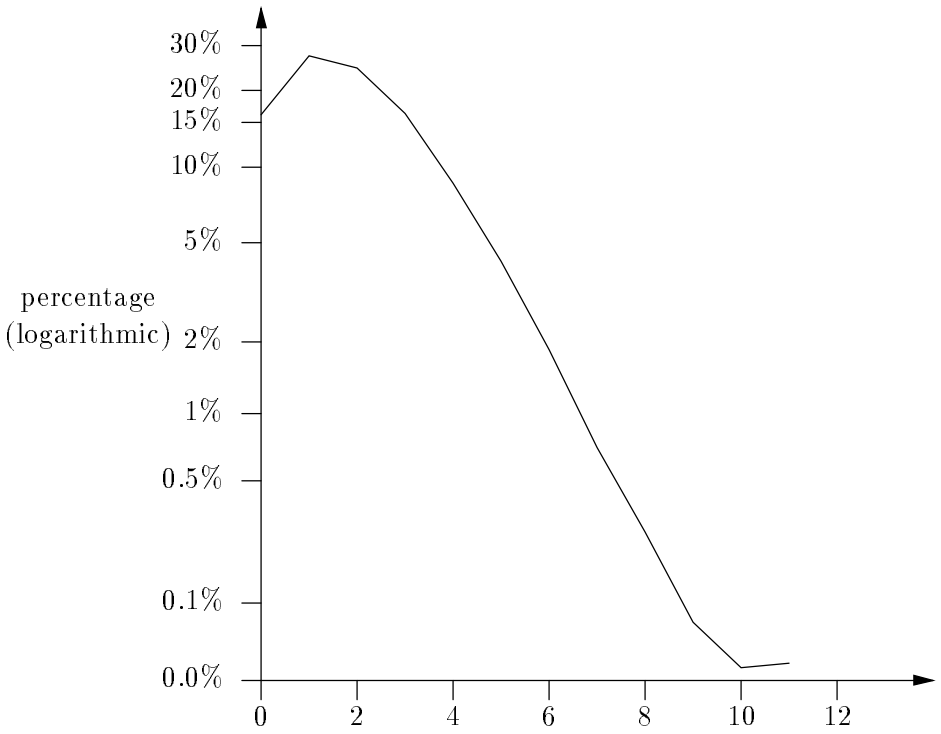

Figure 5: bucket length distributions: algorithm HASHX33

In order to acquire consistent timings, the code was executed under an instruction–level simulator for the SPARC processor. For each test, the sum of instruction cycles, anulled delay slot cycles, and register load–use stall cycles was used as the time measure, *after* the same timings for a special *dryrun* version of the code been subtracted. (The dryrun version performed all the initialization actions and iteration, but performed no calls to the `enter` routine; the reported timings therefore include only the cycles spent in the hash table module itself.)

The reason for using the simulator (which was painfully slow) was that the wall–clock timings reported by the SunOS `getrusage` system call varied wildly and apparently randomly. No matter what precautions were made (link the object file statically; run an essentially inactive stand–alone machine with no other user processes or even a network connection; run the application at highest scheduling priority; iterate the tests a large number of times), the timings for any particular version of the algorithm could vary 10–60% from one run to another. The reasons for this variation are still unknown to this author.

6 Summary

Several effective optimizations of Larson's basic Linear Hashing scheme have been presented. They are all fairly obvious and seem trivial; their combination, however, yielded a performance improvement of roughly a factor of 4.5 in this experiment. It has also been indicated that increasing the load factor only causes a modest, linear slowdown.

A weakness of the `hashpjw` hash function has been identified, and an alternative has been presented that seems to perform very well even though the *randomizing mod* is omitted.

If the hash table is carefully implemented, a simple and fast hashing function may very well lead to better overall performance than a slower, more complicated function that supposedly exhibits better distribution.

Acknowledgements

Gordon Irliam wrote the SPARC Performance Analysis package (SPA) that was used to acquire the detailed and consistent timings this study depended on.

References

[1] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *Data Structures and Algorithms.* Addison–Wesley, 1983.

[2] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers – Principles, Techniques, and Tools.* Addison–Wesley, 1986.

[3] Per–Åke Larson. *Dynamic Hash Tables.* In *Communications of the ACM* 31(4), pages 446–457, April 1988.

[4] Witold Litwin. *Linear Hashing: A new tool for file and table addressing.* In *Proceedings of the 6th International Conference on Very Large Data Bases (VLDB)*, 1980.

[5] B. J. McKenzie, R. Harries, and T. Bell. *Selecting a Hashing Algorithm.* In *Software – Practice and Experience*, 20(2), pages 204–224, Feb. 1990.

[6] Margo Seltzer, and Ozan Yigit. *A New Hashing Package for UNIX.* In *Proceedings of the USENIX – Winter '91 Conference.* The package and paper is also part of the 4.3BSD–NET2 distribution.

Appendix: The Final Algorithm

```
/* linhash.c */
#include <memory.h>     /* get memcmp() */
#include <string.h>     /* get strdup() */
extern void *emalloc(); /* like malloc, but calls exit() on failure */
extern void free(void*);

/* The 'hash_x33' function unrolled four times. */

unsigned hash_x33_u4(str, len)
    register unsigned char *str;
    register int len;
{
    register unsigned int h = 0;

    for(; len >= 4; len -= 4) {
        h = (h << 5) + h + *str++;
        h = (h << 5) + h + *str++;
        h = (h << 5) + h + *str++;
        h = (h << 5) + h + *str++;
    }
    switch( len ) {
      case 3:
        h = (h << 5) + h + *str++;
        /*FALLTHROUGH*/
      case 2:
        h = (h << 5) + h + *str++;
        /*FALLTHROUGH*/
      case 1:
        h = (h << 5) + h + *str++;
        break;
      default:  /*case 0:*/
        break;
    }
    return h;
}
#define HASH(str,len)   hash_x33_u4((str),(len))

/* SEGMENTSIZE must be a power of 2 !! */
#define SEGMENTSIZE     256
#define DIRECTORYSIZE   256
#define DIRINDEX(address)       ((address) / SEGMENTSIZE)
#define SEGINDEX(address)       ((address) % SEGMENTSIZE)
#if     !defined(MAXLOADFCTR)
#define MAXLOADFCTR     2
#endif  /*MAXLOADFCTR*/
```

```
#if     !defined(MINLOADFCTR)
#define MINLOADFCTR     (MAXLOADFCTR/2)
#endif  /*MINLOADFCTR*/

typedef struct element {
    unsigned            len;
    unsigned            hash;
    char                *str;
    struct element      *next;
} element_t;

#define new_element()   (element_t*)emalloc(sizeof(element_t))

typedef struct segment {
    element_t   *elements[SEGMENTSIZE];
} segment_t;

#define new_segment()   (segment_t*)emalloc(sizeof(segment_t))

typedef struct hashtable {
    unsigned    p;              /* next bucket to split */
    unsigned    maxp_minus_1;   /* == maxp - 1; maxp is the upper bound */
    int         slack;          /* number of insertions before expansion */
    segment_t   *directory[DIRECTORYSIZE];
} hashtable_t;

/* initialize the hash table `T' */

void init_hashtable(T)
    hashtable_t *T;
{
    T->p                = 0;
    T->maxp_minus_1     = SEGMENTSIZE - 1;
    T->slack            = SEGMENTSIZE * MAXLOADFCTR;
    T->directory[0]     = new_segment();
    {   /* the first segment must be entirely cleared before used */
        element_t **p = &T->directory[0]->elements[0];
        int count = SEGMENTSIZE;
        do {
            *p++ = (element_t*)0;
        } while( --count > 0 );
    }
    {   /* clear the rest of the directory */
        segment_t **p = &T->directory[1];
        int count = DIRECTORYSIZE - 1;
        do {
            *p++ = (segment_t*)0;
```

```
        } while( --count > 0 );
    }
}

void expandtable(T)
    hashtable_t *T;
{
    element_t **oldbucketp, *chain, *headofold, *headofnew, *next;
    unsigned maxp0 = T->maxp_minus_1 + 1;
    unsigned newaddress = maxp0 + T->p;

    /* no more room? */
    if( newaddress >= DIRECTORYSIZE * SEGMENTSIZE )
        return; /* should allocate a larger directory */

    /* if necessary, create a new segment */
    if( SEGINDEX(newaddress) == 0 )
        T->directory[DIRINDEX(newaddress)] = new_segment();

    /* locate the old (to be split) bucket */
    oldbucketp = &T->directory[DIRINDEX(T->p)]->elements[SEGINDEX(T->p)];

    /* adjust the state variables */
    if( ++(T->p) > T->maxp_minus_1 ) {
        T->maxp_minus_1 = 2 * T->maxp_minus_1 + 1;
        T->p = 0;
    }
    T->slack += MAXLOADFCTR;

    /* relocate records to the new bucket (does not preserve order) */
    headofold = (element_t*)0;
    headofnew = (element_t*)0;
    for(chain = *oldbucketp; chain != (element_t*)0; chain = next) {
        next = chain->next;
        if( chain->hash & maxp0 ) {
            chain->next = headofnew;
            headofnew = chain;
        } else {
            chain->next = headofold;
            headofold = chain;
        }
    }
    *oldbucketp = headofold;
    T->directory[DIRINDEX(newaddress)]->elements[SEGINDEX(newaddress)]
            = headofnew;
}
```

```
/* insert string ‘str’ with ‘len’ characters in table ‘T’ */

void enter(str, len, T)
    unsigned char *str;
    int len;
    hashtable_t *T;
{
    unsigned hash, address;
    element_t **chainp, *chain;

    /* locate the bucket for this string */
    hash = HASH(str, len);
    address = hash & T->maxp_minus_1;
    if( address < T->p )
        address = hash & (2 * T->maxp_minus_1 + 1);

    chainp = &T->directory[DIRINDEX(address)]->elements[SEGINDEX(address)];

    /* is the string already in the hash table? */
    for(chain = *chainp; chain != (element_t*)0; chain = chain->next)
        if( chain->hash == hash &&
            chain->len == len &&
            !memcmp(chain->str, str, len)
          )
            return;     /* already there */

    /* nope, must add new entry */
    chain = new_element();
    chain->len = len;
    chain->hash = hash;
    chain->str = strdup((char*)str);
    chain->next = *chainp;
    *chainp = chain;

    /* do we need to expand the table? */
    if( --(T->slack) < 0 )
        expandtable(T);
}

void shrinktable(T)
    hashtable_t *T;
{
    segment_t *lastseg;
    element_t **chainp;
    unsigned oldlast = T->p + T->maxp_minus_1;

    if( oldlast == 0 )
```

```
        return; /* cannot shrink below this */

    /* adjust the state variables */
    if( T->p == 0 ) {
        T->maxp_minus_1 >>= 1;
        T->p = T->maxp_minus_1;
    } else
        --(T->p);
    T->slack -= MAXLOADFCTR;

    /* insert the chain `oldlast' at the end of chain `T->p' */
    chainp = &T->directory[DIRINDEX(T->p)]->elements[SEGINDEX(T->p)];
    while( *chainp != (element_t*)0 )
        chainp = &((*chainp)->next);
    lastseg = T->directory[DIRINDEX(oldlast)];
    *chainp = lastseg->elements[SEGINDEX(oldlast)];

    /* if necessary, free the last segment */
    if( SEGINDEX(oldlast) == 0 )
        free((void*)lastseg);
}

/* delete string `str' with `len' characters from table `T' */

void delete(str, len, T)
    unsigned char *str;
    int len;
    hashtable_t *T;
{
    unsigned hash, address;
    element_t **prev, *here;

    /* locate the bucket for this string */
    hash = HASH(str, len);
    address = hash & T->maxp_minus_1;
    if( address < T->p )
        address = hash & (2 * T->maxp_minus_1 + 1);

    /* find the element to be removed */
    prev = &T->directory[DIRINDEX(address)]->elements[SEGINDEX(address)];
    for(; (here = *prev) != (element_t*)0; prev = &here->next)
        if( here->hash == hash &&
            here->len == len &&
            !memcmp(here->str, str, len)
          )
            break;
    if( here == (element_t*)0 )
```

```
        return; /* the string wasn't there! */

    /* remove this element */
    *prev = here->next;
    free((void*)here->str);
    free((void*)here);

    /* do we need to shrink the table? the test is:
     *          keycount / currentsize < minloadfctr
     * i.e.     ((maxp+p)*maxloadfctr-slack) / (maxp+p) < minloadfctr
     * i.e.     (maxp+p)*maxloadfctr-slack < (maxp+p)*minloadfctr
     * i.e.     slack > (maxp+p)*(maxloadfctr-minloadfctr)
     */
    if( ++(T->slack) >
        (T->maxp_minus_1 + 1 + T->p) * (MAXLOADFCTR-MINLOADFCTR)
      )
        shrinktable(T);
}
```