Design and Analysis of Hashing Algorithms with Cache Effects

Hongbin Qi
Department of Computer Science
University of California, Davis
qi@cs.ucdavis.edu

Charles U. Martel
Department of Computer Science
University of California, Davis
martel@cs.ucdavis.edu

August 10, 1998

Abstract:

This paper investigates the performance of hashing algorithms by both an experimental and an analytical approach. We examine the performance of three classical hashing algorithms: chaining, double hashing and linear probing. Our experimental results show that, despite the theoretical superiority of chaining and double hashing, linear probing outperforms both for random lookups. We explore variations on the data structures used by these traditional algorithms to improve their spatial locality and hence cache performance. Our results also help determine the optimal table size for a given key set size. In addition to time, we also study the average number of probes and cache misses incurred by these algorithms. For most of the algorithms studied in this paper, our analysis agrees with the experimental results. As a supplementary result, we examine the behavior of random lookups to a hash table. This provides a simple way to estimate the cache miss penalties of different machines. Two conclusions can be drawn from this study. First, cache effects have a significant influence on the performance of hashing algorithms. Second, it is possible to predict fairly accurately the performance of different hashing algorithms based on the algorithm configurations and cache structures.

This work was supported by NSF grant CCR 94-03651.

www.manaraa.com

1 Introduction

Cache miss penalties are increasing dramatically. Currently, more than 30 machine cycles are needed to serve a cache miss in high performance architectures. Therefore, to design efficient algorithms, we must take cache effects into account. In this paper, we perform a study of the cache performance of hashing algorithms. The main purpose of this paper is to use experimental and analytical tools to understand and improve the performance of hashing algorithms.

The Dictionary problem, where keys may be inserted, deleted and looked up, is one of the most fundamental uses of computers, and hashing is often the method of choice for solving it. Thus it is important to find the best practical hashing schemes and to understand the empirical behavior of hashing. While hashing algorithms have been studied extensively under traditional cost models, there has been little prior work focusing on their cache effects.

Chaining, double hashing and linear probing [4, 6] are the three most classic hashing algorithms. Traditionally, chaining and double hashing are considered superior to linear probing because they disperse the keys better and thus require fewer probes. Our experiments show, however, that at least for uniform accesses, linear probing is faster than both for insertions, successful searches and unsuccessful searches. This is true unless the table is almost full or can be stored entirely in the L1 cache. Since it is rarely a good idea to have the hash table be that full, linear probing seems to be the clear winner in the settings we considered.

The reason for this difference in performance is due almost entirely to cache effects. Linear probing exhibits good spatial locality. Although chaining and double hashing require fewer probes, the poor spatial locality of their data access patterns result in more cache misses and thus make them slower than linear probing.

Cache misses can be reduced by improving spacial locality. A program exhibits spatial locality if there is a good chance that subsequently accessed data items are located near each other in memory. In this paper, we show that spatial locality is an important factor in affecting the performance of hashing algorithms, and we introduce some new hashing algorithms which improve spatial locality.

The basic operation on hash tables is a probe, that is, to examine a key in a table entry. To improve the spatial locality of hash tables, we want as many successive probes as possible to take place in a single cache block. Specially, if a cache block can hold multiple keys, we can modify the hashing algorithms in such a way that a probe sequence first examines all of the keys in a cache block.

In traditional algorithm analysis, the expected number of probes is the main standard for comparing and measuring different hashing algorithms. As cache miss penalties increase, however, this standard is no longer valid. Hashing algorithms which exhibit good spatial locality can have good performance even if they require more probes.

To improve the performance of double hashing, we introduce a variation to its table structure. We hash a key to a table entry which contains multiple key slots. The number of key slots in a table entry is set so that a table entry has exactly the same size as a cache block. When a key is hashed to a table entry, the key slots in that table entry are examined sequentially. The advantage of this approach is that all the key slots in one cache block are checked before another cache block is involved, and the number of potential cache misses is therefore reduced. We call this approach packed double hashing. Because of its better data access pattern, packed double hashing achieves notable improvement over the traditional double hashing.

We also introduce a variation on chaining. Instead of storing one key and one pointer in a table entry or list node, we store multiple keys and one pointer, so that a table entry or list node has exactly the same size as a cache block. We call this approach packed chaining. Packed chaining offers two advantages over normal chaining. First, it exhibits better spatial locality. Second, it

www.manaraa.com

usually uses less memory.

The conclusions of this paper can be summarized as follows:

(1) Analysis of hashing algorithms which is entirely based on the number of probes is often misleading.

(2) Linear probing is the clear winner compared to chaining and double hashing for uniform access patterns.

(3) Hashing algorithms designed to improved spacial locality outperform traditional algorithms.

(4) The number of probes and cache misses can often be predicted quite accurately as a function of the hashing algorithm, access pattern, and cache structure. It is also possible to combine these predictions to get an accurate estimate of the time expense of hashing algorithms.

2 Related Work

Lebeck and Wood showed several techniques that could be employed to improve cache performance [9]. They used these techniques to tune cache performance of the SPEC92 benchmarks and achieved significant speedups. The prototypes of the two major techniques used in this paper, packing and aligning, can be found in [9]. LaMarca and Ladner examined the influence of caches on the performance of heaps [7]. They presented optimizations that significantly reduced cache misses and improved the overall performance, and were able to extend their results to improve the performance of sorting algorithms [8]. They also introduced an analytical model called collective analysis that helps predict cache performance [7]. Black and Martel presented simple alternatives to standard graph representations that substantially improved the performance of breadth-first-search and depth-first-search [3].

A recent hashing paper [11] develops a collision resolution scheme which can reduce the probes compared to double hashing for some very specialized settings. However, since they only look at probes rather than execution time they don't address the effects we study here.

3 Experimental Setting

The experiments were run on two platforms: DECstation 5000/25 and ALPHA 21164. The ALPHA has an 8K byte on-chip direct mapped L1 data-cache and a 96K 3-way set-associative on-chip L2 cache, both of which use 32-byte cache blocks [1]. Please refer to appendix A for other relevant system parameters. The DECstation is an older (and slower) architecture while the ALPHA is a faster and more modern machine. We looked at both to see how the performance of different algorithms changes as we move to newer machines with higher clock speeds. The general trends were very similar for the two architectures, so we focus on the ALPHA results.

We used 8-byte keys on the ALPHA. We chose

$$hash(key) = key\%T$$

as the hash function, where T is the table size. By table size we mean the maximum number of keys a table can hold. We distinguish table size from table space, which is the absolute memory space a table occupies. Since we used random integers as keys, the modulo function sufficed. We chose

$$increment(key) = Z - (key\%Z)$$

المنارة للاستشارات
www.manaraa.com

as the increment used in double hashing and packed double hashing, where Z is a prime integer smaller than T. There are no general rules for choosing Z. We simply tried different values and chose the one which minimized the running time. Our experiments show that small Z values often outperform large ones. This is because the increment is likely to be small when Z is small, and two consecutive probes are more likely to be in the same cache block. For example, in double hashing, a Z value of 43 yields the best timing results on the ALPHA for a hash table of size 2^{22}.

A key term in hashing algorithms is load factor, which is the ratio of the number of keys stored in a table to the table size. Load factor describes the storage density of a hash table.

We used two random integer generators. One was RC5 [10], an encryption-decryption function suite, whose encryption function can be used as a random number generator. This function accepts a parameter, and generates the same random integer if passed the same parameter. As we will see in section 4, this property is particularly useful when we examine the performance of successful search. The other generator was random(), a UNIX system call, which returns 4-byte random integers on the DECstation and 8-byte random integers on the ALPHA.

We tested each algorithm with regard to insertion, successful search and unsuccessful search. Each experiment consisted of two parts, one for successful search and the other for unsuccessful search. To examine successful search, we used RC5 as the generator. We generated a set of keys, inserted them into an empty table, generated random elements from the same set of keys, and searched for them in that table. In this case, all the searches were successful. To examine unsuccessful search, we used random() as the generator. We generated a set of keys, inserted them into an empty table, then generated random keys, and searched them in that table. Since the number of keys that could be generated by random() is far larger than the possible table sizes, this means that almost all the searches were unsuccessful. For this experiment, we only measured the time spent computing hash functions, inserting and searching, and excluded the time generating random keys.

All the hash tables are cache-aligned, that is, no table entry spans two cache blocks. On the DECstation, the memory allocating function malloc() returns cache-aligned memory blocks automatically. On the ALPHA, however, some pointer adjustment must be done explicitly by the user to achieve alignment [1].

4 Results of Random Lookups

Before examing the performance of hash tables, we explore the behavior of random lookups to a hash table. Given a table of size N, we repeatedly generate a random index i ($0 \leq i < N$) and read the ith entry of the table. In this experiment we only measure the time spent on the lookup and exclude the time spent generating the random indices. This experiment shows how the cache capacity and the table size affect the Average Lookup Time (ALT) of hash tables. Consider a single-level cache configuration. When the table space is smaller than the cache, most table entries are in the cache, and every lookup is a hit (ignoring conflict misses). In this case, ALT should be a constant value. When the table size exceeds the cache capacity, however, cache misses begin to occur, and ALT should start to increase. As the table grows, the cache miss rate grows, and ALT keeps on increasing. When the table size is far larger than the cache, almost every lookup is a miss. In this case, ALT should become nearly constant again.

Figure 1 shows the results of random lookups. The X-axis is the logarithm of the table space in bytes. The Y-axis is the average time in nanoseconds for a random lookup. The keys are of three different sizes: 1 byte, 2 bytes and 4 bytes.

The results on the ALPHA are roughly what we expect: the curves are flat when the table size

المنارة للاستشارات
www.manaraa.com

is smaller than 2^{13}, which is the size of the L1 cache. The slopes increase again when the table size exceed 2^{17}, which is the size of the L2 cache. Due to paging effects, however, we are not able to observe the final level-off.

The ALT chart provides a simple way to estimate the cache miss penalties. As shown in Figure 1, the largest ALT is roughly 16 times the smallest ALT. Therefore the cost of a cache-miss lookup is roughly 16 times the cost of a cache-hit lookup. We can also use the ALT chart to estimate the cache miss penalties on machines with multiple caches, although more computation is needed.

5 Results of Hashing Algorithms

We study successful searches and unsuccessful searches. To look at successful searches, we use RC5 [10] as the random integer generator. RC5 is an encryption-decryption function suite, whose encryption function can be used as a random number generator. We generate a random set of keys, insert them into an empty table, generate random elements from the set of inserted keys, and look them up in the table. Since we insert and search from the same set of keys, all the searches are successful. Using RC5 we can generate random successful searches while using only minimal extra space to avoid polluting the cache. To look at unsuccessful searches, we use random() as the random integer generator. We generate a set of keys, insert them into an empty table, generate another sequence of keys, and look them up in the table. Since the range of keys generated by random() is far larger than the possible table size, almost all the searches are unsuccessful. We can also use RC5 for unsuccessful searches, but since random() is significantly faster than RC5, we use random() for most experiments.

We refer to hashing schemes which store all elements within the table (such as double hashing) as *closed hashing*, and schemes which use space outside the table (such as chaining) as *open hashing*.

5.1 Results for Constant Table Sizes

In this subsection, we present the results for constant table sizes. While the table sizes are constant, we vary the number of keys inserted into the table.

5.1.1 Results for Closed Hashing

In this subsection, we present the results for linear probing, double hashing and packed double hashing. We first describe how a collision is handled when searching for a key x such that $h(x) = i$. In linear probing, if a collision occurs at entry i, entries $(i+1)$, $(i+2)$, $(i+3)T$, ..., are examined until x or an empty entry is found (all addresses are taken modulo T). In double hashing, after a collision occurs at entry i, we compute *increment* using a second hash function $h_2(x)$, and examine entries $(i+increment)$, $(i+2increment)$, $(i+3increment)T$, ..., until x or an empty entry is found. In packed double hashing, we hash a key to a table entry which contains multiple key slots. The number of key slots in a table entry is set so that a table entry has exactly the same size as a cache block. When a key is hashed to entry i, the slots in that entry are examined sequentially. If x is not found and the entry has no empty slots, we compute $h_2(x)$ as in double hashing, and examine entries $(i+increment)$, $(i+2increment)T$, ..., until x or an empty slot is found (note that now T represents the number of table entries, each of which contains multiple key slots).

Figure 2 shows the results of insertions using linear probing, double hashing and packed double hashing. In Figure 2, the X-axis denotes the final load factor after all keys are inserted. The Y-axis denotes the average time to insert a key when insertions start with an empty table. Figure 2 shows

www.manaraa.com

the results for a table size of $2^{2}2$ on the ALPHA, so the load factor is varied by changing the number of keys inserted. We also ran the experiments for successful and unsuccessful searches and for other table sizes (bigger than the cache). In all cases the results were similar to those in Figure 2.

On both the ALPHA and the DECstation, linear probing outperforms double hashing significantly. As shown in Figure 2, linear probing is 20% to 46% faster than double hashing. The main reason for this is that linear probing exhibits better spatial locality than double hashing. When the table size is far larger than the cache capacity, each probe in double hashing is likely to incur a cache miss. For linear probing, the table entries are examined orderly, and a cache miss occurs only if a probe goes from the last key in a cache block to the first key in the successive cache block. Thus in our setting with four keys per cache block, only 1/4 of the probes after the first are cache misses. This effect is analyzed in section 6.

One may argue that double hashing involves a second hash function, and this may also be a reason that double hashing is not as efficient as linear probing. Our experiments show, however, that the cost of the second hash function only constitutes a negligible part of the total execution time. For example, at a load factor of 0.1, the cost of the second hash function contributes to 1% of the total execution time. Note also that, like the first hash function, the second hash function is computed at most once for each key, no matter how many probes are performed. As the load factor grows and the number of probes increases, the overhead of the second hash function becomes less and less important. Therefore we can ignore the overhead of the second hash function.

Packed double hashing outperforms double hashing, and is significantly faster than double hashing at high load factor. As shown in Figure 2, the performance of packed double hashing is close to that of linear probing. We can also see that the performance of packed double hashing is more stable. As the load factor increases, the cost of packed double hashing does not climb as dramatically as double hashing or linear probling. We believe this is because the number of cache misses incurred in packed double hashing is relatively stable.

5.1.2 Results for Chaining

In this subsection, we present results for chaining and packed chaining. In chaining, a linked list is used to store the extra keys hashed to a table entry. In packed chaining, instead of storing one key and one pointer in a table entry or list node, we store multiple keys and one pointer, so that a table entry or list node has the same size as a cache block. When a key is hashed to a table entry, the key slots in that table entry and linked list are examined sequentially.

The performance of chaining and packed chaining are compared both in terms of table size and in terms of table space. As noted earlier, by table space we mean the absolute memory space a table occupies. The motivation for comparing their performance in terms of table space comes from the fact that chaining uses more space than packed chaining when both have the same table size. By comparing their performance in terms of table size, we see how much benefit packed chaining can gain by using less space for each key.

Figure 2 shows the results of insertions using chaining and packed chaining. There are two curves for chaining. Chaining 1 uses the same table space as packed chaining and thus has a higher load factor. Chaining 2 uses the same table size as packed chaining so it has the same load factor. The table size is 2^{22}. The Y-axis still denotes the average time to insert a keys when insertions start with an empty table. To explain the meaning of the X-axis, we need some further clarification of the table structures.

As noted earlier, we use 8-byte keys and pointers on the ALPHA. Since a cache block is 32 bytes on the ALPHA we can put exactly three keys and one pointer in a cache block. We can see

المنارة للاستشارات
www.manaraa.com

from these numbers that chaining uses one half more space than packed chaining when both have the same table size. For chaining 1 and packed chaining, the X-axis still denotes the load factor. Chaining 2 uses the same table space as packed chaining, which means the load factor of chaining 2 is 1.5 times that shown on the X-axis.

On the ALPHA, packed chaining outperforms chaining not only with the same load factor, but also with the same table space. As our analysis shows, packed chaining incurs more probes but fewer cache misses because of its better spatial locality. Therefore packed chaining excels due to the ALPHA's high cache miss penalty.

Again we ran the experiments for successful and unsuccessful searches and for other table sizes, and packed chaining was consistently better than chaining.

5.2 Results for Constant Key Set Sizes

We repeat the experiments described in subsection 5.1 for various table sizes. By collecting and comparing these results, we are able to answer this question: what table size performs the best for a particular key set size? We choose a key set size, and compare the performance of different table sizes for it. This can be useful in developing applications. Often a rough prediction of the key set size is available. We can then choose the appropriate table size and hashing algorithm. Note that there is likely to be a tradeoff between the number of probes and the rate of cache misses: as the table gets larger the number of probes drops but the cache miss rate rises. We will analyze this in greater detail in section 6.

When the number of keys is larger than the cache capacity but much smaller than the size of the main memory, all of the five hashing algorithms described above achieve their best performance when the load factor is relatively low, and become less efficient as the load factor grows.

We considered load factors ranging from .2 to .9. On the ALPHA, chaining and double hashing are most efficient at a load factor of 0.2, Linear probing, packed chaining and packed double hashing have almost the same performance at load factors of .2 to 0.4, and are best in this range.

Figure 3 shows the average time to insert a key for different table sizes. Here, insertions start with an empty table and end with a load factor of 0.2 for chaining and double hashing and 0.4 for linear probing, packed chaining and packed double hashing.

Among these algorithms, linear probing achieves the best performance across all table sizes for a given key set size. Linear probing is also the winner for both successful and unsuccessful searches when the table load is below 0.8. There are several reasons for this. Linear probing exhibits a good spatial locality. Unlike open hashing, it involves no pointer traversals, which are difficult to optimize and more likely to cause cache misses. It is also simple enough to beat those algorithms which use packed structures.

Processor speeds are still growing at a high rate. As the gap between the memory and the processor widens, we can expect that the advantage of linear probing over other hashing algorithms may increase.

We should add some restrictions on our results. Our successful and unsuccessful search results assume uniform access patterns and smallish keys. If the access pattern is skewed (as is true in many real applications) the number of cache misses will decrease and therefore chaining and double hashing should perform better. Also, if fewer table entries fit in a cache block (due to larger keys or data), LP's advantage due to spacial locality will decline. Preliminary results suggest that Linear Probing is still the winner for moderately skewed access patterns, but this requires further study.

المنارة للاستشارات
www.manaraa.com

6 Performance Analysis

The performance of a hashing algorithm will largely be determined by the expected number of probes and cache misses. When an algorithm probes a cache block which differs from the last accessed one we call this a *jump*. To analyze cache misses we start by studying the expected number of jumps and probes for our hashing algorithms. In our analysis we assume that the hash function hashes a key to location i with probability $1/T$ where T is the table size, and similarly, with double hashing we assume the next location probed is equally likely to be any Table location.

We start by describing our experimental results on the number of probes, jumps and cache misses in various settings. We then compare these results to our predictions.

6.1 Measurements of the Number of Probes and Jumps

Figures 4 and 5 show the average number of probes per insertion. Only the results on the ALPHA are shown, because the two platforms produce very similar results. Note that, for a given algorithm and load factor, the average number of probes is essentially the same, regardless of the table size and the key set size. It is easy to prove that the average number of probes for an insertion is the same as that for a random successful search. The number of probes for an unsuccessful search is different from that for a successful search, but the general ordering of the algorithms is the same. Thus we only show the curves for insertions.

In linear probing, packed chaining and packed double hashing, multiple probes may take place in the same cache block before another cache block is involved. We call a probe that leaves one cache block and enters another a *jump*. Because each of these jumps is likely to incur a cache miss for large table sizes, we also show the number of jumps in Figures 4 and 5. In the following subsections, we give a brief analysis of the number of probes and jumps of each algorithm. Although we rely partly on approximations, our analysis mostly agrees with the experimental results.

6.1.1 Analysis of Linear Probing

It is shown in [6] that the average number of probes incurred by an unsuccessful search in a table with load factor $\alpha = n/T$ is given by

$$\frac{1 + 1/(1-\alpha)^2}{2} \tag{1}$$

The expected number of probes per insertion into a table whose final load factor is λ is thus given by

$$\int_0^\lambda \frac{1 + 1/(1-\alpha)^2}{2} = \frac{1 + 1/(1-\lambda)}{2} \tag{2}$$

To find the expected number of jumps let B denote the number of keys which fit in a cache block, and we assume that these B entries exactly use up one cache block. Suppose an LP serch (for insertions, successful search or unsuc. search) uses k probes. The first probe is a jump. To analyze the number of addtional probes let $k - 1 = dB + r$, $r = (k-1)\ mod\ B$, then the number of additional cache blocks hit when:
$r = 0$, always hits exactly d more
$r = 1$, hits d more unless start in the last entry of a block, then hit $d + 1$
...

المنارة للاستشارات
www.manaraa.com

$r = B - 2$, hit $d + 1$ unless start in first or second entry of a block, then hit d
$r = B - 1$, hit $d + 1$ unless start in first position, in which case hit d.

Since we assume we start the search at a random location, each starting position in a block has probability $1/B$. Thus the expected number of jumps for k consecutive probes is

$$1 + d + \frac{r}{B} = 1 + \frac{dB + r}{B} = 1 + \frac{k-1}{B} \tag{3}$$

Since this formula is linear in k, we can get the expected number of jumps as a function of the load factor by replacing k in equation 3 above by the expressions for the expected number of probes in equations 1 (for unsuc.) and 2 for suc. search.

The resulting formulas match the observed number of jumps almost exactly.

6.1.2 Analysis of Double Hashing

Double hashing is able to scatter the keys in the hash table and thus achieve a high level of uniformity. Thus each successive probe is overwhelming likely to be in a new cache block, so the number of probes and jumps will be essentially equal.

The probability that a particular table entry is occupied is α, and the expected number of probes (and jumps) for an unsuccessful search is:

$$\sum_{K=1}^{T} K(1-\alpha)\alpha^{K-1} = \frac{1}{1-\alpha} \tag{4}$$

where T is the table size. This is also the expected number of probes to insert a key into a table with load factor α, so the expected number of probes and jumps for a successful search is:

$$\int_0^{\lambda} \frac{1}{1-\alpha} = \frac{-\ln(1-\lambda)}{\lambda} \tag{5}$$

6.1.3 Analysis of Packed Double Hashing

This is the most difficult setting for us to analyze, but we can get some reasonable approximations. For Packed Double Hashing (PDH) we assume the table is partitioned into S slots where each slot can hold B keys and takes up one cache block. Thus the table can hold a total of SB keys, and $n/(SB) = \alpha$ is the load factor. To begin with we analyze the expected number of jumps for PDH. Let $\lambda = n/S$ which is the average number of keys per slot.

Suppose we randomly throw n balls into S bins. We want to know the probability that a bin contains a specific number of balls. This is a classic result if there is no upper limit on the number of balls a bin can hold. This is also exactly the case of chaining. For PDH a bin can hold only B balls, and we know of no closed form solution to this problem.

We can approximate this using the following model: In *Round one* randomly throw n balls into S bins. However, if B balls end up in a bin, any additional balls *bounce*, and will have to be assigned to some other bin in a later round.

In Round 2 all balls which bounced in Round 1 are again randomly assigned to bins. If the bin has fewer than B balls (counting those from Round 1 and earlier balls from Round 2), the new ball is added, otherwise it bounces again and must be reinserted in Round 3.

Rounds continue untill all balls have been inserted.

It is fairly easy to show that the rounds model is equivalent to PDH: the P_i values are the same for both, and the expected number of bounces to insert all items is exactly the same as the expected number of double hashes done in PDH.

المنارة للاستشارات
www.manaraa.com

We can easily analyze round 1 using known results on the probability of a certain number of balls in a bin [5]: Let P_j be the probability we end up with j balls in a bin at the end of Round 1.

$$P_j = \frac{\lambda^j \exp(-\lambda)}{j!} \tag{6}$$

Let P_{B+} denote the expected fraction of slots which are full after Round 1. $P_{B+} = 1 - P_0 - P_1 - \ldots - P_{B-1}$. The expected number of balls which do not bounce in round 1 is $NB = S(P_1 + 2P_2 + \ldots + (B-1)P_{B-1} + BP_{B+})$. Thus, the expected number of balls which bounce in Round 1 is $n - NB$.

Analyzing later rounds precisely gets more complicated, however as long as λ is not too close to B not too many balls will bounce in Round 1, and we can approximate later rounds by assuming no bin gets more than one new ball in these rounds. Thus a ball only bounces if it hits a bin which was full in the prior round, and non-full bins increase by one if they are hit by some ball in that round. For example, if $B = 4$ as in our experiments, and $\lambda = 3.2$ which is a .8 load factor, we expect less than 13% of the items to bounce in round 1. Using equation 6 above, if we now insert $(.13)(3.2)S$ balls into S slots, we expect fewer than .5% of the original balls to bounce by having 2+ balls hit a bin which was not full.

For successful search, this approximate analysis of the number of jumps was within 1% of our experimental results for load factors below .8, was off by 4% at .8 but off by 12% at .9. We can get more accurate results by analyzing Round two more precisely, or simply simulating the setting.

6.2 Probes for Packed Double Hashing

Each bounce is preceded by B probes (though if we sorted the keys in a slot it might be faster). The expected number of probes depends on the order we visit locations in a slot. If we insert by starting at position one, and look at two, three, ..., B until we hit a free slot, then each slot containing k keys used i probes, for $i = 1, 2, ..., k$ to insert the ith item into that slot, for a total of $k(k+1)/2$. Recall that $\lambda = n/S$ and let R_i be the expected number of slots which end up with i keys. if J is the expected number of bounces per insertion, the expected number of probes is: $BJ + (1/\lambda)(R_1 + 3R_2 + \ldots + B(B+1)R_B/2)$.

Using the approximation described in the prior section we can estimate J and the R_i values for a given B and α. These results were within 10% of our observed results for loads below .8, but are not as good as for estimating the jumps.

6.2.1 Analysis of Chaining

The probability that a linked list is of length j, is just P_j as in equation 6 of the prior section, and the expected number of probes for successfull and unsuccessful search are well known [6]. In our experiments each linked list node took up an entire cache block, so each probe is a jump. Even if the nodes are smaller than a cache block, since each probe follows a link to an unpredicatable memory location, it is likely to be a jump.

6.2.2 Analysis of Packed Chaining

Packed chaining is very similar to chaining, except that a table entry can contain multiple keys. As in PDH we have S slots each of which can hold B keys plus a pointer (so B is likely smaller than for PDH). We let $\lambda = n/S$, so the probability j keys hash to a slot is given by P_j in equation 6. We also use P_{j+} to denote the probability a slot has j or more keys.

المنارة للاستشارات
www.manaraa.com

Thus the expected fraction of keys in the table is: $I_0 = (P_1+2P_2+\ldots+(B-1)P_{B-1}+BP_{B+})S/n$.

Similarly, using $S/n = 1/\lambda$, let I_j be the expected fraction of keys in the jth node of a chain.

$I_1 = (P_{B+1} + 2P_{B+2} + \ldots + (B-1)P_{2B-1} + BP_{2B+})1/\lambda$

$I_j = (P_{jB+1} + 2P_{jB+2} + \ldots + (B-1)P_{(j+1)B-1} + BP_{(j+1)B+})1/\lambda$.

Looking up a key in the table takes one jump, any key in the first node of a chain takes two jumps, and so on. Thus the expected number of jumps J_s for insertions or a random successful search is just:

$J_s = I_0 + 2I_1 + 3I_2 + \ldots$

The I_j values fall off rapidly once $Bj > \lambda$ so summing a moderate number of terms gives an accurate answer.

For unsuccessful search we have to serch to the end of the chain (unless we added some order to the nodes in the chain). Thus J_u the expected number of jumps for an unsuccessful search which starts at a random slot is:

$J_u = (P_0 + P_1 + \ldots + P_B) + 2(P_{B+1} + \ldots + P_{2B}) + 3(P_{2B+1} + \ldots + P_{3B}) + \ldots$

Again, the P_j terms fall off rapidly once $j > \lambda$ so it is easy to compute J_u.

6.3 Probes in Packed Chaining

To count the expected number of probes for packed chaining note that we can view the keys hashed to a slot as forming an ordinary linked list (if k keys hash to a given slot, one key is looked up with a single probe, one with two probes, ...). Thus we can use the standard analysis of the probes for successful search in chaining treating the load factor as $n/S = \lambda$.

$$1 + \frac{\lambda}{2} \tag{7}$$

Similarly, the expected number of probes per unsuccessful search is given by

$$e^{-\lambda} + \lambda + A \tag{8}$$

where A is an adjustment value. Why do we need an adjustment value? In normal chaining, the probing process is ended when a null pointer is met, and the number of probes is exactly the length of the linked list. In packed chaining, however, a table entry (list node) contains multiple key slots. If a table entry or list node is partially occupied, we cannot stop probing until we find that the next key slot is empty. In this case, one more probe is performed, and A is just the probability that one more probe is performed. Obviously, A is equal to the probability that a table entry (list node) is neither empty nor fully occupied and can be computed using the P_j values.

6.4 Analysis of the Number of Cache misses

The prior analysis looked at jumps and probes. Here we try to look more precisely at which jumps will be cache misses. We first consider the expected time to perform a random unsuccessful search in double hashing. This is modeled quite accurately by assuming that we simply select locations in the table uniformly at random until we hit an empty entry. Let C be the cache capacity measured in units of table entries. Let α be the load factor and P the cache miss penalty (so reading a location in the cache takes one time unit and reading a location not in cache takes $P + 1$ time units). Here we assume a simple two level memory system with a single cache. Let B be the number of table entries which fit into a cache block (so $B = 4$ on the ALPHA if we store 8-byte keys in the table).

For $T > C$, for each probe into the table, the probability that the location probed is not in the cache can be approximated by

المنارة للاستشارات
www.manaraa.com

$$\frac{T-C}{T} \tag{9}$$

And it is well known that the expected number of probes for a random unsuccessful search is $\frac{1}{1-\alpha}$ [6]. Therefore the expected cost of a random unsuccessful lookup is

$$\frac{1}{1-\alpha}(1+\frac{T-C}{T}P) \tag{10}$$

To study the behavior of this function with respect to T we take its derivative which is

$$\frac{PC-(P+1)n}{(T-n)^2} \tag{11}$$

The most interesting feature of the derivative is that it is always negative when $n > C$. Therefore if the keys do not fit in the cache, the expected cost keeps decreasing as we make the table bigger. Note that this is true regardless of P, the cache miss penalty. In fact, we can extend this analysis to a two level cache as well, which again shows that if n is larger than the size of the $L2$ cache it is optimal to keep increasing the table size (presumably up to the point where paging effects start and the models break down). If the key set is bigger than the L1 cache but smaller than the L2 cache, the models suggest setting T to the size of the L2 cache.

To test the predictions of the models, we used a key set which was larger than the $L2$ cache and varied the table size. The expected time for a random unsuccessful search did decrease as the table size increased, and at approximately the rate suggested by the models.

Unfortunately, all other settings are rather complex to model their cache miss behavior precisely. Consider random successful searches in double hashing. The expected number of probes/jumps for a random successful search is well known, but the probability that a probe will be a cache hit is more complicated than in the prior case. First, only those cache blocks which contain at least one key will ever be accessed during a successful search. Thus equation (9) is immediately invalid if we perform only successful searches. In addition, cache blocks which contain different number of keys have different probabilities of being in the cache. Consider the case where there is room for 4 keys in a cache block. A block B_4 with four keys is approximately four times as likely as a block B_1 with only one key to be in the cache, since it is almost four times as likely a key in B_4 was hit recently than the key in B_1. It is straightforward to compute the probability of a cache block containing i keys for $i = 1, 2, ..., B$ for DH. Unfortunately there is also another complication for all successful search settings. Consider a location i in the table. Any key k such that $h(k) = i$ starts its search at location i. Thus some locations will be accessed more frequently to start a search. Location i may also be accessed if it is on the probe sequence for a key which does not hash initially to i (in LP, DH and PDH). Thus different cache blocks may have rather different probabilities of being accessed based on the number of keys they hold and the number of probe sequences which "hit" them.

For Linear Probing consider a region of 25 consecutive filled locations in the hash table. For an unsuccessful search, any probe which starts at any of these locations will end at the empty location following this region. Therefore the cache block containing that empty location is more likely to be in cache than the cache block containing the first location in the filled region. Therefore, even for unsuccessful searches, we cannot just use equation (9) to compute the probability that a jump is a cache hit.

These complications are actually all good things for performance: even if we make the table large, empty cache blocks will not interfere with successful searches, and blocks with more keys are both more likely to be accessed and more likely to be in the cache.

المنارة للاستشارات
www.manaraa.com

An additional consideration is that some new machines (such as the Pentium II) use prefetching to start loading cache block $i+1$ from memory as soon as cache block i is accessed. In this case it is reasonable to approximate the number of cache misses for LP as one per lookup, which makes the analysis much simpler and makes linear probing more attractive.

6.4.1 Analysis of Other Settings

Chaining has some of the same complications as linear probing: memory locations at the start of long chains are more likely to be in cache than others.

6.5 Simulation of the Number of Cache misses

We used Atom [2] to simulate the cache behavior of each hashing algorithm. We simulated a direct-mapped single-level cache which has exactly the same configuration as the DECstation cache. We chose to simulate the DECstation cache because a single-level cache would make our experimental results easier to analyze and more representative. Figure 6 shows the average number of cache misses per insertion.

A cache miss occurs only when a jump occurs. The probability that a jump causes a miss is equal to the probability that the target cache block of this jump is not in the cache.

We see that the number of cache misses roughly tracks the timinging performance we saw in Figure 2. Linear probing performs somewhat better than the cache miss curves suggest and chaining somewhat worse. This may be due to the simpler address calculations in linear probing or due to easier optimizations of non-pointer based code by the compiler.

7 Conclusion

This paper investigates the performance of hashing algorithms by both an experimental and an analytical approach. We examine the performance of several classical hashing algorithms and introduce simple variations to the data structures used by these algorithms to improve their spatial locality and hence cache performance. We also present a brief analysis of the expected number of probes and cache misses. For most of the algorithms studied in this paper, our analysis agrees with the experimental results. Two conclusions can be drawn from this study. First, cache effects have a significant influence on the performance of hashing algorithms. Second, it is possible to predict fairly accurately the performance of different hashing algorithms based on the algorithm configurations and cache structures.

There are several important additional areas to study. First, it is important to consider various data sizes associated with the keys. Second, it would be good to consider skewed access patterns, which occur quite often in real applications. Third, it would be enlightening to study hashing when other memory intensive operations are also being used. Finally, there are a number of other hashing schemes which were not studied in our experiemnts.

8 Appendix A: System Parameters

The ALPHA has 64M memory and runs Linux version 2.0. It has an 8K direct-mapped L1 data cache and a 96K 3-way associative L2 data-instruction cache, with 32 bytes per cache block. The DECstation has 32M memory and runs Ultrix version 4.3. It has a 64K-byte direct-mapped data-instruction cache, with 16 bytes per cache block.

المنارة للاستشارات
www.manaraa.com

Time expense was measured using times(), a UNIX system routine. All the results reported in this paper were the average of 100 timing experiments run at 10 different times. All the programs were written in C language and compiled using the vendor's native cc command under optimization level 4.

References

[1] Digital Semiconductor 21164 ALPHA Microprocessor Hardware Reference Manual. Digital Equipment Corporation, Maynard, MA, 1997.

[2] A. Srivastava and A. Eustace. ATOM: A system for building customized program analysis tools. In Proceedings of the 1994 ACM Symposium on Programming Language Design and Implementation, pages 196-205, 1994.

[3] John Black, Charles Martel. Designing fast graph data structures: an experimental approach. Preprint 1997.

[4] Thomas Cormen, Charles Leiserson and Ronald Rivest. Introduction to algorithms. The MIT Press, 1990.

[5] William Feller. An introduction to probability theory and its applications. Volume 1, second edition. John Wiley and Sons Publishing Company, 1957.

[6] Donald Knuth. Sorting and searching, the art of computer programming, Volume 3. Addison-Wesley Publishing Company, 1973.

[7] Anthony LaMarca and Richard Ladner. The influence of caches on the performance of heaps. Journal of Experimental Algorithms, Volume 1, 1996.

[8] Anthony LaMarca and Richard Ladner. The influence of caches on the performance of sorting. In the Eighth Annual ACM-SIAM Symposium on Discrete Algorithms, 1997.

[9] Alvin Lebeck, David Wood. Cache profiling and the SPEC benchmarks: a case study. Preprint 1997.

[10] Ronald Rivest. The RC5 encryption algorithm. Proceedings of the Second International Workshop on Fast Software Encryption. 1994, Leuven, Belgium.

[11] B. Smith, G. Heileman, and C. Abdallah. The Exponential Hash Function. *Journal of Experimental Algorithms,* Vol.2, 1997.

المنارة للاستشارات

www.manaraa.com

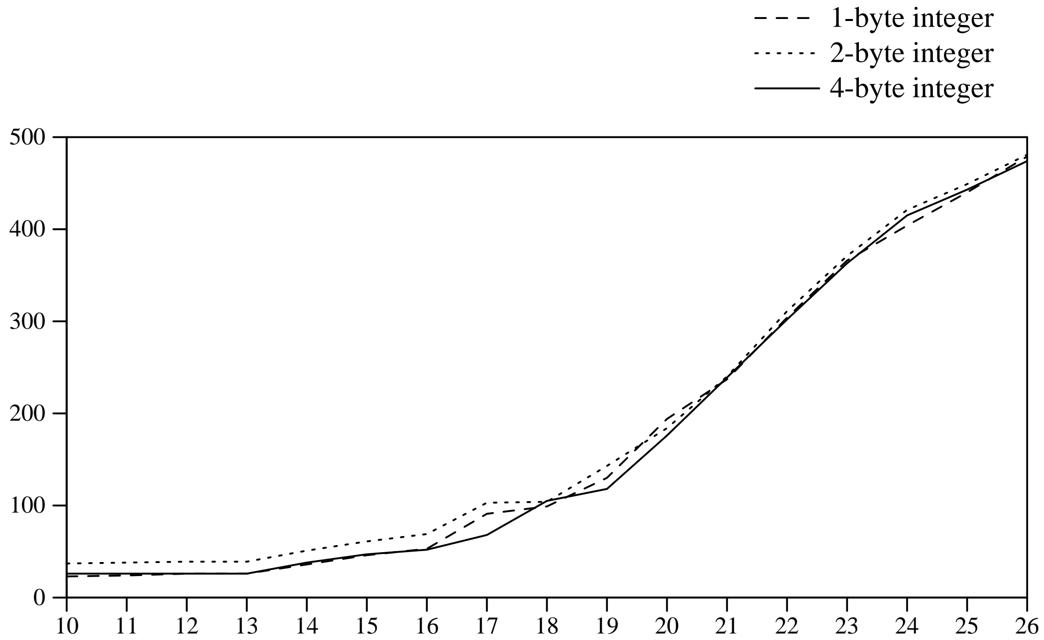

Figure 1. Time expense of random lookups on the ALPHA. The X-axis is the logarithm of table space in bytes. The Y-axis is the average time in nanoseconds to perform a lookup.

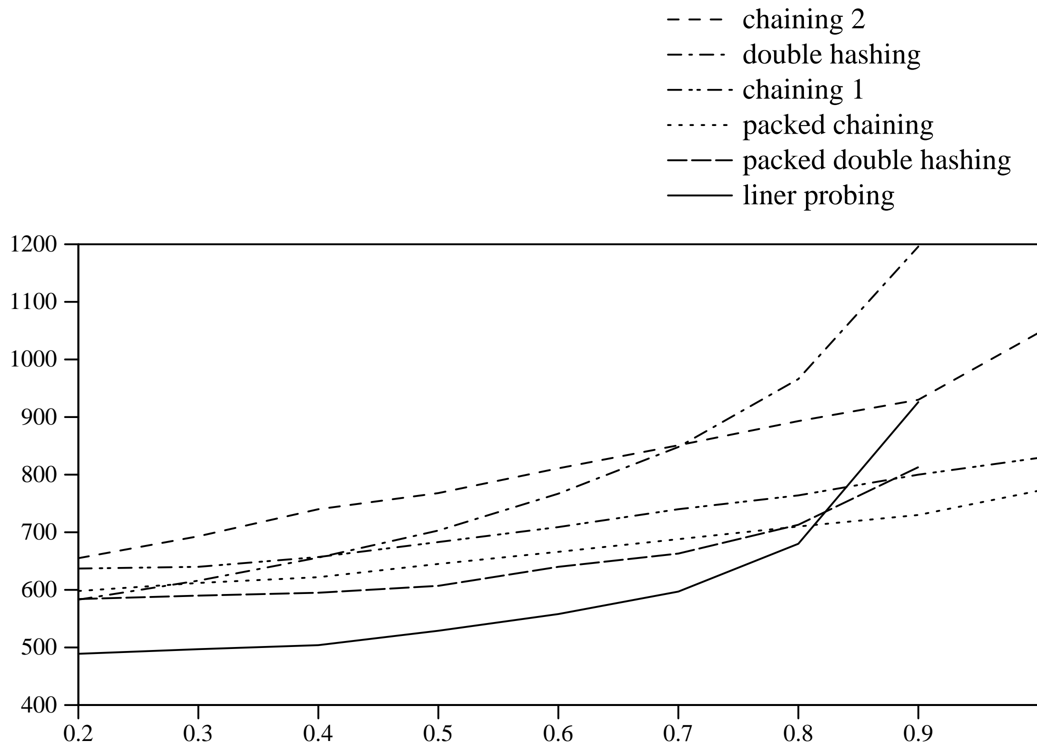

Figure 2. Time expense of insertions on the ALPHA. The X-axis is the load factor. The Y-axis is the average time in nanoseconds to insert a key. Insertions start with an empty table and end at the load factor on the X-axis. The table has 4M key slots.

المنارة للاستشارات
www.manaraa.com

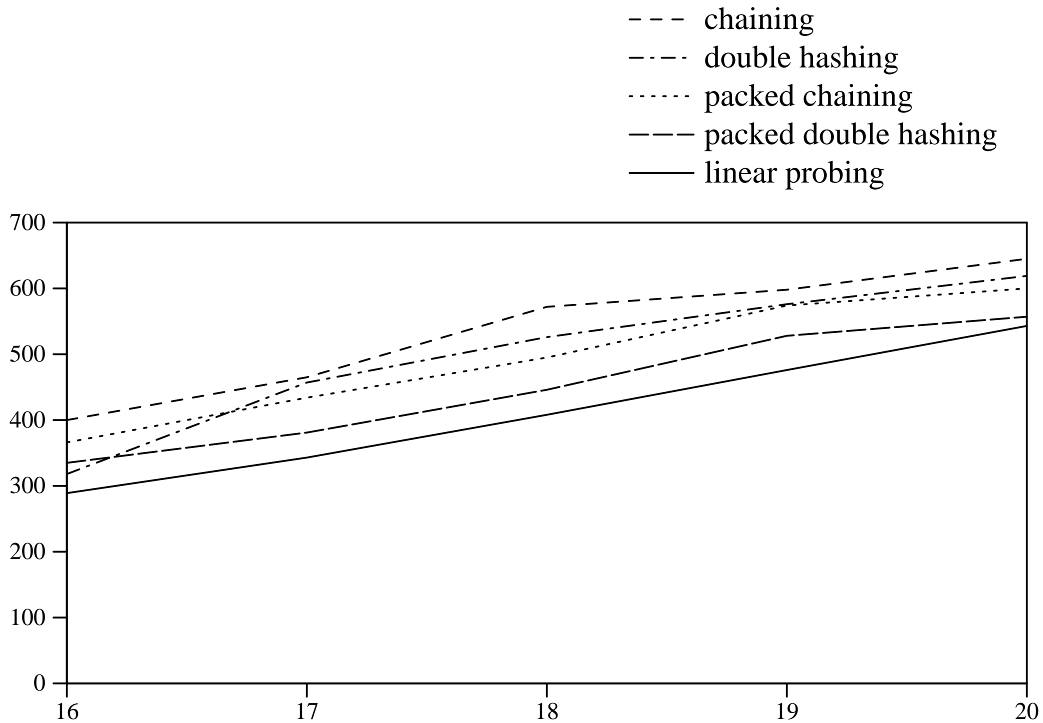

Figure 3. Time expense of insertions on the ALPHA. The X-axis is the logarithm of key set size. The Y-axis is the average time in nano-seconds to insert a key. Insertions start with an empty table and end with a load factor of 0.2 for chaining and double hashing and 0.5 for linear probing, packed chaining and packed double hasing.

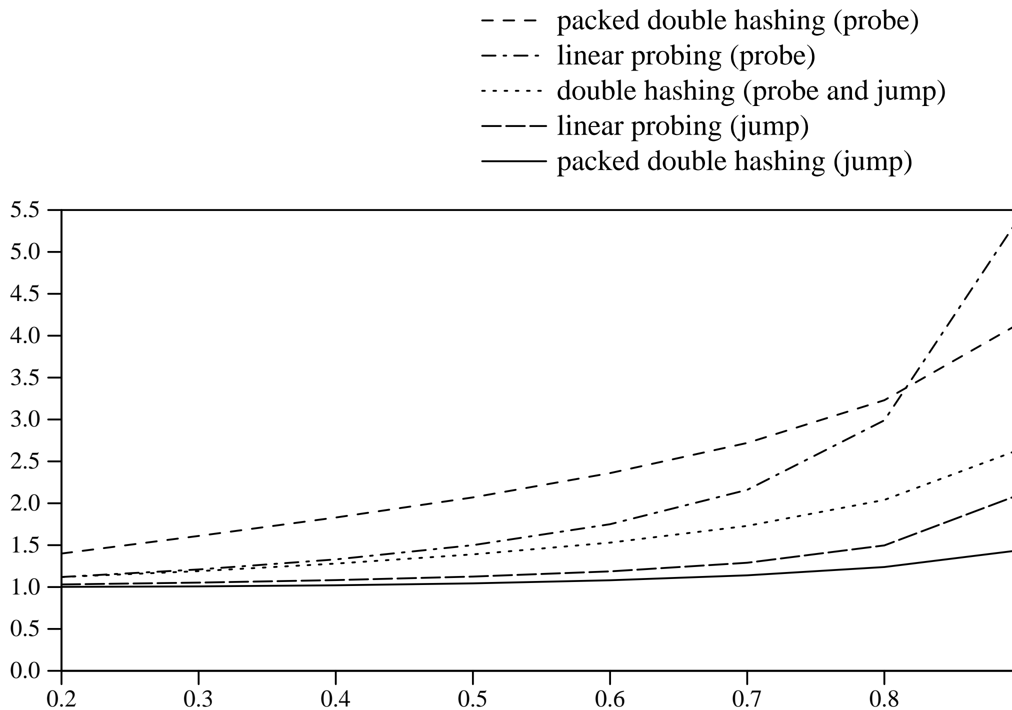

Figure 4. Average number of probes and jumps per insertion using closed hashing. The X-axis is the load factor. The table has 4M key slots.

المنارة للاستشارات
www.manaraa.com

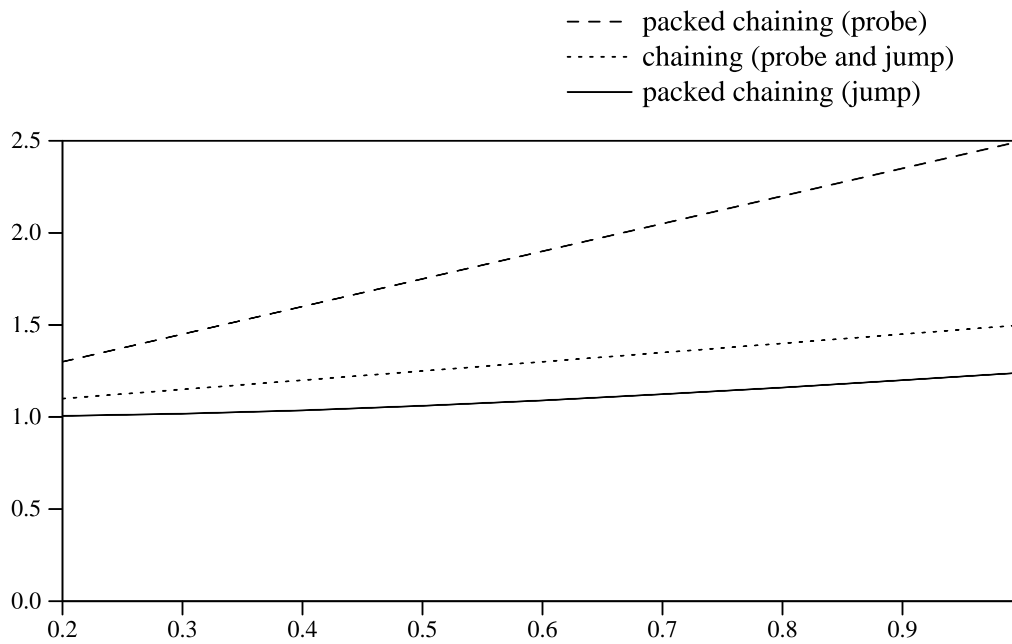

Figure 5. Average number of probes and jumps per insertion using chaining. The X-axis is the load factor. The table has 4M key slots.

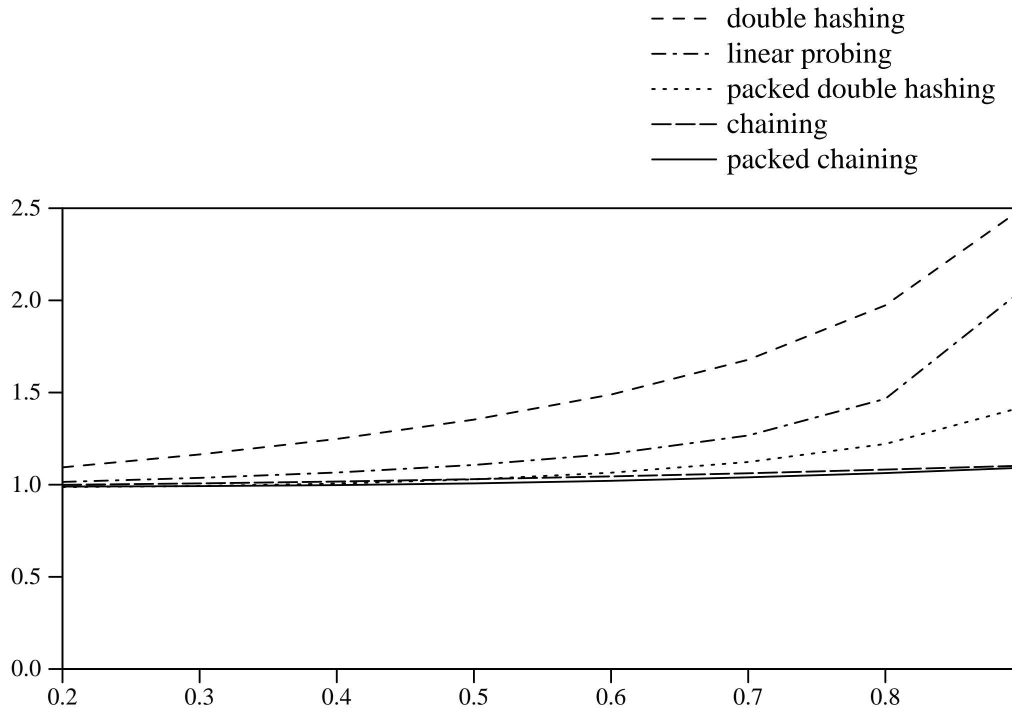

Figure 6. Average number of cache misses per insertion. The X-axis is the load factor. The table has 4M key slots.

المنارة للاستشارات
www.manaraa.com