A Practical Perfect Hashing Algorithm

Craig Silverstein[*]
Computer Science Department
Stanford University
Stanford, CA 94305
csilvers@cs.stanford.edu

8 August 1998

Abstract

Hashing is a fundamental technique in Computer Science, and perfect hashing, which guarantees constant-time lookup, has particular theoretical and practical importance. The algorithm of Fredman, Komlós, and Szemerédi [8], augmented by Dietzfelbinger, *et al.* [7], is a simple perfect hashing scheme that has not seen extensive use due to poor practical performance. In this paper, we develop modifications to the previous techniques that result in a fast, space-efficient algorithm. We show the resulting algorithm is comparable to hash algorithms used in several popular dictionary libraries but is more flexible and has better theoretical performance.

1 Introduction

The *dictionary* abstract data type — which provides insertion, deletion, and lookup of arbitrary items — is one of the most important in computer science, having applications from databases to function optimization. Arguably the most popular implementation of the dictionary data type is via *hashing*, in which items are kept in an array of buckets, indexed by small integers, and a *hash function* is used to map items to buckets. Using hashing it is possible to get constant-time lookup and (amortized) constant-time insertion and deletion. Alternative implementations of the dictionary data type, such as B-trees, do not seem to be able to match these time bounds.

Early hashing algorithms could only guarantee *average case* constant-time lookup (see, e.g., [9]), assuming a friendly distribution of inputs. For instance, lookup in linear probing hashing schemes could take linear time for lookup if all the keys hashed to a small range of values. In the past decade research has focused on algorithms for *perfect* (i.e., collision-free) hash functions, which guarantee constant-time lookup even in the worst case. Fredman, Komlós, and Szemerédi developed a particularly appealing randomized hashing algorithm [8], combining $O(1)$ worst case lookup time with $O(n)$ worst case space use. This algorithm did not support insertion or deletion but only lookup on a static data set. However, Dietzfelbinger, *et al.* [7] remedied this shortcoming, describing a modified algorithm with amortized expected constant time insertion and deletion. We call this modified algorithm *FKS hashing*.

FKS hashing is easy to implement; its major shortcoming is that the order notation hides some fairly large constants. The lookup algorithm, for instance, multiplies two large numbers and is

[*]Supported by the Department of Defense, with partial support from NSF Award CCR-9357849, with matching funds from IBM, Schlumberger Foundation, Shell Foundation, and Xerox Corporation.

therefore quadratic in the number of bits in the largest key. Furthermore, a naive implementation uses $37n$ space, which is unacceptably large in practice. A somewhat slower implementation uses only $2n + o(n)$ space, but even this bound hides some practically significant constants in the $o(n)$ term.

In this paper, we attack these time and space issues, attaining an algorithm that is comparable to hashing schemes that lack the theoretical guarantees of the FKS algorithm. In the rest of this section, we describe the canonical FKS algorithm. In Section 2 we describe how we evaluate the various algorithms discussed in this paper. In Section 3 we describe a series of modifications to the canonical FKS scheme that yields a faster, more compact variant of FKS. In Sections 4 and 5 we discuss various ways to make hashtables smaller, at the cost of speed: by using space-efficient tables (Section 4 and by adjusting load factor parameters (Section 5). Section 6 compares the FKS class of hash table algorithms to other hash table algorithms, including some in widely used applications such as Perl. Finally, in Section 7 we conclude with a discussion of the feasibility of using FKS-based hashing algorithms in practice.

1.1 Hashing with Linear Probing

We start by describing a simple, fast hashing algorithm similar to ones used in such publicly available dictionary libraries as LEDA [4] and AT&T System V's `hsearch` [5].[1] Assume for the moment we know we are going to insert n items. We construct the hashtable, which is an array of size $t > n$. We then choose an arbitrary hash function $h(x)$ that maps a key x to an integer in $\{1, \ldots, t\}$. $h(x)$ should be chosen so that, for most data, different keys will map to different values. Constructing a general, high quality hash function is a difficult task; [2] discusses some of the issues involved. For a good hash function, the time to compute $h(x)$ should be linear in the number of bits in x. Superlinear hash functions are too slow, while faster ones cannot look at all the input and are susceptible to collisions in keys that differ only on the unexamined bits.

To insert a key x, we first look at bucket $h(x)$ in the hashtable. If it is empty, we insert x at that position. If it is already occupied, we have a *collision*, and we instead try to insert x at position $h(x) + 1$. If that bucket is occupied we try $h(x) + 2$, and so on, wrapping around if necessary, until we find an empty bucket. If $h(x)$ causes few collisions on the data set, and the table is not too full, the expected time to insert will be small.

Lookup proceeds similarly to insertion. To look up x, we first examine bucket $h(x)$. If it holds x, we terminate successfully. If it is empty, we terminate unsuccessfully. If it holds some key other than x, we examine bucket $h(x) + 1$, then $h(x) + 2$, and so on, until we find x or an empty bucket. The worst-case time to look up x is the same as the time to insert x, while the average-case time, if x actually is in the hashtable, is about half that.

To delete x, we are tempted to merely look up x and empty the returned bucket. This may cause lookup to fail, however, since it will now terminate (incorrectly) when examining bucket $h(x)$ while before it would have continued to look at $h(x) + 1$. This can cause lookup to miss a key y that had collided with x and is therefore in bucket $h(x) + 1$. The solution is to insert a special "place holder" into bucket $h(x)$ indicating it holds deleted contents. Lookup treats the bucket as if it holds a non-matching key, while insert treats it as empty.

If the total number of insertions n is not known, the hashtable can be resized dynamically. When the number of filled buckets gets too close to t ($t/2$ is a good maximum value in practice, since as the number of empty buckets gets small, insertion and lookup times increase dramatically), the entire contents are rehashed into a hashtable of size $2t$. Likewise, if the number of filled buckets

[1] LEDA actually performs hashing with chaining, while `hsearch` performs double hashing. These approaches are qualitatively similar to hashing with linear probing. See [9] for a discussion of the relative merits of these approaches.

gets too small (say, $t/5$), the entire hashtable can be rehashed into a table of size $t/2$. (Such rehashing is necessary to keep the space use linear in n but does not improve future running time.) Rehashing takes amortized constant time.

For computational reasons, it is advantageous to have t be a power of 2; then the last step of computing $h(x)$ is merely a bit mask. Note that rehashing preserves the property that t is a power of 2.

1.2 Dynamic Perfect Hashing

If the hash function described above could guarantee no collisions for every data set and every potential value of t, it would be a *perfect* hash function. It is not hard to believe that there is no perfect hash function that works for all possible inputs, but it turns out there is a simple *family* of hash functions such that, for any data set and any value of t, most hash functions in the family result in a very small number of collisions. Fredman, Komlós, and Szemerédi, in [8], leveraged this into a perfect hashing algorithm, one in which there are no collisions and all dictionary operations are therefore guaranteed constant time.

Suppose our (static) input consists of n integers in the range $1 \ldots p-1$, where p is a prime.[2] Given a hashtable with t buckets, and an arbitrary parameter $k < p$, we define the hash function $h_k(x)$ to be $(kx \bmod p) \bmod t$. Let $C_k(i)$ be the set of input keys that hash into bucket i $(1 \leq i \leq t)$. From [8] we have that

$$\sum_{k=1}^{p-1} \sum_{i=1}^{t} \binom{|C_k(i)|}{2} < \frac{(p-1)n^2}{t} \tag{1}$$

This implies that, with probability $\frac{1}{2}$, a random k satisfies

$$\sum_{i=1}^{t} \binom{|C_k(i)|}{2} < \frac{2n^2}{t}$$

Clearly $\sum_{i=1}^{t} |C_k(i)| = n$, so a random k, with probability $\frac{1}{2}$, satisfies

$$\sum_{i=1}^{t} |C_k(i)|^2 < \frac{4n^2}{t} + n \tag{2}$$

If $t \in \Omega(n)$, this equation limits, but does eliminate, collisions: $|C_k(i)|$, the number of items in bucket i, can be greater than 1. Therefore we hash the items in each bucket a second time. With each bucket i we associate a second level hashtable of size $t_i = 2|C_k(i)|^2$. The inequality (2) again applies, with n set to $|C_k(i)|$ and t set to $2n^2$, and it can only be satisfied if each bucket holds at most one item. With probability $\frac{1}{2}$, then, our choice of k_i yields a perfect hash function for bucket i. It is easy to see that, in constant expected time, we end up with a perfect hash function for each second level hashtable. The space requirement is $t + \sum_{i=1}^{t} t_i$. If $t = n$, we can apply (2) to bound $\sum t_i$, giving a bound of $11n$ on the number of buckets.

To handle the dynamic case, with insertions and deletions, [7] starts with the static hashing scheme, but the authors make the top level hashtable larger than necessary. In particular, when hashing n_0 items they pretend there are really $(1+c)n_0$ items to be inserted. When the number of items actually grows to $(1+c)n_0$, or drops to $\frac{1+c}{1+2c}n_0$, they rehash the entire hashtable. This rehashing takes linear time but can be amortized over the cn_0 (or more) insertions and deletions

[2] We can easily handle arbitrary data by considering any series of bits to represent a (very large) integer.

that must have taken place since the previous rehashing. We can also rehash if the total number of low-level buckets grows too large, but this does not affect the time analysis. Note this is essentially the same rehashing technique as described in Section 1.1.

The second level hashtables are dealt with in a similar way. If bucket i has $n_0 = |C_k(i)|$ items, we size the second level hashtable as if it held $(1+c')n_0$ items, and we rehash it when the number of items grows to $(1+c')n_0$ or shrinks to $\frac{1+c'}{1+2c'}n_0$. Unlike at the top level, each insertion may require that we re-choose k_i to preserve non-collision. However, with probability $\frac{1}{2}$ each choice of k_i can hash all $(1+c')n_0$ items without collision, so we expect to have to look at only 2 possible k values between rehashings. As a result, overall insertion and deletion take amortized constant time. It is also easy to show the total space required is linear throughout the lifetime of the algorithm.

2 Experimental Setup

All of the modifications to FKS suggested in this paper are heuristics; how well they work, and indeed if they work at all, can only be determined by experiment. Before describing changes to the FKS algorithm, we describe the experimental setup used to test these changes. This is the setup provided by the DIMACS Challenge organizers, along with evaluation metrics appropriate to this study.

2.1 The Test Suite

The test set suite used is the one provided for the Dictionary portion of the DIMACS challenge (see [3]). It is summarized in Table 1. In our analysis we ignore the smaller test sets and only examine the following 34 test sets: `matchexact.*` (26 test sets; we omitted the 100-element subset of `matchexact.trace-3.2-Y-1`), `id.c`, `is.c`, `iu.c`, `iid.c`, `iisd.c`, `iiud.c`, `circ.100000`, and `joyce.dat`.

For those test sets that take a random number seed, we ran the test three times with different seeds, averaging the running times. We also averaged the results of the five `circ` test sets of a given size. In no case was the variance of the tests large enough to be a concern.

All tests were run on three separate machines:

Alpha an AlphaServer 4100 5/300 with 4 EV5 300 MHz. processors and 4 Gig. memory running OSF1 V4.0 and including gcc 2.7.2;

POWER an RS/6000 604e with 4 POWER 166 Mhz. processors and 512 Meg. of memory running AIX 4.2 and including gcc 2.8.1; and

x86 a Pentium Pro with a 200 Mhz. processor and 96 Meg. of memory running Solaris x86 5.5.1 and including gcc 2.8.1.

In general, we will refer to each machine by the bold processor name. In each case, the code — written in C — was compiled using gcc with the `-O6` optimization option.

We chose these three configurations in order to test a variety of processor types, word sizes, and cache configurations. Such variety is particularly important in low-level data structures such as dictionaries, which can depend on specific properties of the architecture for improved performance.

Our implementation of the hash function, for instance, operates on a word-by-word basis. This has several consequences. Machines with 64-bit architectures can, in theory, hash more quickly than 32-bit machines because the same bit string will fit into fewer words. Furthermore, hashing strings will be different on big-endian and little-endian hardware, because the sequence of characters

Test Set	Key size	description
matchexact.*	4 bytes (int)	Dictionary traces of ML and Smalltalk programs.
circ.*n*	36 bytes	Library call data (*n* is number of records).
id.*n*	16 bytes	Inserts of random keys followed by deletion. *n* is one of `a` (1000 items), `b` (10000 items), or `c` (100000 items).
is.*n*	16 bytes	Inserts of random keys followed by successful lookups. *n* is as in `id`.
iu.*n*	16 bytes	Inserts of random keys followed by unsuccessful lookups. *n* is as in `id`.
iid.*n*	16 bytes	Inserts of random keys followed by insert-then-delete operations. *n* is as in `id`.
iisd.*n*	16 bytes	Inserts of random keys followed by insert-then-successful-lookup-then-delete operations. *n* is as in `id`.
iiud.*n*	16 bytes	Inserts of random keys followed by insert-then-unsuccessful-lookup-then-delete operations. *n* is as in `id`.
ed.*t*	20 bytes	`ed.`*t* is equivalent to *t*`.a` — where *t* is one of `id`, `iu`, etc. as above — for words in a book by Eddington. For example, *ed.iid* inserts 1000 words, then does 1000 insert-delete operations.
jy.*t*	20 bytes	Same as for `ed`, but using words from a book by Joyce.
*.dat	20 bytes	Lookup followed by insertion (if necessary) of words from various books.

Table 1: DIMACS standard suite of Dictionary test sets.

will be interpreted as different numbers in each case. In order to explore these issues, we chose test platforms that include both 32- and 64-bit architectures and both big- and little-endian word orders. In particular, the x86 is 32-bit and little-endian, the POWER is 32-bit and big-endian, and the Alpha is 64-bit and little-endian.

2.2 Evaluating Modifications of the FKS Algorithm

We would like to obtain a single number encapsulating how fast, or how small, a given variant of the FKS algorithm is. To do this with the 34 test sets we include in our analysis, we report the performance of each algorithm relative to the performance of the canonical FKS algorithm as described in Section 1.2. To measure time performance, for instance, we would, for each test, take the time of the new algorithm on that test and divide it by the time of the canonical algorithm on that test. We then average the 34 resulting ratios to get the relative performance of the new algorithm. On occassion, we also report the minimum and maximum ratio.

When measuring time, we record the user time spent by each process. User time is an imperfect measure of hash performance. It does not include the time spent paging, for instance, which is often the major time bottleneck for large dictionaries. On the other hand, user time does include such activities as making a copy of each hash command, converting strings to integers, and so forth — I/O-related operations which should not be counted as a cost of hashing. To account for this I/O time, we ran each test on a "dummy" hash routine, which parsed the commands in the same manner as all the other algorithms, but immediately returned FALSE on all inserts, lookups, and deletes. We subtracted the time taken by the dummy hash routine from all the times calculated. We subtracted this time even from the DIMACS demo hash and Perl code, even though they use different parsing routines, because we have no way of accurately measuring the time spent parsing for these implementations. The parsing code used by the dummy hash is not particularly efficient, so we believe the normalization is conservative, making the demo and Perl hashing routines perhaps seem faster than they actually are.

As a proxy for paging cost, we report the space used for the various algorithms. The less space an algorithm uses, the larger the dictionary problem that can be efficiently handled. When evaluating space use, we measure the maximum amount of memory allocated during any part of the algorithm. Because resizing the hashtable requires two copies of the hashtable to be in use simultaneously, the high-water memory need will typically be higher than that of the final hashtable, even if no hash table items are ever deleted.

3 Improving the FKS algorithm

There are several reasons the algorithm described in Section 1.2 is impractical. One is the space use, which is linear in the number of items but has large constants. For instance, we must store a value of k_i with each bottom-level bin, and each k_i requires as much space to store as the longest key. While space/time trade-offs are possible — the user can modify c and c', the overestimation parameters used in the dynamic FKS algorithm — they cannot improve the algorithm enough to make it space efficient.

Time inefficiencies are even more problematic. Computing $h_k(x)$ requires time quadratic in the number of bits in x. If x is actually a string, this means that hash computations are quadratic in the length of the string. Furthermore, every insert and lookup requires computing two hash values. In addition, while the number of times a low-level bin is grown may be linear in the number of operations, it is certainly mcuh larger than the number required by, say, hashing with linear probing, and every bin-growth operation requires a slow memory-allocation step. Finally, the two-level scheme requires accessing more parts of memory than does hashing with linear probing. In modern machines, the cost of random (as opposed to sequential) memory access can be a significant part of the cost of a hashing algorithm.

In the remainder of this section, we propose a series of modifications to the FKS algorithm that improve its time and space profiles. Each modification is added to the previous ones, so the final algorithm includes all the proposed modifications. The major benefits come from two of the optimizations: (1) storing some data directly in the top level bin, thus dramatically decreasing the number of hashes that need be performed, and (2) modifying the algorithm in such a way that almost all k values can be 1, significantly speeding up the hash function.

3.1 Storing Items in the Top-Level Bin

One easy optimization, noted by the original authors of the algorithm in [8], is to store singleton items of low-level bins in the top level. That is, instead of having, in the top-level bucket, a pointer to a bin with only one item in it, store the item itself in lieu of the pointer. This saves space because there is no need to allocate space for a low-level bin. In addition, since the value of k_i is already stored in the top-level bin, along with the pointer, there is no need to make the top-level buckets larger to accomodate holding an item instead of merely a pointer. Finally, lookup on singleton items requires only one hash, instead of two.

As can be seen in Section 3.7, this simple modification greatly improves the performance of the FKS algorithm.

3.2 Making Small Low-level Bins Smaller

In modifying FKS for the dynamic case, Dietzfelbinger et al. required that an extra factor of $(1+c')$ space be alloted when allocating low-level bins. In this way, when a third item is inserted there is no need to regrow the table.

However, if the hash values are evenly distributed, very few low-level bins will ever see three items. Therefore, we can save space by not allocating the extra space when inserting a second item into a low-level bin. (Because of the previous optimization, low-level bins are never even allocated until the second item is inserted into the same top-level buckets.) Only when three items find their way into the same bin, indicating that perhaps the distribution of hash values is not uniform, is the extra space allocated.

This optimization obviously saves space, but by doing so it also saves time. This is because the smaller each low-level bin, the more bins can fit on a single page of memory. This makes caching more effective. It also decreases the number of buckets that must be examined for recopying when growing or shrinking the hashtable.

3.3 Making All Low-level Bins Smaller

An extension of the above modification is to never allocate the extra space on a low-level bin, under the theory that if adding a third item in a bin is rare, adding a fourth must be even rarer, so there is even less cause to allocate extra space for the eventuality.

Note that with this modification, the algorithm no longer has amortized constant-time bounds since if n items get put in one low-level bin, we will grow the bin n times. (The previous scheme, which refrained from resizing only a constant number of times — namely once — remains theoretically optimal.) The modification does not noticeably affect either the running time, but we include it in our modified algorithm since, according to our analysis, it improves the space usage.

3.4 Preferring $k = 1$

With the modification of Section 3.1, this is the single most effective modification to the FKS algorithm: When choosing a value for k, at both the top and low levels, try $k = 1$ before picking k at random.

Having $k = 1$ provides several advantages. The most obvious is that it speeds up the algorithm: calculating kx, usually the most expensive operation in the hash function, is particularly straighforward when $k = 1$. It also saves space; normally we require many bits to store k, particularly if the keys can be large. By storing a short code for the common $k = 1$ case, however, we can reduce the memory requirement needed to store k values.[3]

Note that having $k = 1$ is a particular benefit at the top level, since every hash operation requires a hash at the top level, but only those items stored in low-level bins require a second hash. However, we have found that if $k = 1$ at the top level, it is advantageous to have the top bin size not be a power of 2. This is because if both the top bin and low bins are powers of 2, as might often happen, items that hash to the same bucket in the top level will hash to the same bucket in the low level as well, unless $k \neq 1$ for one of the bins (since both levels are looking at the same low-order bits of the item keys). The most time-efficient way of avoiding this problem is to have the top bin be slightly smaller than a power of 2.

The theoretical effect of trying $k = 1$ before trying a random k is to increase the average number of k that may need to be examined in each resizing step. The constant time bounds, however, remain unchanged.

3.5 Resizing When $k \neq 1$

While having $k = 1$ is advantageous, it is not always possible. On the low level, $k = 1$ may cause a collision, while on the top level it may result in an uneven distribution of items into low-level bins. One solution to this problem is to modify the hash function h, not by changing k, but by changing t, the size of the hash table. In practice, we grow t so the theoretical properties governing the number of collisions continue to hold. The new hash function, with the larger t, hopefully will be collision-free for $k = 1$. If necessary, several values of t can be tried. In our implementation, we try 10 different values of t before settling for a non-unit k.

This modification is not cost-free. Besides the cost of a few extra words of memory, changing the size of the hash table requires a new memory allocation. It also makes the hash table slower, since t is no longer a power of 2 and hence the final mod cannot be implemented as a bitmask. However, the cost of these operations is dwarfed by the cost of hashing with a non-unit k. As we see from the figure in Section 3.7, this modification improves the performance of the algorithm.

In the experiments described in Section 3.7, the version of the algorithm incorporating all the above modifications needed to allocate very few non-unit k values, typically zero or one per test set. As a result, we get an additional space savings in our implementation. We store k values as pointers to memory locations, with the NULL pointer indicating $k = 1$. With few non-zero k values, it suffices to use a single byte, rather than a full word, to store the pointer. This saves 3 bytes per bucket on a 32 bit machine.

3.6 Using Non-Prime p

An expensive operation in computing the hash function h is taking kx mod a prime p. While this operation is crucial for the theoretical correctness of the time bound, it is onerous. On the other

[3]In practice, we store each k as a pointer to memory that's allocated on demand. Thus, we have one word of overhead for each k. However, we use the NULL pointer to indicate $k = 1$, saving memory in that case.

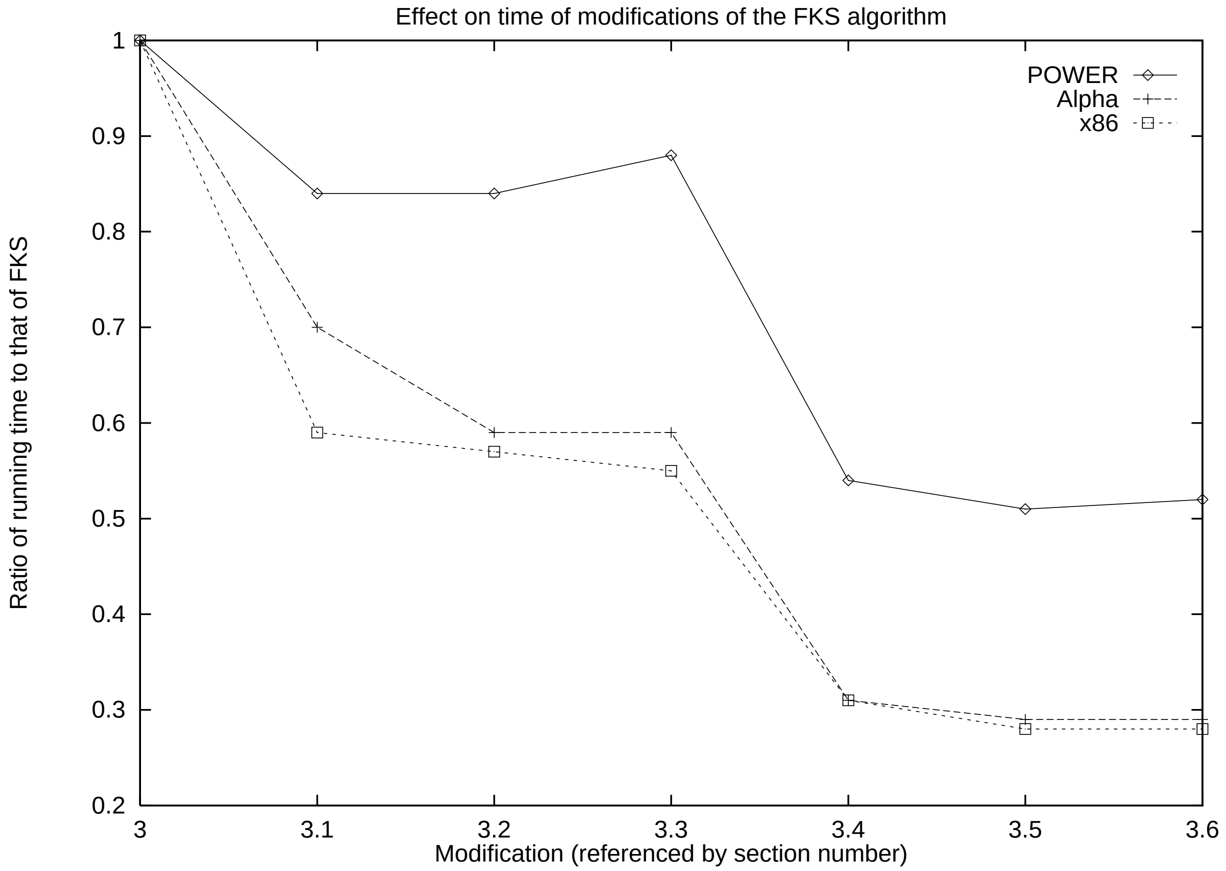

Figure 1: The improvement in running time as the FKS algorithm is modified, for three different computer architectures. The numbers along the x-axis refer to the section of the paper in which the modification is proposed.

hand, calcuating $kx \bmod 2^i + 1$, for some i, can be done quite efficiently (see [10, p. 272]). Note that if $k = 1$, then the value of p does not matter, since $kx < p$ in this case. Since, as we've noted, $k = 1$ almost always in practice, setting $p = 2^i + 1$ will probably not destroy the correctness of the algorithm — though in theory there could be two keys that collide on every value of k for a swath of t values. On the other hand, making p non-prime probably will not improve the algorithm very much either, even in the best case. As we see in Section 3.7, this modification has little effect on the running time.

One reason being able to use non-prime p is important is because finding a large prime, while a one-time cost, is expensive and can dominate the time of FKS hashing. Assuming the range of keys is always a power of 2, it is possible to pre-compute a table listing the smallest prime larger than every power of 2, but this limits the maximum key size that may be used. One compromise is to use a prime p if there is an appropriate table entry and to use $2^i + 1$ otherwise.

The version of the algorithm with all of the above modifications included is called the *mFKS* algorithm, for "modified FKS."

3.7 Experimental Results

Figure 1 shows the reduction in average time per test set as each modification is added to the basic FKS algorithm. The modifications are labeled on the x-axis by the section in which they are

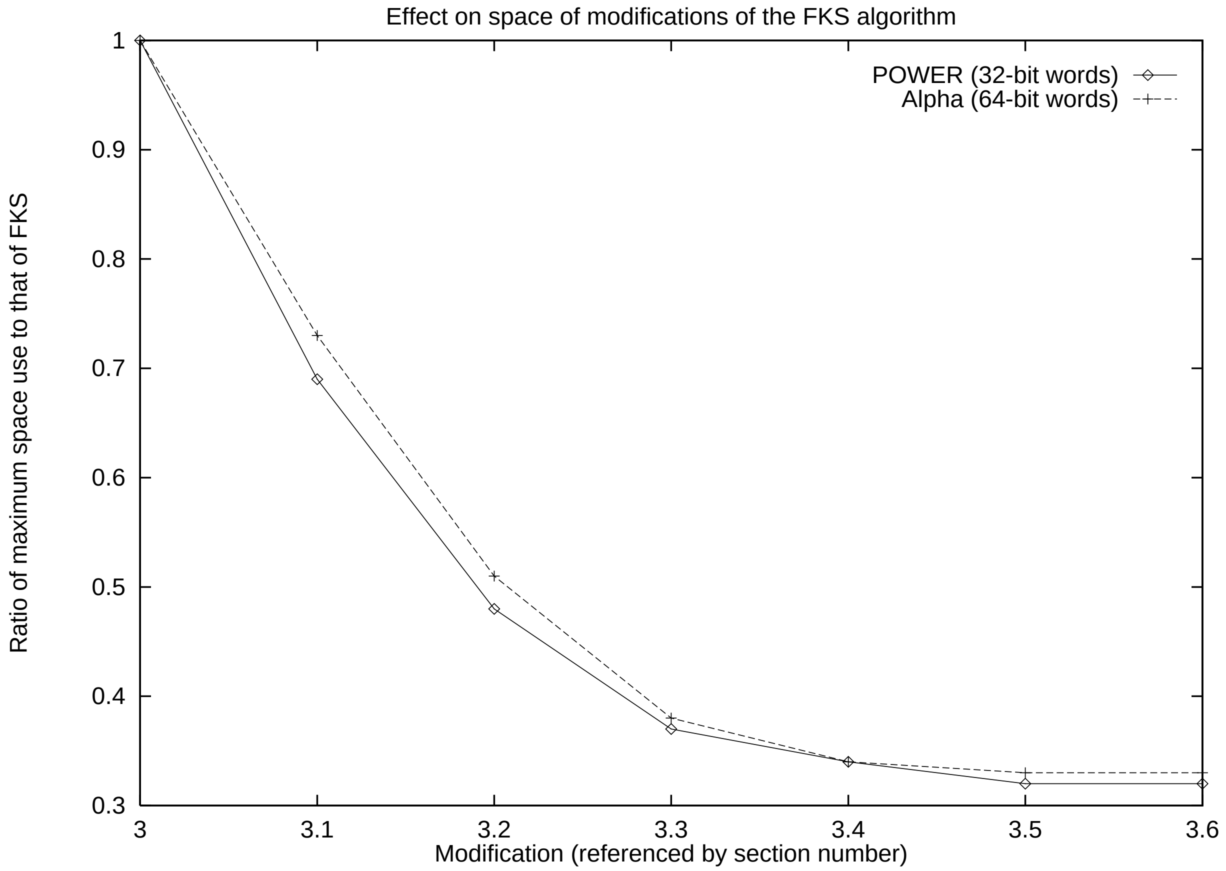

Figure 2: The improvement in space use as the FKS algorithm is modified, for three different computer architectures. The numbers along the x-axis refer to the section of the paper in which the modification is proposed.

discussed. The times are shown relative to the unmodified FKS algorithm — which therefore has a time ratio of 1. We show results for all three architectures we examined.

The fully modified FKS scheme, mFKS, is almost 4 times faster than the unmodified scheme on the x86 and Alpha processors. It is about twice as fast on the POWER. The data support the hypothesis that the POWER machine is relatively faster at memory access, and relatively slower at bit-level operations such as multiplying large words: the improvement in time for the POWER machine is relatively small when adding top-level buckets (3.1), but relatively large when adding preference for $k = 1$ (3.4).

In Figure 2 we show the improvement in space use as the FKS algorithm is modified. We show results both for 32- and 64- bit machines. The space performance on both architectures is the same.

It is worth noting that making all low-level bins static (3.3) yields a substantial improvement in space use, though, as Figure 1 indicates, it made the algorithm slower on many architectures. Deciding whether to incorporate this modification therefore involves making a time/space trade-off, one of several available with the FKS algorithm (see Section 5).

4 The Table Data Structure

The primary data structure used for hash tables is the table. The obvious representation of a table is an array. Access is straightforward and takes constant time. On the other hand, an array is profligate with memory use if many of the cells of the table are empty.[4]

There are several data structures that can represent empty table cells more efficiently than the array. These variants, typically called *sparse arrays*, only store non-empty cells. In addition, they store a small number of bits indicating which cells are empty and which are full. It is possible, using sparse arrays, to look up a cell's contents in constant time, though the constants are larger than they are using a *dense* (i.e., not sparse) array. Thus, choosing whether to use a sparse or a dense array involves a time/space trade-off.

Note: The time/space trade-off in using dense or sparse arrays for hash tables is not as clear-cut as the previous discussion indicates. While dense arrays are faster than sparse arrays for lookup and (particularly) for insert, they are actually slower in performing another common hash table operations: traversing a hash table (that is, examining each item of the hash table in turn). This is because a dense array must skip over empty cells. A sparse array, which only allocates memory for non-empty cells, does not suffer from this time sink. In addition, sparse arrays have better spatial locality than dense arrays due to their smaller size, yielding time advantages over dense arrays with respect the memory subsystem.

In the following sections we describe two implementations of sparse arrays, one proposed in [8] and one a modified version that uses less memory. We then compare the performance of the mFKS algorithm under the different table implementations.

4.1 Sparse Arrays Using Offsets

This method of efficiently reprsenting a table is from [8]. The idea is to consider the table of size t as consisting of t/M blocks of M cells each, for some value M. Cell x can be found in cell $x \bmod M$ of block $\lfloor x/M \rfloor$.

Memory is saved by storing each block sparsely: Only occupied buckets in the block are allocated. Occupied buckets are stored in a dense array, which we call the *store*. When a previously unoccupied bucket becomes occupied, the store is grown and the item is stored in the empty cell now available at the end of the store. An array A is used to map from the position in the sparse array, i.e. $x \bmod M$, to the position in the store. $A[i] = j$ if cell i is the j^{th} item in the store. If cell i is empty, $A[i] = -1$. When a cell is deleted, the appropriate entry in A is set to -1, but the space in the store is not reclaimed. Periodically recopying the sparse array can keep memory use from growing excessively with many deletes. This recopying can be amortized over the cost of the deletes, so maintaining the sparse array takes constant time per operation.

This technique saves space because we require only $t \log M$ bits to store all the A's, not $t \log t$. In [8], the authors recommend setting $\log M$ to $\sqrt{\log n}$, meaning all the arrays A, as well as the top and low level hash tables, can be stored in $O(n)$ space. In practice, we find setting M to a convenient power of 2, such as 256, provides good space savings while being easy to implement. Only for $n > 2^{64}$ will $\log 256 = 8$ be smaller than $\sqrt{\log n}$.

[4] In the context of hash tables, we have been referring to "bins" and "buckets." These terms are equivalent to "tables" and "cells," respectively. We use new terminology here to distinguish discussion that applies to all tables from that applying only to hash tables.

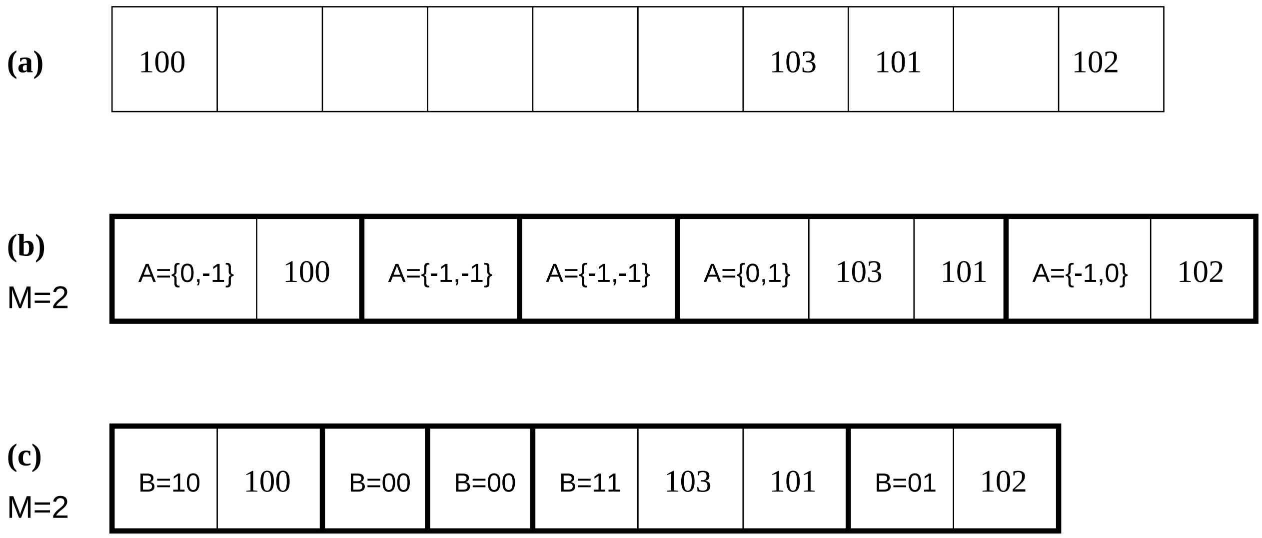

Figure 3: **(a)** The dense array scheme. **(b)** The sparse array scheme using offsets. The thicker lines separate groups. In practice, M would be much larger than 2. **(c)** The sparse array scheme using bitmaps.

4.2 Sparse Arrays Using Bitmaps

The technique described above can be modified to yield even more space savings by using a bitmap B instead of an offset array A. B consists of M bits; bit i is 1 if and only if cell i of the group is occupied. If there are b bits set in $B[0,\ldots,i-1]$, then item i is the b^{th} item in the store.

Unlike in the offset sparse array, in which newly inserted items are appended to the end of the store, in the bitmap sparse array inserted items must stay in place relative to other non-empty cells. If the calculations place the item as the b^{th} item in the store, all items currently in position $b+1$ and larger must be moved one space down in the store. Likewise, when an item is deleted all items after it in the store must be moved one space up. Both move steps can be done efficiently if the store is implemented as a linked list, but such an implementation defeats the purpose of saving memory. Instead, the store is recopied. If M is a constant, the recopying is a constant-time operation.

Besides copying the store, an event that is only necessary on insert and delete, the most expensive operation for sparse arrays is counting the number of bits set in $B[0,\ldots,i-1]$. This can be done reasonably efficiently — in particular, in time proportional to the number of bits set — in the RAM model. By treating B as a computer word, and using tables, it is possible to calculate the number of bits set even more efficiently. In any case, even the naive implementation takes constant time if M is constant.

Figure 3 illustrates the three implementations of the table data structure.

4.3 Experimental Comparison

The FKS data structure includes more than one table. It is possible to use one representation of a table for the top level bin and another representation for the low level bins. For instance, since the top level bin is accessed on every query, it might make sense to use the fast dense array for the top level. Each low level bin, on the other hand, might be represented using a sparse array. If many items are held on the top level, this implementation gives a space savings over the canonical, dense-only implementation without unduly affecting running time.

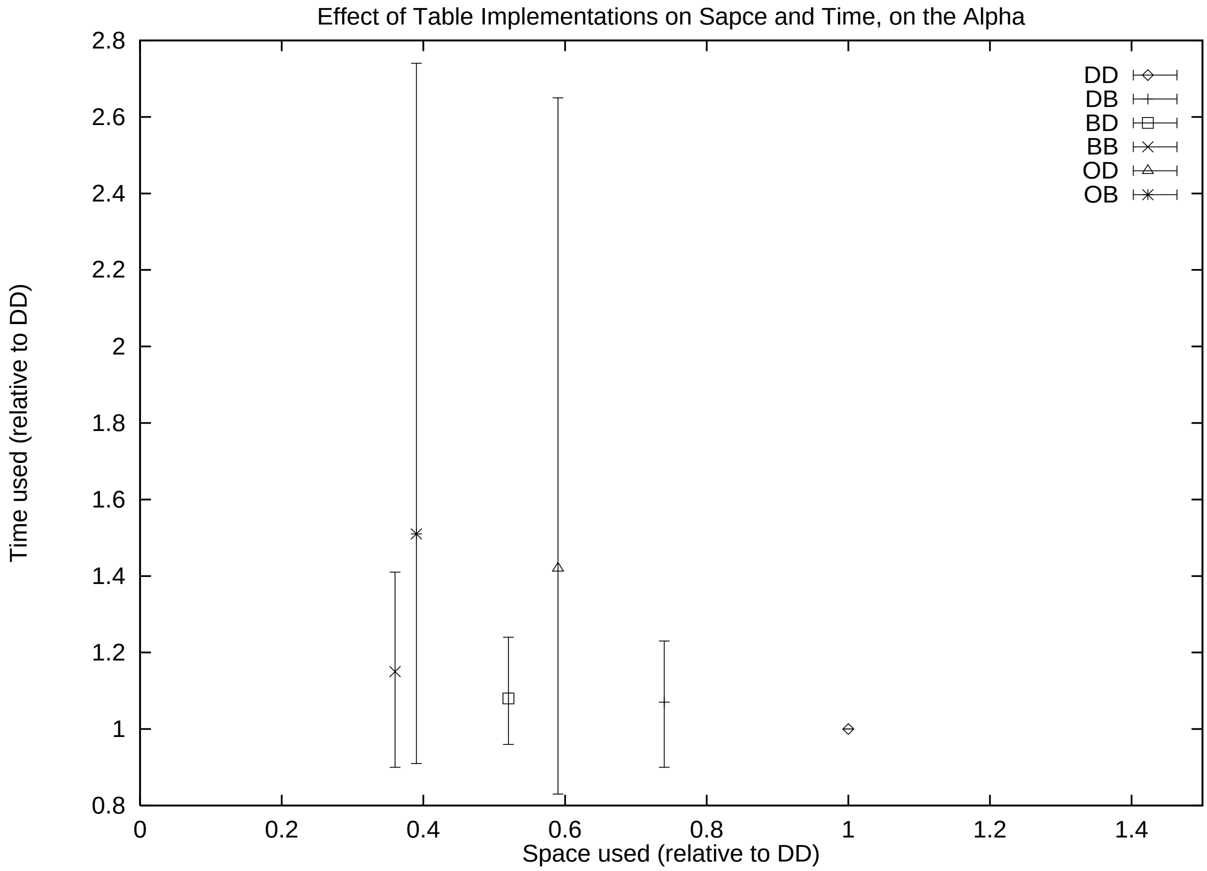

Figure 4: A scatterplot comparing space and time use for various table implementations on the Alpha. `D` = dense arrays, `B` = sparse arrays with bitmaps, and `O` = sparse arrays with offsets. The first letter of the implementation name is for the top-level bin; the second is for the low-level bins. The vertical lines indicate the minimum and maximum time ratio among the 34 test sets.

We explored six different implementations of the FKS data structure:

`DD` dense array at top level, dense array at low level,

`DB` dense array at top level, bitmap sparse array at low level,

`BD` bitmap sparse array at top level, dense array at low level,

`BB` bitmap sparse array at both levels,

`OD` offset sparse array at top level, dense array at low level, and

`OB` offsedt sparse array at top level, bitmap sparse array at low level.

The `DD` implementation is the canonical one; we represent every other implementation by the ratio of the running time, or space used, of the implementation to `DD`. We plot each implementation in a scatterplot, with time ratio along one axis and space ratio along the other. In addition to plotting the average time ratio, as described in Section 2, we use vertical lines to indicate the minimum and maximum time ratio among the 34 tests. The length of the vertical lines is an indication of how consistently the choice of table implementation affects running time.

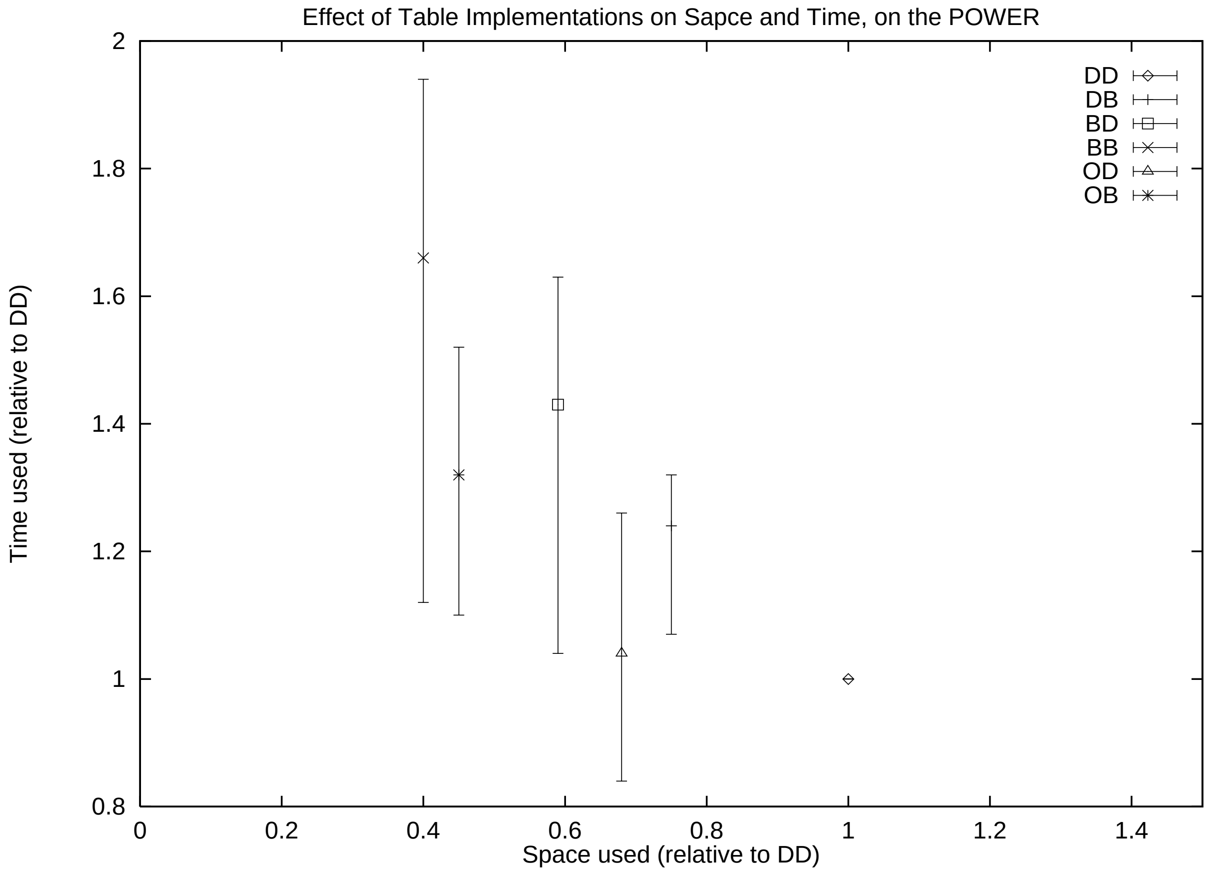

Figure 5: A scatterplot comparing space and time use for various table implementations on the POWER. `D` = dense arrays, `B` = sparse arrays with bitmaps, and `O` = sparse arrays with offsets. The first letter of the implementation name is for the top-level bin; the second is for the low-level bins. The vertical lines indicate the minimum and maximum time ratio among the 34 test sets. Unlike on the Alpha and x86, offset arrays are faster than bitmap arrays on the POWER.

In Figure 4 we show results for the Alpha architecture, while Figure 5 shows results for the POWER. The performance on the x86 was similar to that of the Alpha. We see that the bitmap schemes performed much worse on the POWER than the other architectures, and also the range of time ratios (among the 34 DIMACS test sets) was higher on the POWER. This is not surprising, since the bitmap implementations depend on efficient bit manipulation for good performance, and the POWER processor was already seen to be relatively weak in this area.

While the sparse arrays generally lead to longer running times, on several test sets, particularly on the Alpha, the sparse array variants were faster than the canonical mFKS algorithm. This perhaps indicates these test sets benefitted from the improved memory locality a sparse implementation provides.

5 Parameters for Trading Off Time and Space

There are several parameters that can be tweaked to obtained a time/space trade-off for FKS hashing. One is available only for the bitmap sparse array implementations: modifying M, the size of the array groups. (Modifying M is theoretically possible in the offset array case as well, but in practice M is efficient only if stored in a series of bytes. Thus, the next-highest value above $M = 256$ is the impractical 65,536.) Larger values of M decrease the size by reducing the number of groups, and therefore the per-group overhead. At the same time, they make lookup slower by increasing the size of each bitmap array.

In Section 1.2, several constants are introduced that can be modified. Each of them, coincidentally, has a recommended value of 2 that is used in the canonical version of FKS.

$\mathbf{1 + c}$ — top-level size multiplier for dynamic hash tables
 Effect of increase on **space**: top level hash table gets larger
 Effect of increase on **time**: fewer rehashes needed
 Recommended value [7]: 2

$\mathbf{1 + c'}$ — low-level size multiplier for dynamic hash tables
 Effect of increase on **space**: each low-level hash table gets larger
 Effect of increase on **time**: fewer rehashes needed
 Recommended value [7]: 2

c in $\mathbf{4cn^2/t + n}$ — maximum allowable sum of squares of top level collisions
 Effect of increase on **space**: increases the low-level hash tables' cumulative size
 Effect of increase on **time**: decreases the expected number of k's examined
 Recommended value [8]: 2

c in $\mathbf{c|C_k(i)|^2}$ — size of low-level hash tables
 Effect of increase on **space**: each low-level hash table is larger
 Effect of increase on **time**: decreases the expected number of k_i's examined
 Recommended value [8]: 2

As a group, these constants affect the *load factor*, that is, the percent of hash table buckets are occupied. A higher load factor on the top level tends to cause larger low-level bins. A higher load factor on the bottom level tends to complicate the search for a perfect hash function.

There is one constant that can be solved optimally: the size t of the top hash table versus the (cumulative) size of the low level hash tables. While theory requires that $t \in \Omega(n)$, we can pick the

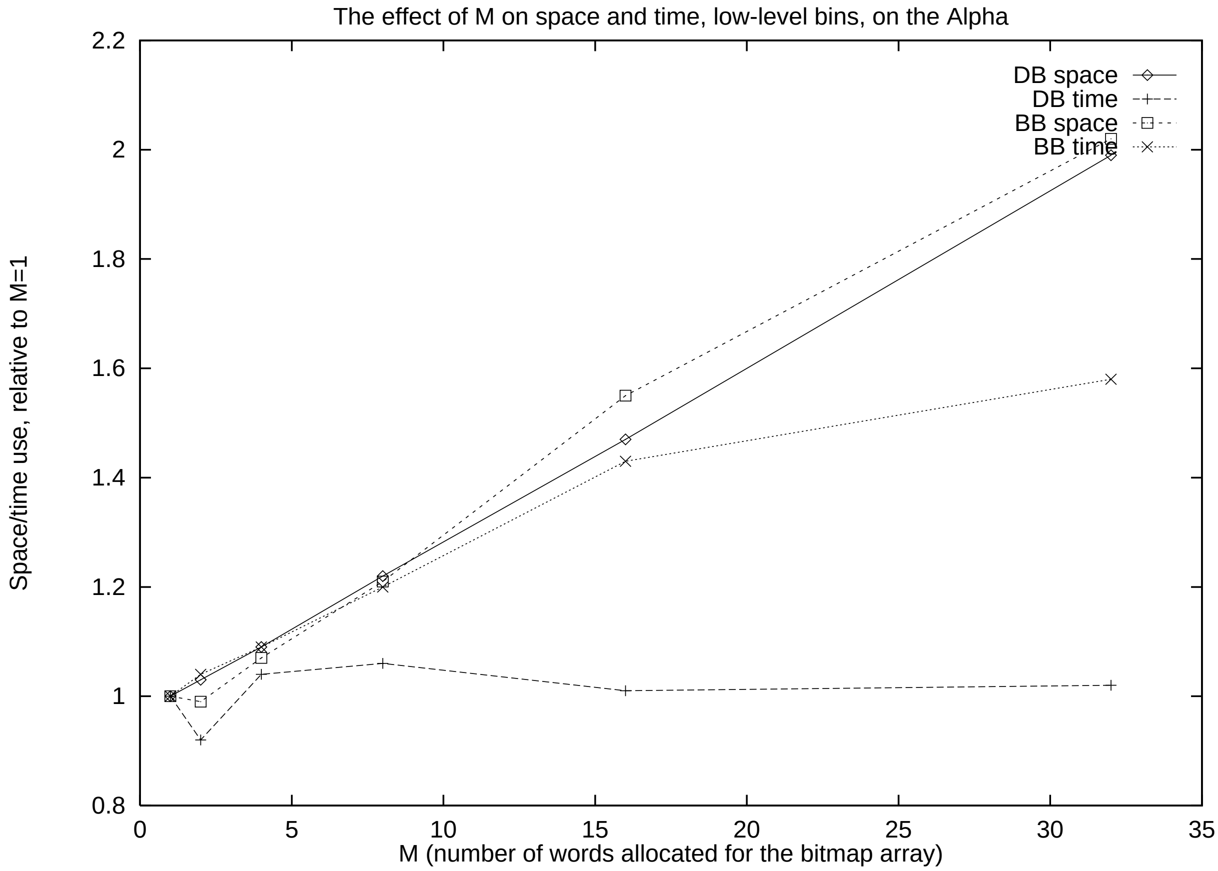

Figure 6: The effect of increasing M on space and time use when the low-level bins are sparse arrays, on the Alpha. The space use increases for large M because the number of buckets in a low-level bin is small, so increasing M increases the overhead without improving the addressing capability of the bitmap scheme.

constant. If we assume that the keys are distributed randomly in the hash table,[5] we can predict the number of collisions, and thus the size of the hash table, before inserting any elements. Under this model, the total hash table size is a second-order function of t, and we can use bisection or other methods to find the value of t that minimizes expected space use. Note that the value thus obtained is merely an estimate, since keys may not in fact be distributed randomly, and it will work better on some data sets than others.

In the following sections we experiment with modifying the values of M and of the constants affecting the load factor.

5.1 Modifying M

We experimented with letting M range from 1 to 32 words. (In all previous experiments, M had been set to 1 word.) Most of the results we show are for the Alpha, where one word has 64 bits. On all other architectures, one word has 32 bits.

In Figure 6 we explore the effect of increasing M on space and time use for those implementations where the *low-level* bins are sparse. Results are only shown for the Alpha, but the effect on other

[5] This is, of course, not the same as saying they are distributed randomly throughout the possible key space.

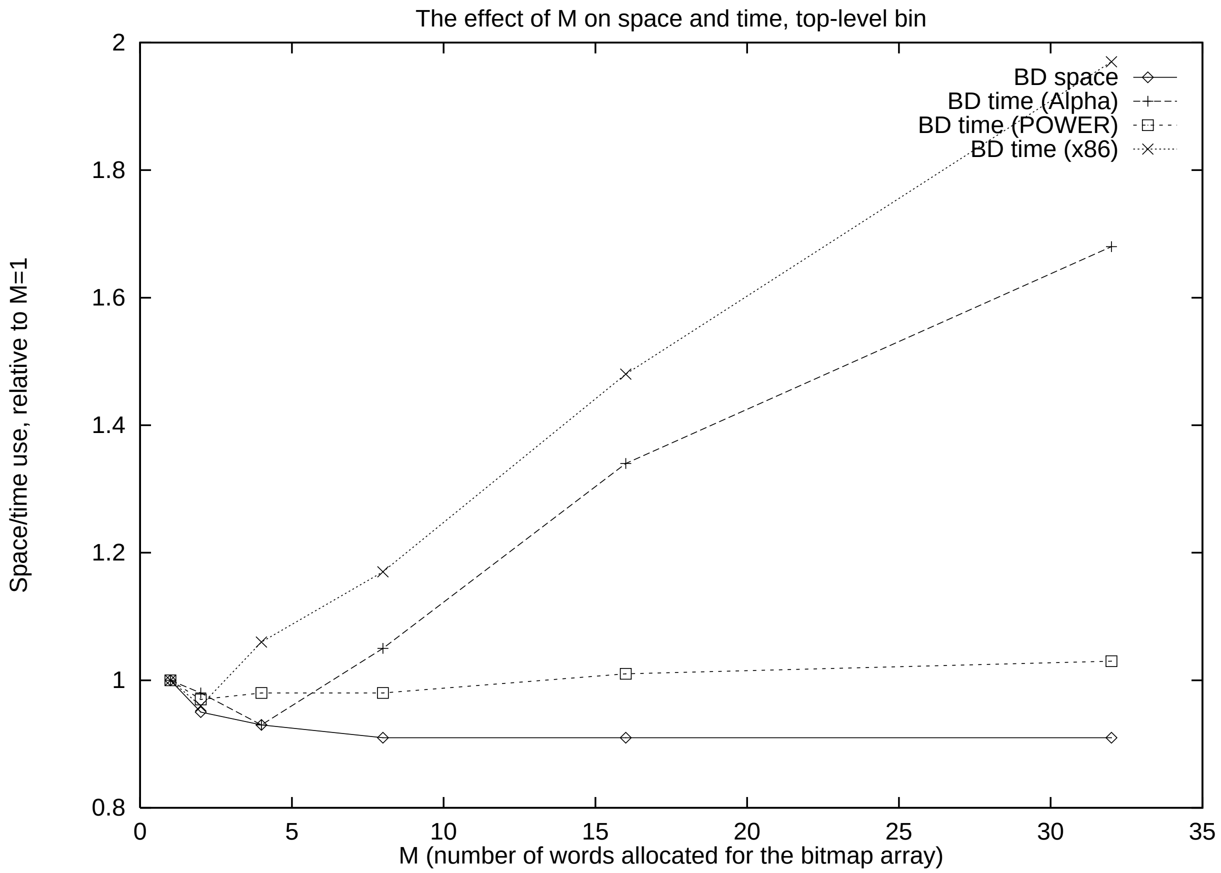

Figure 7: The effect of increasing M on space and time use when the top-level bins is a sparse array. The space improves only marginally — we only show data for the Alpha, but that for the other architectures is similar — while the running time increases significantly for large M.

machines is similar. Perhaps surprisingly, the space use increases with M, rather than decreasing. This is because low-level bins tend to be small; large M provides a space savings by requiring fewer array groups, but for low-level bins all the buckets fit into one group even for small M. Since there are no time savings from using larger M in this case, it makes sense to use $M = 1$ or 2 for low-level bins.

Changing M for the top-level bin shows the expected time/space trade-off, as Figure 7 shows. We only show show memory use for the Alpha, but the pattern of memory use on the other, 32-bit machines is the same. The pattern of time use, however, differs remarkably among machines. For all the machines, the running time grows approximately linearly with M as M gets large. However, for small M, the time actually improves with increasing M. This seems to be due to the improved memory locality of larger M. In the mFKS implementation, each group of M buckets is stored in a separate location in memory, though all the buckets in a group are stored contiguously. Since only buckets in the same group are guaranteed to be on the same page of memory, as M increases the memory locality of the buckets is likely to increase as well, decreasing the number of cache misses. Fewer cache misses results in a time decrease since, unlike time spent handling page misses, time spent handling cache misses is attributed to the application.

Another potentially surprising observation from Figure 7 is that the running time increases very slowly on the POWER as M increases. Given our theory that the POWER processor is relatively slow with bit operations, we might expect the opposite: that as the bitmaps get larger, the performance of the POWER machine would degrade. We have no explanation for why the running time instead stays relatively constant.

In general, as M grows the running time increases much faster than space use decreases. There does not seem to be much reason to ever use $M > 8$. $M = 4$ words seems optimal on the Alpha, while $M = 2$ words is fastest on the POWER and x86 architectures.

5.2 Modifying the Load Factor Constants

For all four of the load factor parameters in Section 5, the values suggested in the original text is 2. We experimented with letting these values range, in concert, from 1.1 to 6.

As Figure 8 shows, the load factor has a consistent effect on running time regardless of machine architecture and table implementation. However, as we see in Figure 9, the space performance depends heavily on the table implementation. The space use does not depend very much on architecture, so we concentrate only on the Alpha architecture.

One observation is that space use actually increases for small load factor parameters when the top-level bin is sparse. This is because small load factor parameters make the top-level bin more full, reducing the number of empty top-level buckets while increasing the number of low-level buckets. For a sparse array, however, the cost of empty top-level buckets is much lower than the added cost of the low-level buckets. Combining a sparse top-level bin with a dense low-level bin provides the worse performance of all: not only is space use inefficient for small values of the constants, it is inefficient for large values of the constants as well, since in this case the (dense) low-level bins, though they have fewer items in them because of the low load, are all exceedingly large since c' is high.

5.3 Putting It All Together

In Figure 10 we show a scatterplot of the time/space trade-off for the following parameters: $M = 1$, 2, 4, 8, 16, and 32 words for `DB`, `BD`, and `BB` tables, and load factor parameters = 1.1, 1.5, 2.0, 2.5, 3.0, and 6.0, for `DD`, `DB`, `BD`, and `BB` tables. Space and time are calculated relative to the canonical

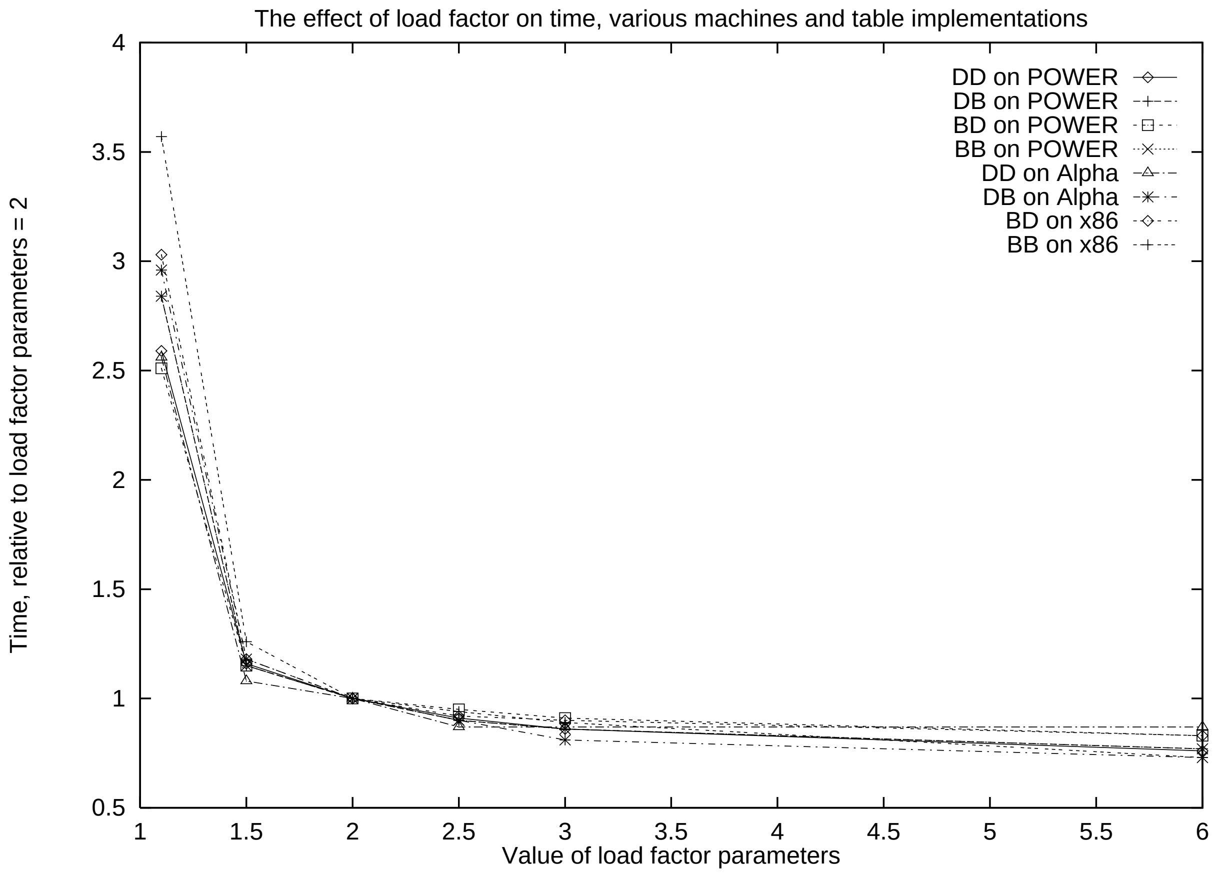

Figure 8: The effect of load factor parameter on running time. To keep clutter to a minimum, only a subset of all the architecture/table implementation pairs is shown. Behavior is similar for those pairs omitted.

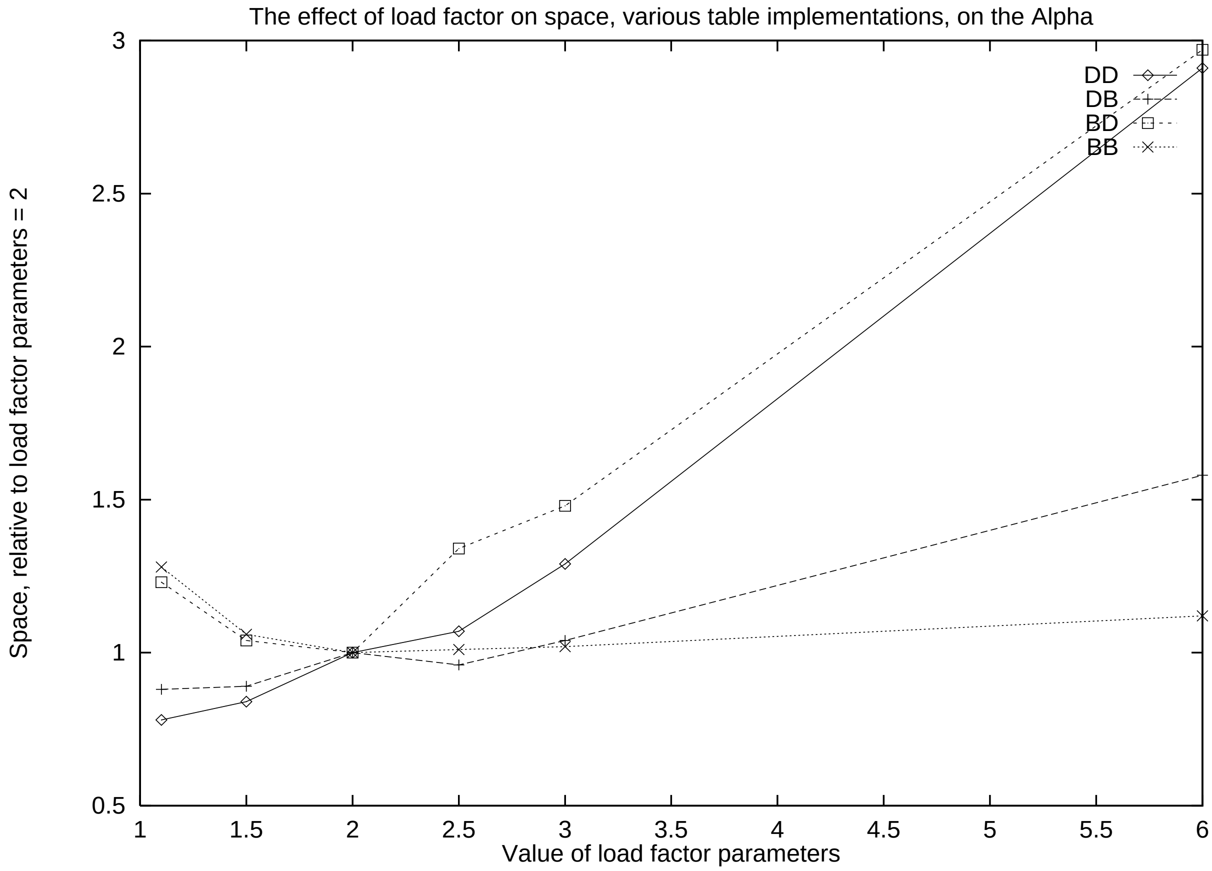

Figure 9: The effect of load factor parameter on space use, for the Alpha. Behavior is similar for other architectures, though these results are not shown.

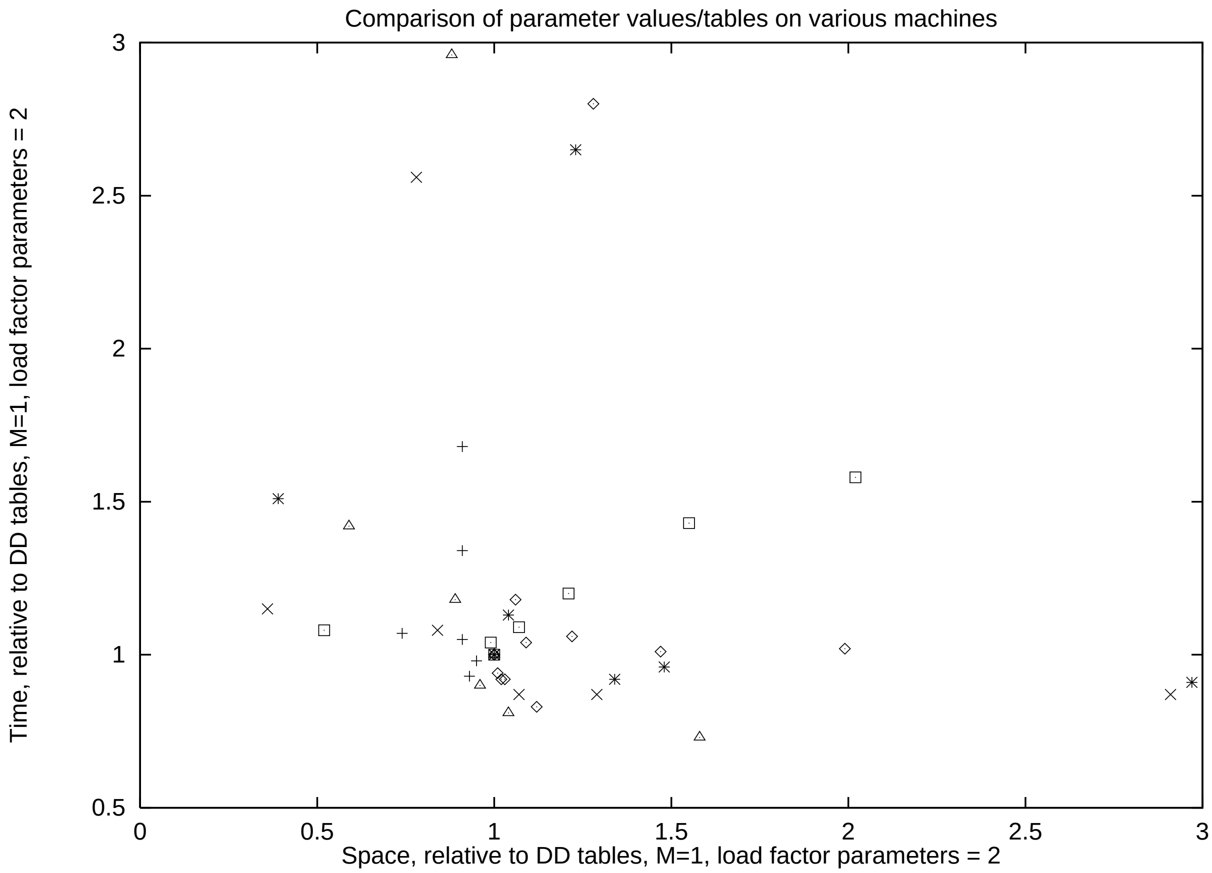

Figure 10: Space use versus running time, on the Alpha, relative to the canonical implementation of the mFKS algorithm (`DD` tables, $M = 1$, and load factor constants $= 2$). The data points represent different values and implementations for these three facets of the mFKS algorithm.

implementation: $M = 1$, all load time parameters at 2, and dense tables at both levels. Results are shown for the Alpha architecture. Results for other architectures are comparable.

While for reasons of space we cannot label each point with the parameter/table pair that produced it, which would allow us to study how individual choices affect the time and space use, the figure provides an indication of the capability to trade off time and space in the mFKS algorithm. The canonical algorithm, with its small M, relatively small values for load factor constants, and dense tables, yields a particularly fast but space-inefficient variant of mFKS. By increasing the value of the constants and using sparse arrays, it is possible to significantly reduce the space use. It is much more difficult, however, to attain a faster implementation of the mFKS algorithm using techniques in this and the previous sections.

6 FKS Hashing vs. Other Hashing Algorithms

While we have shown how to improve the the performance of FKS hashing via modifications, and how to effect a time/space trade-off through parameter tuning, we have not shown that the resulting algorithms are competitive with other hashing schemes in terms of either time or space. It is not necessary that FKS-style hashing be superior in all circumstances, as it might still be useful in situations where the guarantee of constant-time operations is essential. But if the performance is not competitive, the FKS hashing technique will remain an object of only theoretical interest.

We compare mFKS against two other dictionary schemes: an implementation of hashing with linear probing based on discussion in Knuth [9], and the dictionary implemetation used for associative arrays in Perl 5.[6] Linear probing yields a fast and compact hash table implementation. As we demonstrate below, on the DIMACS test set it is strictly superior to the more common hashing with separate chaining, which is used for instance in LEDA. The Perl implementation, while not intended for the large data sets used in the DIMACS Challenge, gives a feel for how the FKS variants compare to an off-the-shelf hashing routine. The Perl implementation compares favorably to other off-the-shelf implementations such as those found in LEDA and the C++ Standard Template Library [6] because it is written in C rather than the significantly slower C++ with templates. Also, unlike the LEDA implementation, the Perl implementation is optimized for performance over readability and rigorous error handling.

The perl code for testing the dictionary data structure is very straightforward, since the dictionary data type is a basic one in perl. We present the entire code in Table 2.

For comparison purposes, on some of the test sets we also ran a demo hashing algorithm, provided by the DIMACS Challenge organizers [1], that uses a binary tree to hold hashing information. Because the demo hash took an unacceptably long time to run on the large `matchexact` data sets, we did not test it on these data sets, leaving only eight data sets for hashing the demo algorithm. Thus, the variance of the time use of this algorithm is higher than that of the other algorithms.

Hashing with linear probing was described in Section 1.1. We use the hash function described in [2] — it includes versions for both 32- and 64-bit architectures —, and for the table implementation we explore all three options described in Section 4. We also explore quadratic probing in addition to linear probing. In quadratic probing, instead of trying bucket $h(x)+i$, as i ranges from 1 to t, until an empty bucket is found, the lookup function tries $h(x)+i(i+1)/2$. If the table size is a power of 2, this quadratic probing scheme examines all t buckets in the first t probes. While, for large i, quadratic probing has poor locality when compared to linear hashing, it avoids problems with item clustering and should therefore prove superior in most situations.

[6] Perl 5.003 was used on the Alpha machine. The other machines used Perl 5.004_04.

```
line: while ( <> )
{
   /^ins (\S*)/ && do { $count++ if $dict{$1}++ == 0;              next line};
   /^lkp (\S*)/ && do { ($lastkey, $lastvalue) = ($1, $dict{$1}); next line};
   /^dlk (\S*)/ && do { $count-- if delete($dict{$1});            next line};
   /^dli /      && do { $count-- if delete($dict{$lastkey});      next line};
   /^kyv /      && do { $lastkey;                                 next line};
   /^inv /      && do { $lastvalue;    next line};
   /^siz /      && do { $count;    next line};
   /^clr /      && do { undef %dict; $count = 0;                  next line};
   # dch and com can be ignored
}
print "$count items in the hashtable at algorithm's end.\n";
```

Table 2: A perl dictionary implementation that accepts DIMACS dictionary commands.

Hashing with linear probing uses tables and thus has dense and sparse array variants. We consider three variants of tables — dense (`D`) sparse with offsets (`O`), and sparse with bitmaps (`B`) — for hashing with linear probing. Results for quadratic probing are virtually identical. We also consider `DD` and `BB` mFKS algorithms, which show the extremes of the table implementations, with $M = 1$ and the canonical values for the load factor parameters. Finally, we consider the Perl and demo implementations.

Figure 11 shows the performance of each algorithm relative to the canonical mFKS algorithm (with `DD` tables), in terms of both time and space, on the Alpha. Figure 12 shows the same algorithms on the POWER, and Figure 13 on the x86.

Since we do not have space measurements for the demo and Perl implementations, we assign a space ratio of 0 to these tests. The times for the Perl implementation did not fit on the graph: 10.56 times as much time as mFKS for Alpha, 4.35 for POWER, and 13.95 for x86.

As the figures show, the hashing with linear probing is the fastest and the smallest on all three architectures. The reasons for this are discussed below. Nevertheless, the mFKS algorithms are competitive in performance. They are within a factor of two in terms of both time and space. The mFKS schemes are also significantly better than the demo hash on all the machines but the POWER, where the demo hash presumably benefits from not having to do any arithmetic to compute a hash function. The Perl implementation is at least five times slower than either the mFKS or probing schemes on all architectures.

Hashing with linear probing was the best performer in terms of both time (with dense arrays) and space (with sparse bitmap arrays) on all architectures. The reason for the space advantage is straightforward: by having only one level of tables instead of two, hashing with probing avoids much of the overhead of the FKS-based algorithms. It is possible to modify load factor parameters, as it is in the FKS case, to trade off running time and space use. See [9] for details.

The time advantage of hashing with linear probing is dependent on the distribution of key hash values. If many hash keys clump together in the table, causing items to be inserted far from their hashed-to buckets, each lookup and insert will have to examine many buckets. While there will be only one hash performed, with poorly distributed keys there will be many probes into the hash table until an empty bucket is found. On the other hand, if the hash function distributes keys well, and at least half the table cells stay empty, the average-case time for lookup and insert will remain

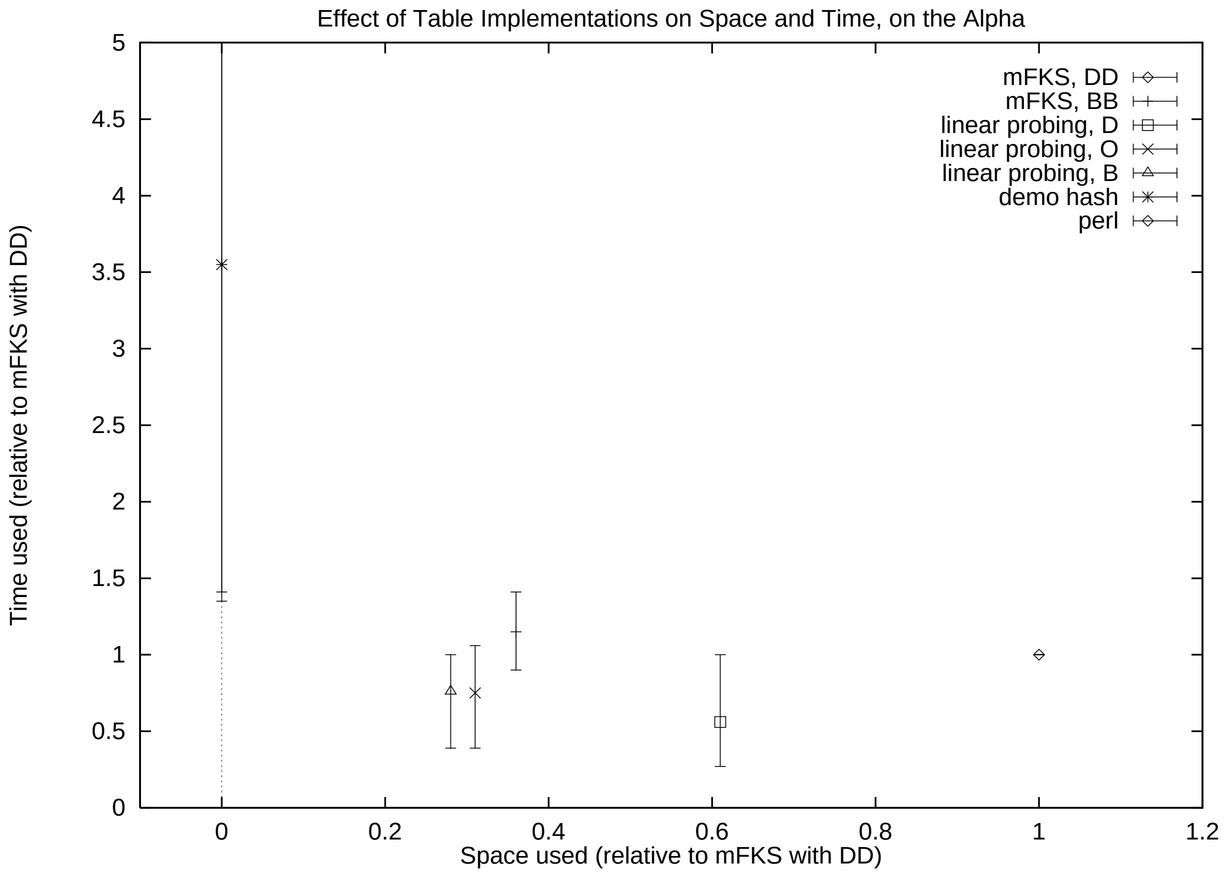

Figure 11: A comparison of various non-mFKS-based hashing schemes against various mFKS-based schemes, on the Alpha. While mFKS is faster and smaller than the demo hashing routine and also than Perl's dictionary code, it is slower and larger than hashing with linear probing.

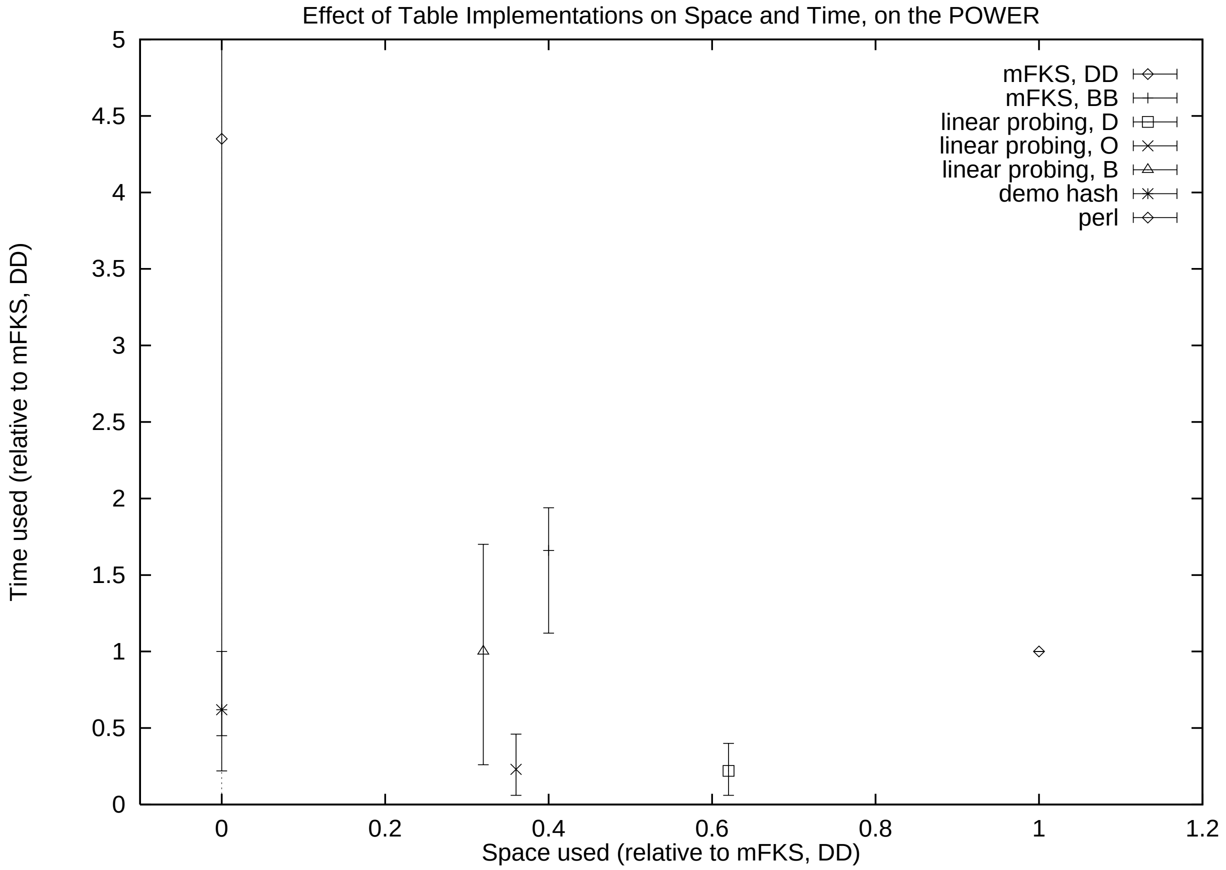

Figure 12: A comparison of various non-mFKS-based hashing schemes against various mFKS-based schemes, on the POWER. As with the Alpha, the probing schemes are the fastest and smallest, while the Perl implementation is the slowest. The demo implementation does well, presumably because it involves no hashing, which the POWER implements relatively slowly.

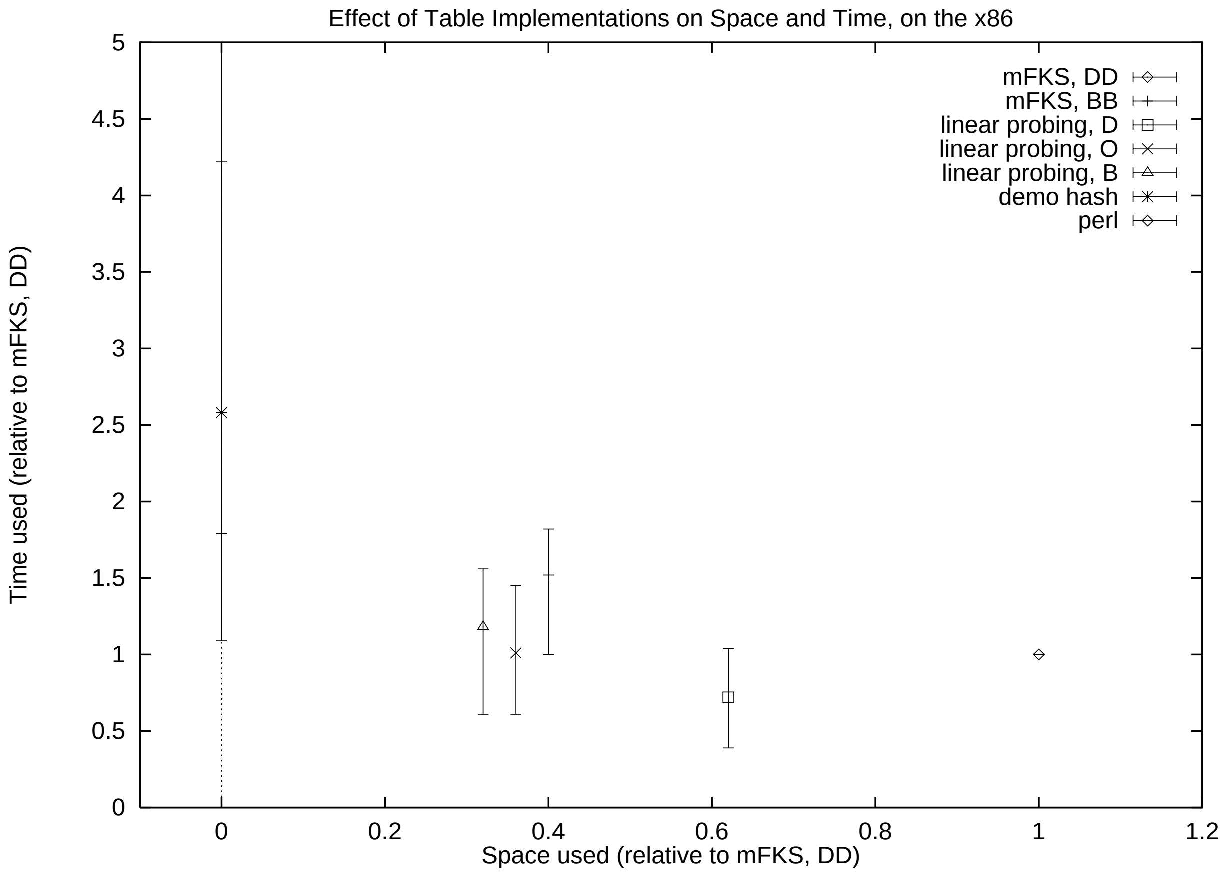

Figure 13: A comparison of various non-mFKS-based hashing schemes against various mFKS-based schemes, on the x86. Again the probing schemes are the smallest and fastest, but the mFKS schemes are competitive.

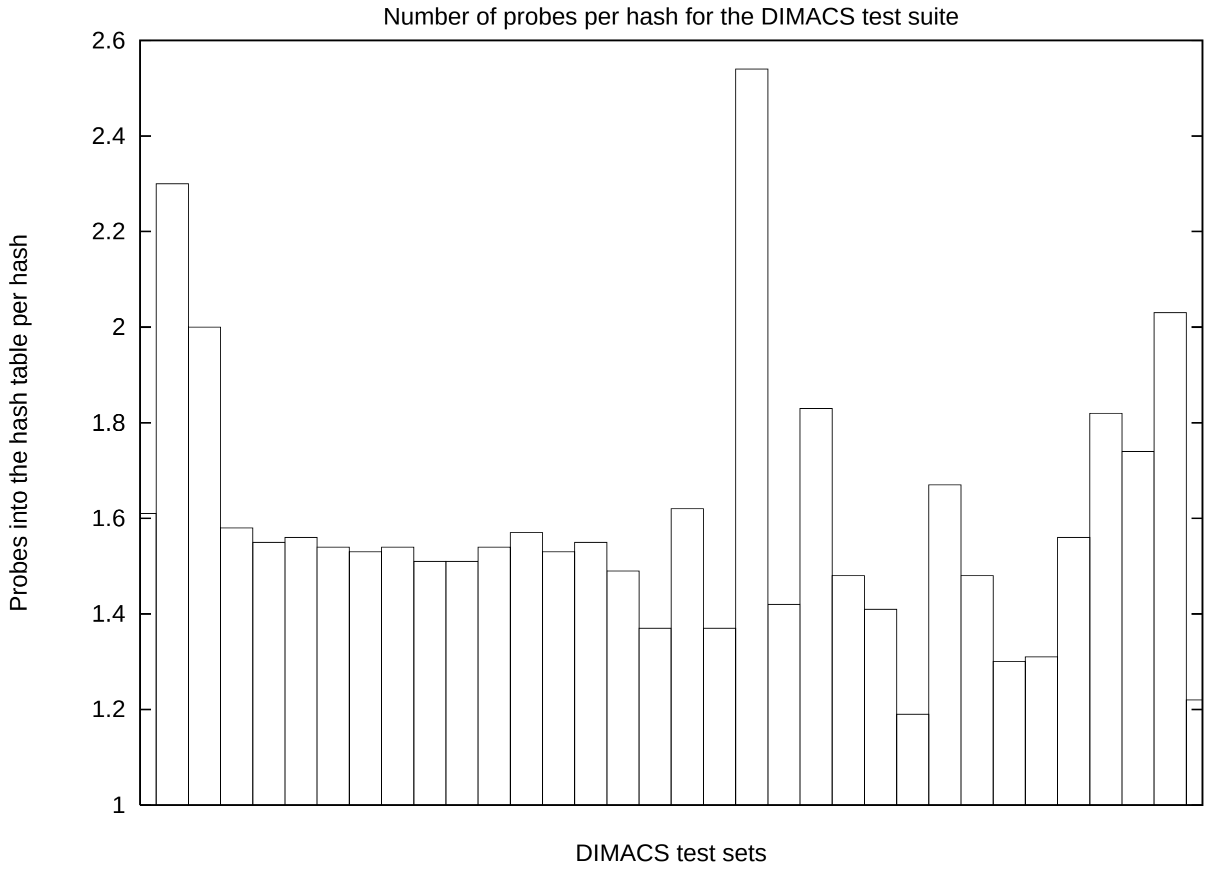

Figure 14: The number of probes per hash for hashing with linear probing, on the 34 test sets in the DIMACS test suite. The closer this value is to 1, the faster are lookup, insertion, and deletion.

constant, with only a few probes per hash. Furthermore, since linear probing is a simple technique, the constants will be small.

For the DIMACS data set, using the hash function from [2], the distribution of keys does in fact turn out to be almost ideal. As we see in Figure 14, the number of probes per hash for the 34 test sets in the DIMACS suite is very close to the optimal value of 1 and only rarely tops 2. The values for hashing with quadratic probing are similar, improving on the linear probing values by an average, over the 34 test sets, of 0.06 probes per hash.

Because the number of probes per hash is low, hashing with linear probing is superior to hashing with separate chaining on this data set. Hashing with separate chaining uses a linked list of items in each bucket; all items that hash to bucket i are appended to bucket i's linked list. When there are many hash collisions, hashing with separate chaining is faster than hashing with probing, since items that hash to distinct buckets cannot collide as they can in the probing case. For the DIMACS test set, however, no such time advantage accrues. On the other hand, hashing with separate chaining requires more memory than hashing with probing, since it requires overhead of at least one pointer per non-empty bucket to store the linked lists. We expect that, because of the low collision rate, hashing with linear probing is near the optimal hashing algorithm for the test sets in the DIMACS suite.

7 Conclusion

The perfect hashing algorithm as described in [8] may not be practical due to the cost of multiplying and dividing large numbers. However, a slightly modified algorithm is not only fast in practice but also parsimonious in terms of its space requirements. The modified algorithm retains the guarantee of constant time lookup, and can also easily be made to retain the guarantee of constant time insertion and deletion as well.[7]

Experimental results on the DIMACS data set indicate that FKS hashing is within a factor of two of a much simpler algorithm in terms of both time and space utilization. This is significant, because the data set was, in general, very well behaved with respect to the simple algorithm, indicating that the added complexity of FKS hashing does not necessarily make it impractical. At the same time, its theoretical guarantees allow it to outperform simpler algorithms in cases where the data is not as well behaved. Furthermore, with several parameters controlling the load factor and the sparse array constant M, the FKS algorithm can be tuned to favor space use over time or *vice versa*.

For general applications, hashing with linear probing, or perhaps with separate chaining, makes the most sense. These algorithms are fast when the keys hash uniformly over the bucket space, a situation which can often be achieved through either high-quality, general purpose hashing functions or hashing functions tuned for the specific application. In other circumstances, however, the average-case guarantees of hashing with linear probing may not be strict enough. In such situations, if a loss of a factor of two in running time and space use is acceptable, a modified version of FKS gives superior theoretical guarantees with an acceptably small degradation in performance.

References

[1] `ftp://cs.amherst.edu/pub/dimacs/dc_demo.tar.Z`.

[2] `http://ourworld.compuserve.com/homepages/bob_jenkins/evahash.htm`.

[3] `http://www.cs.amherst.edu/ ccm/challenge5/`.

[4] `http://www.mpi-sb.mpg.de/LEDA/`.

[5] AT&T. Hsearch(ba_lib). *Unix System User's Manual, System V.3*, pages pp. 105–109, 1985.

[6] Javier Barreiro and David R. Musser. An stl hash table implementation with gradual resizing, February 1995. Available by anonymous ftp from `ftp.cs.rpi.edu` in `pub/stl/hashimp2.Z`.

[7] M. Dietzfelbinger, A. Karlin, K. Mehlhorn, F. Meyer auf der Heide, H. Rohnert, and R. Tarjan. Dynamic perfect hashing: Upper and lower bounds. *Symposium on the Theory of Computing*, 18:524–531, 1988.

[8] M. Fredman, J. Komlós, and E. Szemerédi. Storing a sparse table with $O(1)$ worst case access time. *Journal of the ACM*, 31(3):538–544, July 1984.

[9] D. Knuth. *The Art of Computer Programming*, volume III. Addison-Wesley, second edition, 1981.

[10] D. Knuth. *The Art of Computer Programming*, volume II. Addison-Wesley, second edition, 1981.

[7] It would be necessary to remove the relatively unimportant modifications in Sections 3.3 and 3.6.