Alcom-FT Technical Report Series
ALCOMFT-TR-01-124

Cuckoo Hashing

Rasmus Pagh* and Flemming Friche Rodler

BRICS**
Department of Computer Science
University of Aarhus, Denmark
{pagh,ffr}@brics.dk

Abstract. We present a simple and efficient dictionary with worst case constant lookup time, equaling the theoretical performance of the classic dynamic perfect hashing scheme of Dietzfelbinger et al. The space usage is similar to that of binary search trees, i.e., three words per key on average. The practicality of the scheme is backed by extensive experiments and comparisons with known methods, showing it to be quite competitive also in the average case.

1 Introduction

The *dictionary* data structure is ubiquitous in computer science. A dictionary is used to maintain a set S under insertion and deletion of elements (referred to as *keys*) from a universe U. Membership queries ("$x \in S$?") provide access to the data. In case of a positive answer, the dictionary also provides a piece of *satellite data* that was associated with x when it was inserted.

A large literature, surveyed in Sect. 1.1, is devoted to practical and theoretical aspects of dictionaries. It is common to study the case where keys are bit strings in $U = \{0,1\}^w$ and w is the word length of the computer (for theoretical purposes modeled as a RAM). Section 2 briefly discusses this restriction. It is usually clear how to return associated information once membership has been determined, e.g., in all methods discussed in this paper, the associated information of $x \in S$ can be stored together with x in a hash table. Therefore we disregard the time and space used to handle associated information and concentrate on the problem of maintaining S. In the following, we let n denote $|S|$.

The most efficient dictionaries, in theory and in practice, are based on hashing techniques. The main performance parameters are of course lookup time, update time and space. In theory, there is no trade-off between these, as one can simultaneously achieve constant lookup time, expected amortized constant update time, and space within a constant factor of the information theoretical minimum of $B = \log \binom{|U|}{n}$ bits [3]. In practice, however, the various constant factors are crucial for many high performance applications such as large databases. In particular, lookup time is a critical parameter. The challenge is to combine

* Partially supported by the IST Programme of the EU under contract number IST-1999-14186 (ALCOM-FT). Work initiated while visiting Stanford University.
** Basic Research in Computer Science (www.brics.dk), funded by the Danish National Research Foundation.

this with a reasonable space usage, as the expected time for all operations can be made a factor $(1+\epsilon)$ from optimal (one universal hash function evaluation, one memory lookup) if space $O(n/\epsilon)$ is allowed. In particular, we only consider schemes using $O(n)$ words of space.

The contribution of this paper is a new, simple hashing scheme called *cuckoo hashing*. A description and analysis of the scheme is given in Sect. 3, showing that it possesses the same theoretical properties as the dynamic dictionary of Dietzfelbinger et al. [8], namely worst case constant lookup time and amortized expected constant time for updates. A special feature of the lookup procedure is that (disregarding accesses to a small hash function description), there are just two memory accesses and they are *independent* and can be done in parallel if this is supported by the hardware. Our scheme works for space similar to that of binary search trees, i.e., three words per key in S on average.

Using weaker hash functions than those required for our analysis, the scheme is very simple to implement. Section 4 describes such an implementation, and reports on extensive experiments and comparisons with known methods, having no worst case guarantee on lookup time. Our experiments show the scheme to be quite competitive, in particular when the dictionary is small enough to fit in cache. We thus believe it to be attractive in practice, when a worst case guarantee on lookups is desired.

1.1 Previous work

Hashing, first described by Dumey [10], emerged in the 1950's as a space efficient heuristic for fast retrieval of keys in sparse tables. Knuth surveys the most important classical hashing methods in [14, sec. 6.4]. These methods also seem to prevail in practice. The most prominent, and the basis for our experiments in Sect. 4, are CHAINED HASHING (with separate chaining), LINEAR PROBING and DOUBLE HASHING. We refer to [14, sec. 6.4] for a general description of these schemes, and detail our implementation in Sect. 4.

Theoretical work. Early theoretical analysis of hashing schemes was typically done under the assumption that hash function values were random and independent. Precise analyses of the average and expected worst case behaviors of the abovementioned schemes have been made, see e.g. [12]. We mention just that for LINEAR PROBING and DOUBLE HASHING the expected *longest* probe sequence, when n is a constant fraction (< 1) of the table size, is of length $\Theta(\log n)$. For CHAINED HASHING the expected maximum chain length is $\Theta(\log n / \log\log n)$.

Though the results seem to agree with practice, the randomness assumptions used for the above analyses are questionable in applications. Carter and Wegman [4] succeeded in removing such assumptions, introducing the concept of *universal* hash function families. When implemented with a random function from Carter and Wegman's universal family, chained hashing has constant expected time per dictionary operation (plus an amortized expected constant cost for resizing the table). Constructions of universal hash function families with very efficient evaluation have since appeared [5, 7].

A dictionary with worst case constant lookup time was first obtained by Fredman, Komlós and Szemerédi [11], though it was *static*, i.e., did not support updates. It was later augmented with insertions and deletions in amortized expected constant time by Dietzfelbinger et al. [8]. Dietzfelbinger and Meyer auf der Heide improved the update performance by exhibiting a dictionary in which operations are done in constant time with high probability, i.e., probability at least $1-n^{-c}$, where c is any constant of our choice [9]. A simpler dictionary with the same properties was later developed [6]. When $n = |U|^{1-o(1)}$ a space usage of $O(n)$ words is not within a constant factor of the information theoretical minimum. The dictionary of Brodnik and Munro [3] offers the same performance as [8], using $O(B)$ bits in all cases.

Experimental work. Although the above results leave little to improve from a theoretical point of view, large constant factors and complicated implementation hinder direct practical use. For example, the "dynamic perfect hashing" scheme of [8] uses more than $35n$ words of memory. The authors of [8] refer to a more practical variant due to Wenzel that uses space comparable to that of binary search trees. According to [13], the implementation of this variant in the LEDA library [17], described in [21], has average insertion time larger than that of AVL trees for $n \leq 2^{17}$, and more than four times slower than insertions in chained hashing[1]. The experimental results listed in [17, Table 5.2] show a gap of more than a factor of 6 between the update performance of chained hashing and dynamic perfect hashing, and a factor of more than 2 for lookups[2].

Silverstein [20] explores ways of improving space as well as time of the dynamic perfect hashing scheme of [8], improving both the observed time and space by a factor of roughly three. Still, the improved scheme needs 2 to 3 times more space than linear probing to achieve similar time per operation. It should be noted that emphasis in [20] is very much on space efficiency. For example, the hash tables of both methods are stored in a packed representation, presumably slowing down linear probing considerably.

In the above we focused on experimental work on dictionaries assuring worst case constant lookup time. A survey of other experimental work is beyond the scope of this paper, but we do remark that Knuth's selection of algorithms seems to be in agreement with current practice for implementation of general purpose dictionaries. In particular, the excellent cache usage of Linear Probing makes it a prime choice on modern architectures.

2 Preliminaries

Our algorithm uses hash functions from a *universal* family:

Definition 1. *A family* $\{h_i\}_{i\in I}$, $h_i : U \to R$, *is* (c,k)-universal *if, for any* k *distinct elements* $x_1,\ldots,x_k \in U$, *any* $y_1,\ldots,y_k \in R$, *and uniformly random* $i \in I$, $\Pr[h_i(x_1) = y_1,\ldots,h_i(x_k) = y_k] \leq c/|R|^k$.

1 On a Linux PC with an Intel Pentium 120 MHz processor.
2 On a 300 MHz SUN ULTRA SPARC.

One appealing construction of a $(1, k)$-universal family, for $U = \{0,1\}^u$ and range $R = \{0,1\}^r$, is due to Dietzfelbinger [5]: For every choice of $0 \le a_0, a_1, \ldots, a_{k-1} < 2^{u+r}$ the family has the function $h(x) = ((\sum_{i=0}^{k-1} a_i x^i) \bmod 2^{u+r}) \operatorname{div} 2^u$. Note that modulo and division by a power of two can be implemented using bit masks and shifts. For our experiments, we will make use of the following $(2, 2)$-universal family from [7]: For every odd a, $0 \le a < 2^u$, the family has the function $h_a(x) = (ax \bmod 2^u) \operatorname{div} 2^{u-r}$.

We assume that keys from U fit in a single machine word, i.e., $U = \{0,1\}^w$. This is not a serious restriction, as universal families on longer keys can easily be obtained, e.g. by choosing a random function from the family of [5] for each word of the key, computing function values as the bitwise exclusive or of the individual function values [4]. An $(O(1), 2)$-universal family formed in this way can be used to map S injectively to $\{0,1\}^{2\log n + O(1)}$, thus effectively reducing the universe size to $O(n^2)$. A random function from such an $(O(1), 2)$-universal family is, with constant probability, injective on any sequence of n consecutive sets in a dictionary (see [8]). A result of Siegel [19] says that for any constant $\epsilon > 0$, if the universe is (reduced to be) of size $n^{O(1)}$, there is an $(O(1), O(\log n))$-universal family that can be evaluated in *constant* time, using space and initialization time $O(n^\epsilon)$. However, the constant factor of the evaluation time is rather high.

We reserve a special value $\perp \in U$ to signal an empty cell in hash tables. For Double Hashing, an additional special value is used to indicate a deleted key.

3 Cuckoo hashing

Cuckoo hashing is a dynamization of a static dictionary described in [18]. The dictionary uses two hash tables of length r (call them T_1 and T_2) and two hash functions $h_1, h_2 : U \to \{1, \ldots, r\}$. Every key $x \in S$ is stored in cell $h_1(x)$ of T_1 or $h_2(x)$ of T_2, but never in both. That is, the lookup function is:

function lookup(x)
  return $T_1[h_1(x)] = x \ \vee \ T_2[h_2(x)] = x$.
end;

It is shown in [18] that if $r \ge (1+\epsilon)\,n$ for some constant $\epsilon > 0$ (i.e., the tables are to be a bit less than half full), and h_1, h_2 are picked uniformly at random from an $(O(1), O(\log n))$-universal family, the probability that there is no way of arranging the keys of S according to h_1 and h_2 is $O(1/n)$. A suitable arrangement was shown to be computable in linear time by a reduction to 2-SAT.

We now consider a simple dynamization of the above. Deletion is of course simple to perform in constant time, not counting the possible cost of shrinking the tables if they are becoming too sparse. As for insertion, it turns out that the "cuckoo approach", kicking other keys away until every key has its own "nest", works very well. Specifically, if x is to be inserted, we first see if cell $h_1(x)$ of T_1 is occupied. If not, we are done. Otherwise set $T_1[h_1(x)] \leftarrow x$ anyway, thus making the previous occupant "nestless". This key is then inserted in T_2 in the same way, and so forth. As it may happen that this process loops, the number of iterations is bounded by a value "MaxLoop" to be specified below. If this number of iterations is reached, everything is rehashed with new hash functions, and we try once again to accommodate the nestless key. Using the notation $x \leftrightarrow y$ to express that the values of variables x and y are swapped, the following code summarizes the insertion procedure.

> **procedure** insert(x)
>   **if** lookup(x) **then return**;
>   **loop** MaxLoop **times**
>   **if** $T_1[h_1(x)] = \perp$ **then** { $T_1[h_1(x)] \leftarrow x$; **return**; }
> $x \leftrightarrow T_1[h_1(x)]$;
>   **if** $T_2[h_2(x)] = \perp$ **then** { $T_2[h_2(x)] \leftarrow x$; **return**; }
> $x \leftrightarrow T_2[h_2(x)]$;
>   **end loop**
>   rehash(); insert(x);
> **end**;

The above procedure assumes that the tables remain larger than $(1+\epsilon)\,n$ cells. When no such bound is known, a test must be done to find out when a rehash to larger tables is needed.

3.1 Analysis

We first show that if the insertion procedure loops for MaxLoop $= \infty$, it is not possible to accommodate all the keys of the new set using the present hash functions. To see this, consider the sequence $a_1, a_2, \ldots$ of nestless keys in the infinite loop. For $i, j \geq 1$ we let $A_{i,j} = \{a_i, \ldots, a_j\}$, and let j be the smallest index such that $a_j \in A_{1,j-1}$. At the time when a_j becomes nestless for the second time, the change in the tables relative to the configuration before the insertion is that a_k is now in the previous location of a_{k+1}, for $1 \leq k < j$. Let $i < j$ be the index such that $a_i = a_j$. If $i > 1$, a_j reclaims its previous location, occupied by a_{i-1}. If $i > 2$, a_{i-1} then reclaims its previous position, which is occupied by a_{i-2}, and so forth. Thus we have $a_{j+z} = a_{i-z}$ for $z = 0, 1, \ldots, i-1$ and end up with a_1 occurring again as a_{i+j-1}. Let $j' > j$ be the minimum index such that $a_{j'} \in A_{1,j'-1}$, and define $s_k = |h_1[A_{1,k}]| + |h_2[A_{1,k}]|$, i.e., the number of table cells available to $A_{1,k}$. Obviously $s_k \leq s_{k-1} + 1$, as every key a_i, $i > 1$, has either $h_1(a_i) = h_1(a_{i-1})$ or $h_2(a_i) = h_2(a_{i-1})$. In fact, $s_{j-1} = s_{j-2} \leq j-1$, because the key a_j found in $T_1[h_1(a_{j-1})]$ or $T_2[h_2(a_{j-1})]$ occurred earlier in the sequence. As all of the keys $a_j, \ldots, a_{j+i-1}$ appeared earlier in the sequence, we thus have $s_{j+i-1} = s_{j-2}$. Similarly, $s_{j'-1} = s_{j'-2}$. In conclusion, $|A_{1,j'-1}| = j' - i - 1$ and $s_{j'-1} = s_{j'-2} \leq s_{j+i-1} + (j'-2) - (j+i-1) = s_{j-2} + j' - j - i - 1 \leq j' - i - 2$. Thus, there are insufficiently many cells in the tables to accommodate $A_{i,j'-1}$ for the current choice of hash functions.

In conjunction with the result from [18], the above shows that the insertion procedure loops without limit with probability $O(1/n)$. We now turn to the analysis for the case where there is no such loop. Consider a prefix $a_1, a_2, \ldots, a_l$ of the sequence of nestless keys. The crucial fact is that there must be a subsequence of at least $l/3$ keys without repetitions, starting with an occurrence of the key a_1, i.e., the inserted key. This is obvious if there are no repetitions. As earlier, we pick i and j, $i < j$, such that $a_i = a_j$ and j is minimal, and once again have $a_{j+z} = a_{i-z}$ for $z = 0, 1, \ldots, i-1$. There can be no index $j' > j + i - 1$ such that $a_{j'} \in A_{1,j'-1}$, in that our earlier argument showed that when such indices i, j and j' can be chosen, the set cannot be accommodated. This means that both of the sequences $a_1, \ldots, a_{j-1}$ and $a_{j+i-1}, \ldots, a_l$ have no repetitions. As $a_1 = a_{j+i-1}$ and $i < j$, one of the sequences must be the desired one of length at least $l/3$.

Now, suppose the insertion loop runs for at least t iterations. By the above there is a sequence of keys $b_1, \ldots, b_m$, $m \geq (2t-1)/3$, such that b_1 is the key to be inserted, and such that for some $p \in \{0, 1\}$:

$$h_{2-p}(b_1) = h_{2-p}(b_2),\ h_{1+p}(b_2) = h_{1+p}(b_3),\ h_{2-p}(b_3) = h_{2-p}(b_4), \ldots \quad (1)$$

Given b_1, there are at most n^{m-1} length m sequences of keys. For any such sequence and any $p \in \{0, 1\}$, if the hash functions were chosen from a (c, m)-universal family, the probability that (1) holds is bounded by $c\,r^{-(m-1)}$. Thus, the probability that there is *any* sequence of length m satisfying (1) is bounded by $2c(n/r)^{m-1} \leq 2c(1+\epsilon)^{-(2t-1)/3+1}$. Suppose we use a $(c, 6\log_{1+\epsilon} n)$-universal family, for some constant c (e.g., Siegel's family with constant time evaluation [19]). Then the probability of more than $3\log_{1+\epsilon} n$ iterations is $O(1/n^2)$. Thus, we can set $\mathrm{MaxLoop} = 3\log_{1+\epsilon} n$ with a negligible increase in the probability of a rehash. When there is no rehash, the expected number of iterations is bounded by:

$$1 + \sum_{t=2}^{\infty} 2c\,(1+\epsilon)^{-(2t-1)/3+1} = O(1 + 1/\epsilon) \ .$$

To sum up, a rehash has no failed insertions with probability $O(1/n)$. In this case, the expected time per insertion is constant, so the expected time is $O(n)$. As the probability of having to start over with new hash functions is bounded away from 1, the total expected time is $O(n)$. This implies that the expected time for insertion is constant if $r \geq (1+\epsilon)(n+1)$. Resizing of tables can be done in amortized expected constant time per update by the usual doubling/halving technique.

4 Experiments

To examine the practicality of Cuckoo Hashing, we experimentally compare it to three well known hashing methods, Chained Hashing (with separate chaining), Linear Probing and Double Hashing, as described in [14, sec. 6.4]. We also consider Two-Way Chaining [1], implemented in a cache-friendly way (with average operation time $O(\log \log n)$ for linear space), as recently suggested in [2].

4.1 Data structure design and implementation

We consider positive 32 bit signed integer keys and use 0 as $\perp$. The data structures are *robust* in that they correctly handle attempts to insert an element already in the set and attempts to delete an element not in the set. A slightly faster implementation can be obtained if this is known not to occur.

Our focus is on achieving high performance dictionary operations with a reasonable space usage. By the *load factor* of a dictionary we will understand the size of the set relative to the memory used[3]. As seen in [14, fig. 44] there is not much to be gained in terms of average number of probes for the classic schemes by going for load factor below, say, 1/2 or 1/3. As CUCKOO HASHING only works when the size of each table is larger than the size of the set, we can only perform a comparison for load factors less than 1/2. To allow for doubling and halving of the table size, we allow the load factor to vary between 1/5 and 1/2, focusing especially on the "typical" load factor of 1/3. For CUCKOO HASHING and TWO-WAY CHAINING there is a chance that an insertion may fail, so we have a "forced rehash". If the load factor is larger than a certain threshold (somewhat arbitrarily set to 5/12) we use the opportunity to double the table size. By our experiments this only slightly decreases the average load factor.

Apart from CHAINED HASHING, the schemes considered have in common the fact that they have only been analyzed under randomness assumptions that are currently (or inherently) unpractical to implement ($O(\log n)$-wise independence or n-wise independence). However, experience shows that rather simple and efficient hash function families yield performance close to that predicted under stronger randomness assumptions. As mentioned earlier, we use the $(2,2)$-universal family from [7], whose evaluation consists of a 32 bit multiplication and a shift. This restricts us to considering hash tables whose sizes are powers of two. A random function from this family (chosen using C's `rand` function) appears to work fine with all schemes except CUCKOO HASHING. For CUCKOO HASHING we found that using a $(1,3)$-universal family resulted in fewer forced rehashes than when using a $(1,2)$-universal family. However, it turned out that the exclusive or of three independently chosen functions from the family of [7] was faster and worked equally well. We have no good explanation for this phenomenon. For all schemes, various other families were tried, with a decrease in performance.

All methods have been implemented in C. We have striven to obtain the fastest possible implementation of each scheme. Details differing from the references and specific choices made are:

CHAINED HASHING. We store the first element of each linked list directly in the hash table. This often saves one cache miss, and slightly decreases memory usage (in the expected sense) as every non-empty chained list is one element shorter. C's `malloc` and `free` functions were found to be a performance bottleneck, so a simple "free list" memory allocation scheme is used. Half of

[3] For CHAINED HASHING, the notion of load factor traditionally disregards the space used for chained lists, but we desire equal load factors to imply equal memory usage.

the allocated memory is used for the hash table, and half for list elements. If the data structure runs out of free list elements, its size is doubled.

DOUBLE HASHING. Deletions are handled by putting a "deleted" marker in the cell of the deleted key. Queries skip over deleted cells, while insertions overwrite them. To prevent the tables from clogging up with deleted cells, resulting in poor performance for unsuccessful lookups, all keys are rehashed whenever the hash table is 2/3 occupied by keys and "deleted" markers.

TWO-WAY CHAINING. We allow four keys in each bucket. This is enough to keep the probability of a forced rehash low for hundreds of thousands of keys, by the results in [2]. For larger collections of keys one should allow more keys in each bucket, resulting in general performance degradation.

CUCKOO. The architecture on which we ran our experiments was not able to parallelize the two memory accesses in lookups. Therefore we only evaluate the second hash function after the first memory lookup has shown unsuccessful.

Some experiments were done with variants of CUCKOO HASHING. In particular, we considered ASYMMETRIC CUCKOO, in which the first table is twice the size of the second one. This results in more keys residing in the first table, thus giving a slightly better average performance for successful lookups. For example, after a long sequence of alternate insertions and deletions at load factor 1/3, we found that about 76% of the elements resided in the first table of ASYMMETRIC CUCKOO, as opposed to 63% for CUCKOO HASHING. There is no significant slowdown for other operations. We will describe the results for ASYMMETRIC CUCKOO when they differ significantly from those of CUCKOO HASHING.

4.2 Setup and results

Our experiments were run on a PC running Linux (kernel version 2.2) with an 800 MHz Intel Pentium III processor and 256 MB of memory (PC100 RAM). The processor has a 16 KB level 1 data cache and a 256 KB level 2 "advanced transfer" cache. We believe this machine to be a good representative of a modern multi-level memory architecture. Further, our results can be explained in terms of processor, cache and memory speed in our machine, and are thus believed to have significance for other configurations. An advantage of using the Pentium processor for timing experiments is its `rdtsc` instruction which can be used to measure time in clock cycles. This gives access to very precise data on the behavior of functions. In our case it also supplies a way of discarding measurements significantly disturbed by interrupts from hardware devices or the process scheduler, as these show up as a small group of timings significantly separated from all other timings. Programs were compiled using the `gcc` compiler version 2.95.2, using optimization flags `-O9 -DCPU=586 -march=i586 -fomit-frame-pointer -finline-functions -fforce-mem -funroll-loops -fno-rtti`. As mentioned earlier, we use a global clock cycle counter to time operations. If the number of clock cycles spent exceeds 5000 and there was no rehash, we conclude that the call was interrupted and disregard the result (it was empirically observed that no operation ever took between 2000 and 5000 clock cycles). If a rehash is made, we have no way of filtering away time spent in interrupts. However, all tests were made on a "clean" machine with no irrelevant user processes, so disturbances should be minimal.

Our first test was designed to model the situation in which the size of the dictionary is not changing too much. It considers a sequence of mixed operations generated at random. We constructed the test operation sequences from a collection of high quality random bits publicly available on the web [15]. The sequences start by insertion of n distinct random keys, followed by $3n$ times four operations: A random unsuccessful lookup, a random successful lookup, a random deletion, and a random insertion. We timed the operations in the "equilibrium", where the number of elements is stable. For load factor 1/3 our results appear in Fig. 1, which shows an average over 10 runs. As LINEAR PROBING was consistently faster than DOUBLE HASHING, we chose it as the sole open addressing scheme in the plots. Time for forced rehashes was added to the insertion time. Results had a large variance for sets of size 2^{12} to 2^{16} – outside this range the extreme values deviated from the average by less than about 7%.

As can be seen, the time for lookups is almost identical for all schemes as long as the entire data structure resides in level 2 cache. After this, the average number of random memory accesses (with the probability of a cache miss approaching 1) shows up. Filling a cache line seems to take around 160 clock cycles, with the memory location looked up arriving at the processor after about 80 clock cycles on average. This makes linear probing an average case winner, with CUCKOO HASHING and TWO-WAY CHAINING following about half a cache miss behind. For insertion, the number of random memory accesses again dominates the picture for large sets, while the higher number of in-cache accesses and more computation makes CUCKOO HASHING and in particular TWO-WAY chaining relatively slow for small sets. The cost of forced rehashes sets in for TWO-WAY CHAINING for sets of more than a million elements, at which point better results would be obtained by a larger bucket size. For deletion, CHAINED HASHING lags behind for large sets due to random memory accesses when freeing list elements, while the simplicity of CUCKOO HASHING makes it the fastest scheme. We believe that the slight rise in time for the largest sets in the test is due to saturation of the bus, as the machine runs out of memory and begins swapping. It is interesting to note that all schemes would run much faster if the random memory accesses could bypass the cache (using perhaps 20 clock cycles per random memory access on our machine).

The second test concerns the cost of insertions in growing dictionaries and deletions in shrinking dictionaries. Together with Fig. 1 this should give a fairly complete picture of the performance of the data structures under general sequences of operations. The first operation sequence inserts n distinct random keys, while the second one deletes them. The plot is shown in Fig. 2. For small sets, the time per operation seems unstable and dominated by memory allocation overhead (if minimum table size 2^{10} is used, the curves become monotone). For sets of more than 2^{12} elements the largest deviation from the averages was about 6%. Disregarding the constant minimum amount of memory used by any

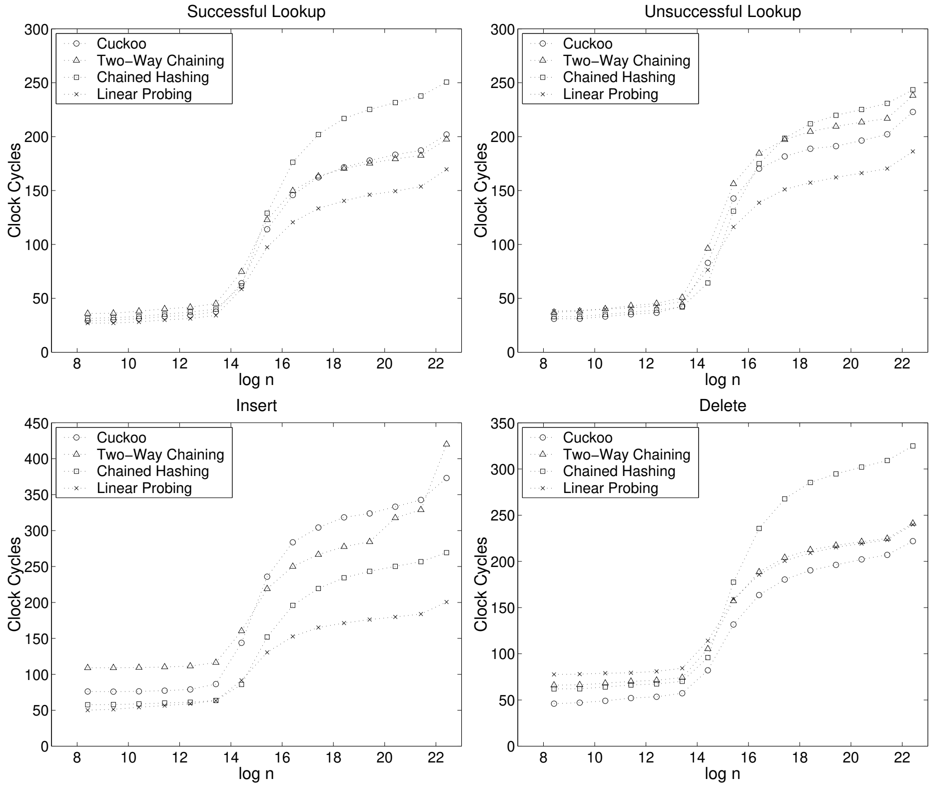

Fig. 1: The average time per operation in equilibrium for load factor 1/3.

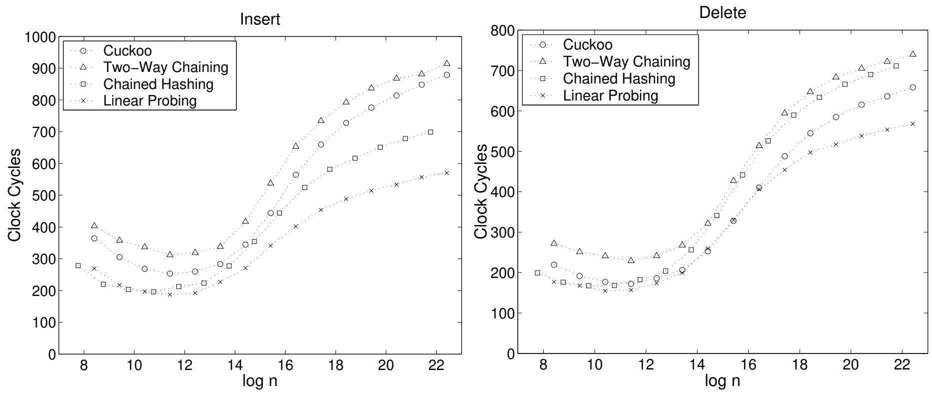

Fig. 2: The average time per insertion/deletion in a growing/shrinking dictionary for average load factor $\approx 1/3$.

dictionary, the average load factor during insertions was within 2% of 1/3 for all schemes except Chained Hashing whose average load factor was about 0.31. During deletions all schemes had average load factor 0.28. Again, the winner is Linear Probing due to very fast rehashes.

Access to data in a dictionary is rarely random in practice. In particular, the cache is more helpful than in the above random tests, for example due to repeated lookups of the same key and quick deletions. As a rule of thumb, the time for such operations will be similar to the time when all of the data structure is in cache. To perform actual tests of the dictionaries on more realistic data, we chose a representative subset of the dictionary tests of the 5th DIMACS implementation challenge [16]. The tests involving string keys were preprocessed by hashing string keys to 32 bit integers, preserving the access pattern to keys. Each test was run six times – minimum and maximum average time per operation can be found in Table 1, which also lists the average load factor. Linear probing is again the fastest, but mostly only 20-30% faster than the Cuckoo schemes.

	Joyce	Eddington	3.11-Q-1	Smalltalk-2	3.2-Y-1
Linear	42 - 45 (.35)	26 - 27 (.40)	99 - 103 (.30)	68 - 72 (.29)	85 - 88 (.32)
Double	48 - 53 (.35)	32 - 35 (.40)	116 - 142 (.30)	77 - 79 (.29)	98 - 102 (.32)
Chained	49 - 52 (.31)	36 - 38 (.28)	113 - 121 (.30)	78 - 82 (.29)	90 - 93 (.31)
A.Cuckoo	47 - 50 (.33)	37 - 39 (.32)	166 - 168 (.29)	87 - 95 (.29)	95 - 96 (.32)
Cuckoo	57 - 63 (.35)	41 - 45 (.40)	139 - 143 (.30)	90 - 96 (.29)	104 - 108 (.32)
Two-Way	82 - 84 (.34)	51 - 53 (.40)	159 - 199 (.30)	111 - 113 (.29)	133 - 138 (.32)

Table 1: Average clock cycles per operation and load factors for the DIMACS tests.

We have seen that the number of random memory accesses (i.e., cache misses) is critical to the performance of hashing schemes. Whereas there is a very precise understanding of the probe behavior of the classic schemes (under suitable randomness assumption), the analysis of the expected time for insertions in Sect. 3.1 is rather crude, establishing just a constant upper bound. Figure 3 shows experimentally determined values for the average number of probes during insertion for various schemes and load factors below 1/2. We disregard reads and writes to locations known to be in cache and the cost of rehashes. Measurements were made in "equilibrium" after 10^5 insertions and deletions, using tables of size 2^{15} and truly random hash function values. It is believed that this curve is independent of the table size (up to vanishing terms). The curve for Linear Probing does not appear, as the number of non-cached memory accesses depends on cache architecture (length of the cache line), but it is typically very close to 1.

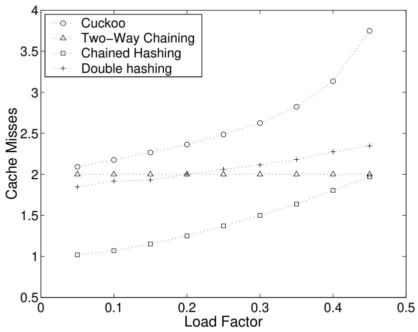

Fig. 3: The average number of random memory accesses for insertion.

5 Conclusion

We have seen a new dictionary with worst case constant lookup time that is very simple to implement and has average case performance comparable to the best previous dictionaries. Earlier schemes with worst case constant lookup time were more complicated to implement and had considerably worse average case performance. Several challenges remain: First of all an explicit practical hash function family that is provably good for the scheme has yet to be found. Secondly, we lack a precise understanding of why the scheme exhibits quite low constant factors — in particular the curve of Fig. 3 and the fact that forced rehashes are rare for load factors quite close to 1/2 need to be explained.

References

[1] Yossi Azar, Andrei Z. Broder, Anna R. Karlin, and Eli Upfal. Balanced allocations. *SIAM J. Comput.*, 29(1):180–200 (electronic), 1999.

[2] Andrei Broder and Michael Mitzenmacher. Using multiple hash functions to improve IP lookups. To appear in INFOCOM 2001, 2001.

[3] Andrej Brodnik and J. Ian Munro. Membership in constant time and almost-minimum space. *SIAM J. Comput.*, 28(5):1627–1640 (electronic), 1999.

[4] J. Lawrence Carter and Mark N. Wegman. Universal classes of hash functions. *J. Comput. System Sci.*, 18(2):143–154, 1979.

[5] Martin Dietzfelbinger. Universal hashing and k-wise independent random variables via integer arithmetic without primes. In *Proceedings of STACS '96*, volume 1046 of *Lecture Notes in Computer Science*, pages 569–580. Springer-Verlag, Berlin, 1996.

[6] Martin Dietzfelbinger, Joseph Gil, Yossi Matias, and Nicholas Pippenger. Polynomial hash functions are reliable (extended abstract). In *Proceedings of ICALP '92*, volume 623 of *Lecture Notes in Computer Science*, pages 235–246. Springer-Verlag, Berlin, 1992.

[7] Martin Dietzfelbinger, Torben Hagerup, Jyrki Katajainen, and Martti Penttonen. A reliable randomized algorithm for the closest-pair problem. *Journal of Algorithms*, 25(1):19–51, 1997. doi:10.1006/jagm.1997.0873.

[8] Martin Dietzfelbinger, Anna Karlin, Kurt Mehlhorn, Friedhelm Meyer auf der Heide, Hans Rohnert, and Robert E. Tarjan. Dynamic perfect hashing: Upper and lower bounds. *SIAM J. Comput.*, 23(4):738–761, 1994.

[9] Martin Dietzfelbinger and Friedhelm Meyer auf der Heide. A new universal class of hash functions and dynamic hashing in real time. In *Proceedings of ICALP '90*, volume 443, pages 6–19. Springer-Verlag, Berlin, 1990.

[10] Arnold I. Dumey. Indexing for rapid random access memory systems. *Computers and Automation*, 5(12):6–9, 1956.

[11] Michael L. Fredman, János Komlós, and Endre Szemerédi. Storing a sparse table with $O(1)$ worst case access time. *J. Assoc. Comput. Mach.*, 31(3):538–544, 1984.

[12] Gaston Gonnet. *Handbook of Algorithms and Data Structures*. Addison-Wesley Publishing Co., London, 1984.

[13] Jyrki Katajainen and Michael Lykke. Experiments with universal hashing. Technical Report DIKU Report 96/8, University of Copenhagen, 1996.

[14] Donald E. Knuth. *Sorting and Searching*, volume 3 of *The Art of Computer Programming*. Addison-Wesley Publishing Co., Reading, Mass., second edition, 1998.

[15] George Marsaglia. The Marsaglia random number CDROM including the diehard battery of tests of randomness. http://stat.fsu.edu/pub/diehard/.

[16] Catherine C. McGeoch. The fifth DIMACS challenge dictionaries. http://cs.amherst.edu/∼ccm/challenge5/dicto/.

[17] Kurt Mehlhorn and Stefan Näher. *LEDA*. Cambridge University Press, Cambridge, 1999. A platform for combinatorial and geometric computing.

[18] Rasmus Pagh. On the Cell Probe Complexity of Membership and Perfect Hashing. In *Proceedings of STOC '01*. ACM Press, New York, 2001. To appear.

[19] Alan Siegel. On universal classes of fast high performance hash functions, their time-space tradeoff, and their applications. In *Proceedings of FOCS '89*, pages 20–25. IEEE Comput. Soc. Press, Los Alamitos, CA, 1989.

[20] Craig Silverstein. A practical perfect hashing algorithm. Manuscript, 1998.

[21] M. Wenzel. Wörterbücher für ein beschränktes universum. Diplomarbeit, Fachbereich Informatik, Universität des Saarlandes, 1992.