Cuckoo Hashing

Rasmus Pagh [1]

IT University of Copenhagen, Rued Langgaardsvej 7, 2300 København S, Denmark

Flemming Friche Rodler [2]

ON-AIR A/S, Digtervejen 9, 9200 Aalborg SV, Denmark.

Abstract

We present a simple dictionary with worst case constant lookup time, equaling the theoretical performance of the classic dynamic perfect hashing scheme of Dietzfelbinger et al. *(Dynamic perfect hashing: Upper and lower bounds. SIAM J. Comput., 23(4):738–761, 1994).* The space usage is similar to that of binary search trees. Besides being conceptually much simpler than previous dynamic dictionaries with worst case constant lookup time, our data structure is interesting in that it does not use perfect hashing, but rather a variant of open addressing where keys can be moved back in their probe sequences. An implementation inspired by our algorithm, but using weaker hash functions, is found to be quite practical. It is competitive with the best known dictionaries having an average case (but no nontrivial worst case) guarantee on lookup time.

Key words: data structures, dictionaries, information retrieval, searching, hashing, experiments

Email addresses: pagh@itu.dk (Rasmus Pagh), ffr@onair-dk.com (Flemming Friche Rodler).

[1] Partially supported by the Future and Emerging Technologies program of the EU under contract number IST-1999-14186 (ALCOM-FT). This work was initiated while visiting Stanford University, and the draft manuscript completed at Aarhus University.

[2] This work was done while at Aarhus University.

1 Introduction

The *dictionary* data structure is ubiquitous in computer science. A dictionary is used for maintaining 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. 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. The constant factors involved are crucial for many applications. In particular, lookup time is a critical parameter. It is well known that, by using a simple universal hash function, the expected number of memory probes for all dictionary operations can be made arbitrarily close to 1 if a sufficiently sparse hash table is used. Therefore the challenge is to combine speed with a reasonable space usage. In particular, we only consider schemes using $O(n)$ words of space. Section 3 surveys the literature on such dictionaries.

The contribution of this paper is a new hashing scheme called Cuckoo Hashing, which possesses the same theoretical properties as the classic dictionary of Dietzfelbinger et al. [10], but is much simpler. The scheme has *worst case* constant lookup time and amortized expected constant time for updates. Furthermore, the space usage is roughly $2n$ words, which should be compared with the $35n$ words used in [10]. This means that the space usage is similar to that of binary search trees. A special feature of our lookup procedure is that (disregarding accesses to an asymptotically small hash function description) there are just two memory accesses, which are *independent* and can be done in parallel if this is supported by the hardware.

Using weaker hash functions than those required for our analysis, Cuckoo Hashing is very simple to implement. Section 4 describes such an implementation, and reports on experiments and comparisons with the most commonly used hashing methods, having no nontrivial worst case guarantee on lookup time. It seems that such an experiment, performed on a modern multi-level memory architecture, has not previously been described in the literature. Our experiments show Cuckoo Hashing to be quite competitive, especially 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. In contrast, the hashing scheme of [10] is known to exhibit high constant factors. The LEDA library of efficient data structures and algorithms [25] now incorporates an implementation of Cuckoo Hashing based on ours.

As in most other theoretical works on hashing we consider 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). If keys are longer, two things should be changed. 1. The keys should be stored outside the hash table, and hash table cells should contain pointers to keys. 2. Hashing of long keys should be handled using a standard technique, described for completeness in Appendix A.

It is usually, though not always, clear how to return associated information once membership has been determined. E.g., in the hash table based 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. We will reserve a special value $\perp \in U$ to signal an empty cell in hash tables.

Our algorithm uses hash functions from a *universal* family. We use the following well-known generalization of the original notion of Carter and Wegman [7].

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 $i \in I$, $\Pr[h_i(x_1) = y_1, \ldots, h_i(x_k) = y_k] \leq c/|R|^k$.*

2 Cuckoo Hashing

Cuckoo Hashing is a dynamization of a static dictionary described in [26]. The dictionary uses two hash tables, T_1 and T_2, each consisting of r words, and two hash functions $h_1, h_2 : U \to \{0, \ldots, r-1\}$. Every key $x \in S$ is stored either in cell $h_1(x)$ of T_1 or in cell $h_2(x)$ of T_2, but never in both. Our lookup function is

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

Two table accesses for lookup is in fact optimal among all dictionaries using linear space, except for special cases, see [26].

It is shown in [26] that if $r \geq (1+\epsilon)\, n$ for some constant $\epsilon > 0$ (i.e., the tables are 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

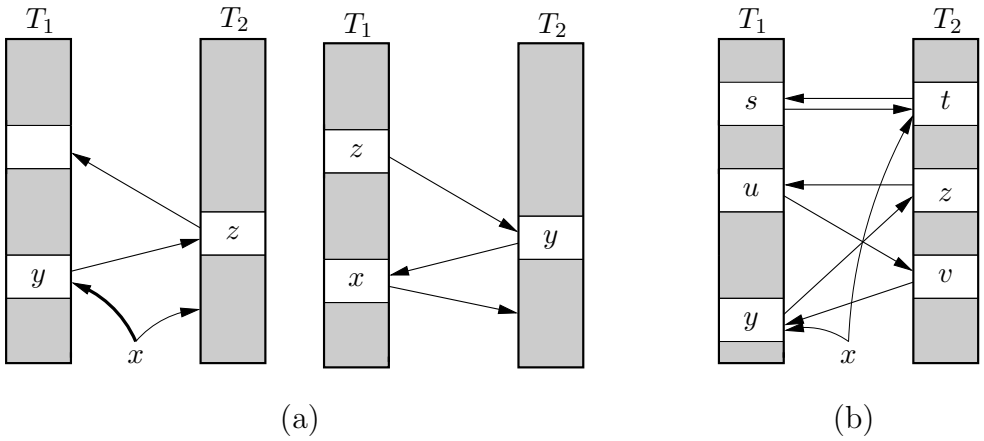

Fig. 1. Examples of Cuckoo Hashing insertion. Arrows show possibilities for moving keys. (a) Key x is successfully inserted by moving keys y and z from one table to the other. (b) Key x cannot be accommodated and a rehash is necessary.

arranging the keys of S according to h_1 and h_2 is $O(1/n)$. A suitable arrangement of the keys was shown in [26] to be computable in expected linear time, by a reduction to 2-SAT.

We now consider a simple dynamization of the above, still assuming $r \geq (1+\epsilon)\,n$ for some constant $\epsilon > 0$. 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 we 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 iteratively, see Figure 1(a).

It may happen that this process loops, as shown in Figure 1(b). Therefore the number of iterations is bounded by a value "MaxLoop" to be specified in Section 2.3. If this number of iterations is reached, we rehash the keys in the tables using new hash functions, and try once again to accommodate the nestless key. There is no need to allocate new tables for the rehashing: We may simply run through the tables to delete and perform the usual insertion procedure on all keys found not to be at their intended position in the table. (Note that kicking away a key that is not in its intended position simply corresponds to starting a new insertion of this 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
    x ↔ T1[h1(x)]
    if x = ⊥ then return
    x ↔ T2[h2(x)]
    if x = ⊥ then return
  end loop
  rehash(); insert(x)
end
```

The procedure assumes that each table remains 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. Resizing of tables can be done in amortized expected constant time per update by the usual doubling/halving technique (see, e.g., [10]).

If the hash tables have size r, we enforce that no more than r^2 insertions are performed without changing the hash functions. More specifically, if r^2 insertions have been performed since the beginning of the last rehash, we force a new rehash.

2.1 Hash functions

By a result of Siegel [35] (detailed in Appendix A) we can construct a hash function family that, when restricted to any set of r^2 keys, is $(1, n^\delta)$-universal, for some constant $\delta > 0$, with probability $1 - O(1/n^2)$. Also, we can pick from the family random functions h_1 and h_2 having constant evaluation time and a description of $o(n)$ words. Since there are at most r^2 keys inserted using a particular pair of hash functions this means that:

- With probability $O(1/n^2)$ the hash functions have some unspecified behavior (i.e., we should expect the worst possible).
- Otherwise, the hash functions behave exactly as if they had been picked from a $(1, n^\delta)$-universal family.

For n larger than some constant we will have MaxLoop $< n^\delta$, i.e., with high probability the family will be $(1, \text{MaxLoop})$-universal. This means that h_1 and h_2 will act like truly random functions on any set of keys processed during the insertion loop.

2.2 Variants

The lookup call preceding the insertion loop ensures robustness if the key to be inserted is already in the dictionary. A slightly faster implementation can be obtained if this is known not to occur.

Note that the insertion procedure is biased towards inserting keys in T_1. As will be seen in Section 4 this leads to faster successful lookups, due to more keys being found in T_1. This effect is even more pronounced if one uses an *asymmetric* scheme where T_1 is larger than T_2. In both cases, the insertion time is only slightly worse than that of a completely symmetric implementation.

Another variant is to use a single table T of size $2r$ for both hash functions. The results and analysis for this case are similar to what is described here for the two table scheme. The following trick due to John Tromp [38] can be used in this case to avoid keeping track of the hash function according to which each key is placed: If we change the possible locations for key x to be $h_1(x)$ and $(h_2(x) - h_1(x)) \bmod 2r$, we can jump from one location of x to the other using the map $i \mapsto (h_2(x) - i) \bmod 2r$.

In the following we will consider just the symmetric two table scheme.

2.3 Analysis

As in all other analyses of randomized hashing schemes, we assume the *oblivious adversary model*, i.e., that the keys inserted are independent of the random choices made by the algorithm.

Our analysis of the insertion procedure has three main parts:

(1) We first exhibit some useful characteristics of the behavior of the insertion procedure.
(2) We then derive a bound on the probability that the insertion procedure uses at least t iterations.
(3) Finally we argue that the procedure uses expected amortized constant time.

Behavior of the Insertion Procedure

The simplest behavior of the insertion procedure occurs when it does not visit any hash table cell more than once. In this case it simply runs through a

sequence of nestless keys $x_1, x_2, \ldots$ with no repetitions, inserting x_1 in T_1 and moving the remaining keys in the sequence from one table to the other.

If, at some point, the insertion procedure returns to a previously visited cell, the behavior is more complicated, as shown in Figure 2. The key x_i in the first previously visited cell will become nestless for the second time (occurring at positions i and $j > i$ in the sequence) and be put back in its original cell. Subsequently all keys $x_{i-1}, \ldots, x_2$ will be moved back where they were at the start of the insertion (assuming that the maximum number of iterations is not reached). This means that x_1 ends up nestless again, and the procedure will try placing it in the second table. At some point after this there appears a nestless key x_l that is either moved to a vacant cell or a previously visited cell (again assuming that the maximum number of iterations is not reached). In the former case the procedure terminates. In the latter case a rehash must be performed, since we have a "closed loop" of $l - i + 1$ keys hashing to only $l - i$ cells. This means that the loop will run for the maximum number of iterations, followed by a rehash.

Lemma 1 *Suppose that the insertion procedure does not enter a closed loop. Then for any prefix $x_1, x_2, \ldots, x_p$ of the sequence of nestless keys, there must be a subsequence of at least $p/3$ consecutive keys without repetitions, starting with an occurrence of the key x_1, i.e., the key being inserted.*

Proof. In the case where the insertion procedure never returns to a previously visited cell, the prefix itself is a sequence of p distinct nestless keys starting with x_1. Otherwise, the sequence of nestless keys is as shown in Figure 2. If $p < i+j$, the first $j-1 \geq \frac{i+j-1}{2} \geq p/2$ nestless keys form the desired sequence. For $p \geq i + j$, one of the sequences $x_1, \ldots, x_{j-1}$ and $x_{i+j-1}, \ldots, x_p$ must have length at least $p/3$. □

Probability Bounds

We now consider the probability that the insertion loop runs for at least t iterations. For $t >$ MaxLoop the probability is of course 0. Otherwise, using the above analysis, iteration number t may be performed in three (not mutually exclusive) situations:

(1) The hash function family used is not $(1, \text{MaxLoop})$-universal when restricted to the set of keys in the dictionary (including the key being inserted).
(2) The insertion procedure has entered a "closed loop", i.e., x_l in Figure 2 was moved to a previously visited cell, for $l \leq 2t$.
(3) The insertion procedure has processed a sequence of at least $(2t - 1)/3$

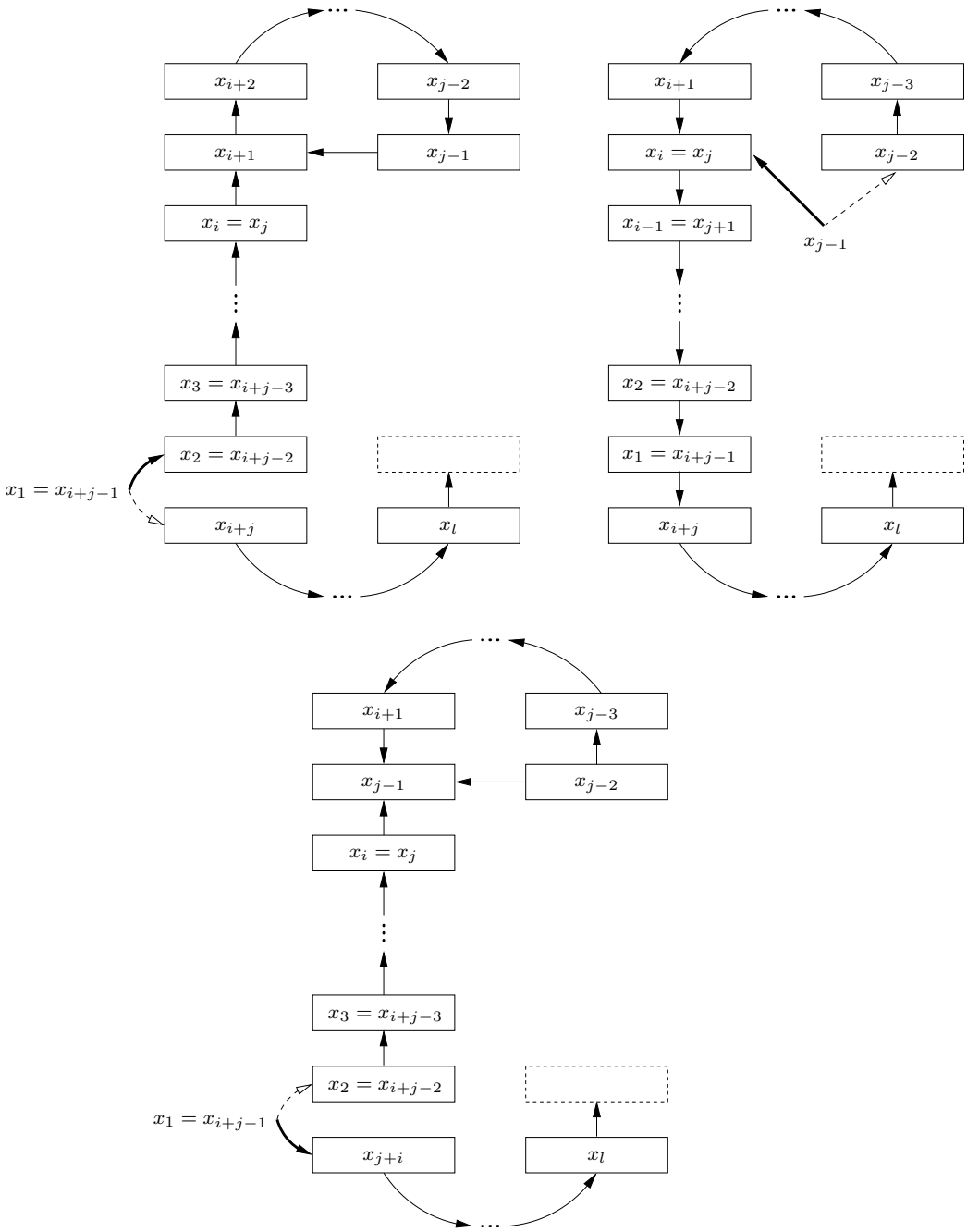

Fig. 2. Three stages of an insertion of key x_1, involving the movement of keys $x_1, \ldots, x_l$. Boxes correspond to cells in either of the two tables, and arcs show possibilities for moving keys. A bold arc shows where the nestless key is to be inserted.

consecutive nestless keys starting with the newly inserted key.

We chose the hash function family such that the first situation occurs with probability $O(1/n^2)$. Under the condition that the first situation does *not* occur, we now bound the probability of the two last situations.

In the second situation let $v \leq l$ denote the number of distinct nestless keys. The number of ways in which the closed loop can be formed is less than $v^3 r^{v-1} n^{v-1}$ (v^2 possible values for i and j, v possible positions for x_l ,r^{v-1} possible choices of cells, and n^{v-1} possible choices of keys other than x_1). Since $v \leq$ MaxLoop, the hash functions are $(1, v)$-universal. This means that each possibility occurs with probability at most r^{-2v}. Summing over all possible values of v, and using $r/n > 1 + \epsilon$, we get that the probability of situation 1 is at most:

$$\sum_{v=3}^{l} v^3 r^{v-1} n^{v-1} r^{-2v} \leq \frac{1}{rn} \sum_{v=3}^{\infty} v^3 (n/r)^v = O(1/n^2) \ .$$

The above derivation follows a suggestion of Sanders and Vöcking [32], and improves the $O(1/n)$ bound in the conference version of this paper [27].

In the third situation there is a sequence of $v = \lceil (2t-1)/3 \rceil$ distinct nestless keys $b_1, \ldots, b_v$, such that b_1 is the key to be inserted, and such that for either $(\beta_1, \beta_2) = (1, 2)$ or $(\beta_1, \beta_2) = (2, 1)$:

$$h_{\beta_1}(b_1) = h_{\beta_1}(b_2),\ h_{\beta_2}(b_2) = h_{\beta_2}(b_3),\ h_{\beta_1}(b_3) = h_{\beta_1}(b_4), \ldots \tag{1}$$

Given b_1 there are at most n^{v-1} possible sequences of v distinct keys. For any such sequence and any of the two choices of (β_1, β_2), the probability that the $b-1$ equations in (1) hold is bounded by $r^{-(v-1)}$, since the hash functions were chosen from a $(1, \text{MaxLoop})$-universal family. Hence the probability that there is *any* sequence of length v satisfying (1), and thus the probability of situation 2, is bounded by

$$2\,(n/r)^{v-1} \leq 2\,(1+\epsilon)^{-(2t-1)/3+1} \ . \tag{2}$$

Concluding the Analysis

From the previous section it follows that the expected number of iterations in the insertion loop is bounded by

$$\begin{aligned} & 1 + \sum_{t=2}^{\text{MaxLoop}} \left(2\,(1+\epsilon)^{-(2t-1)/3+1} + O(1/n^2) \right) \qquad (3) \\ & \leq 1 + O(\tfrac{\text{MaxLoop}}{n^2}) + 2 \sum_{t=0}^{\infty} ((1+\epsilon)^{-2/3})^t \\ & = O(1 + \tfrac{1}{1-(1+\epsilon)^{-2/3}}) \\ & = O(1 + 1/\epsilon) \ . \end{aligned}$$

Finally, we consider the cost of rehashing. First we consider only *forced* rehashes, caused by failed insertions. These occur if the insertion loop runs for $t = \text{MaxLoop}$ iterations. By the previous section, the probability that this happens because of entering a closed loop, or because the hash function family fails to be $(1, \text{MaxLoop})$-universal, is $O(1/n^2)$. Setting $\text{MaxLoop} = \lceil 3 \log_{1+\epsilon} r \rceil$, the probability of rehashing without entering a closed loop is, by (2), at most

$$2\,(1+\epsilon)^{-(2\,\text{MaxLoop}-1)/3+1} = O(1/n^2) \ .$$

Altogether, the probability that any given insertion causes a rehash is $O(1/n^2)$. In particular, the n insertions performed during a rehash all succeed (i.e., cause no further rehash) with probability $1 - O(1/n)$. The expected time used per insertion is $O(1)$, so the total expected time for trying to insert all keys is $O(n)$. If an insertion fails during the rehash, a recursive rehash is started. Since we keep all keys in the tables all the time, this simply corresponds to starting over with another attempt at rehashing all keys. As the probability of having to start over with new hash functions is bounded away from 1, the total expected time for a rehash sums to $O(n)$. Thus, for any insertion the expected time used for forced rehashing is $O(1/n)$.

There will also be a rehash if r^2 insertions have been performed with no failed insertions. Since the expected cost of the rehash is $O(n)$, the amortized expected cost per insertion of such rehashes is $O(1/n)$.

Summing up, we have shown that the amortized expected time for insertion is bounded by a constant. The small probability of rehashing, together with (2), in fact implies that also the *variance* of the insertion time is constant.

3 Background and Related Work on Linear Space Dictionaries

Hashing, first described in public literature by Dumey [13], emerged in the 1950s as a space efficient heuristic for fast retrieval of information in sparse tables. Knuth surveys the most important classical hashing methods in [20, Section 6.4]. The most prominent, and the basis for our experiments in Section 4, are Chained Hashing (with separate chaining), Linear Probing and Double Hashing. Judging from leading textbooks on algorithms, Knuth's selection of algorithms is 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. A more recent scheme called Two-Way Chaining [2] will also be investigated. All schemes are briefly described in Section 4.

3.1 Analysis of early hashing schemes

Early theoretical analysis of hashing schemes was done under the assumption that hash function values are uniformly random and independent. Precise analyses of the average and expected worst case behaviors of the abovementioned schemes have been made, see for example [16,20]. We mention just a few facts, disregarding asymptotically vanishing terms. Note that some figures depend on implementation details – the below hold for the implementations described in Section 4.

We first consider the expected number of memory probes needed by the two "open addressing" schemes to insert a key in a hash table where an α fraction of the table, $0 < \alpha < 1$, is occupied by keys. For Linear Probing the expected number of probes during insertion is $\frac{1}{2}(1 + \frac{1}{(1-\alpha)^2})$. This coincides with the expected number of probes for unsuccessful lookups, and with the number of probes needed for looking up the key if there are no subsequent deletions. A deletion rearranges keys to the configuration that would occur if the deleted key had never been inserted. In Double Hashing the expected cost of an insertion is $\frac{1}{1-\alpha}$. As keys are never moved, this coincides with the number of probes needed for looking up the key and for deleting the key. If a key has not been inserted in the hash table since the last rehash, the expected cost of looking it up (unsuccessfully) is $\frac{1}{1-\beta}$, where β is the fraction of keys and "deleted" markers in the hash table. If the key still has a "deleted" marker in the table, the expected cost of the unsuccessful lookup is one probe more.

For Chained Hashing with hash table size n/α, the expected length of the list traversed during an unsuccessful lookup is α. This means that the expected number of probes needed to insert a new key is $1 + \alpha$, which will also be the number of probes needed to subsequently look up the key (note that probes to pointers are not counted). A deletion results in the data structure that would occur if the key had never been inserted.

In terms of expected number of *probes*, the above implies that, for any given α, Chained Hashing is better than Double Hashing, which is again better than Linear Probing. It should be noted, however, that the space used by Chained Hashing is larger than that in the open addressing schemes for the same α. The difference depends on the relative sizes of keys and pointers.

Suppose $\alpha < 1$ is a constant. The *longest* probe sequence in Linear Probing is then of expected length $\Omega(\log n)$. For Double Hashing the longest successful probe sequence is expected to be of length $\Omega(\log n)$, and there is a nonzero probability that the length of the longest unsuccessful search is linear. The expected maximum chain length in Chained Hashing is $\Theta(\log n / \log \log n)$.

Though the above results seem to agree with practice, the randomness as-

sumptions used for the analyses are questionable in applications. Carter and Wegman [7] succeeded in removing such assumptions from the analysis of Chained Hashing, 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). Using the hash function family of Siegel [35], also used in this paper, Linear Probing and Double Hashing provably satisfy the above performance bounds [33,34].

3.2 Key rearrangement schemes

A number of (open addressing) hashing schemes have been proposed that share a key feature with Cuckoo Hashing, namely that keys are moved around during insertions [4,17,21,22,31]. The main focus in these schemes is to reduce the average number of probes needed for finding a key in a (nearly) full table to a constant, rather than the $O(\log n)$ average exhibited by standard open addressing. This is done by occasionally moving keys forward in their probe sequences.

Our new algorithm rearranges keys in order to reduce the *worst case* number of probes to a constant. A necessary condition for this is reuse of hash function values, i.e., that keys are moved back in their probe sequence. Backward moves were not used in any previous rearrangement scheme, presumably due to the difficulty that moving keys back does not give a fresh, "random" placement. We can make lookups use constant time in the worst case because we do not deal with full hash tables, but rather hash tables having a constant fraction of unoccupied cells.

Arrangements of keys with optimal worst case retrieval cost were in fact already considered by Rivest in [31]. He assumes that the probe sequences are given, and presents a polynomial time algorithm for finding an arrangement that minimizes the length of the longest successful search. It is also shown that if one updates the key set, the expected number of keys that need to be moved to achieve a new optimal arrangement is constant. (The analysis requires that the hash table is sufficiently sparse, and assumes the hash function to be truly random.) This suggests a dictionary that solves a small assignment problem after each insertion and deletion. It follows from [26] and this paper, that Rivest's dictionary achieved worst case constant lookup time and expected amortized constant update time, 8 years before an algorithm with the same performance and randomness assumption was published by Aho and Lee [1]. Furthermore, Siegel's hash functions suffice for the analysis. However, the Cuckoo Hashing insertion algorithm is much simpler and more efficient

than that suggested by Rivest.

Another key rearrangement scheme with similarities to Cuckoo Hashing is Last-come-first-served Hashing [29], which has low variance on search time as its key feature. It uses the same greedy strategy for moving keys as is used in this paper, but there is no reuse of hash function values.

3.3 Hashing schemes with worst case lookup guarantee

Two-Way Chaining [2] is an alternative to Chained Hashing that offers $O(\log \log n)$ maximal lookup time with high probability (assuming truly random hash functions). This scheme shares the feature with Cuckoo Hashing that keys are stored in one of two places (in this case linked lists). The implementation that we consider represents the lists by fixed size arrays of size $O(\log \log n)$ (if a longer chain is needed, a rehash is performed). To achieve linear space usage, one must then use a hash table of size $O(n/\log \log n)$, implying that the *average* chain length is $\Omega(\log \log n)$ [3]. (We remark that the idea of storing keys in one out of two places was used even earlier by Karp, Luby, and Meyer af der Heide [18] in the context of PRAM simulation.)

Another scheme with this worst case guarantee is *Multilevel Adaptive Hashing* [5]. However, lookups can be performed in $O(1)$ worst case time if $O(\log \log n)$ hash function evaluations, memory probes and comparisons are possible in parallel. This is similar to Cuckoo Hashing, though the latter uses only *two* hash function evaluations, memory probes, and comparisons.

A dictionary with worst case *constant* lookup time was first obtained by Fredman, Komlós and Szemerédi [15], 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. [10]. Dietzfelbinger and Meyer auf der Heide [11] improved the update performance by exhibiting a dictionary in which operations are done in constant time with high probability, namely at least $1 - n^{-c}$, where c is any constant of our choice. A simpler dictionary with the same properties was later developed [8]. When $n = |U|^{1-o(1)}$ a space usage of $O(n)$ words is not within a constant factor of the information theoretical minimum of $B = \log \binom{|U|}{n}$ bits. The dictionary of Raman and Rao [30] offers the same performance as [10], using $B + o(B)$ bits in all cases. However, it does not support information associated with keys.

Very recently, Fotakis et al. [14] analyzed a generalization of Cuckoo Hashing with d possible locations for each key, showing that in this case a space utilization of $1 - 2^{-\Omega(d)}$ can be achieved, with constant expected time for insertions.

4 Experiments

To examine the practicality of Cuckoo Hashing we experimentally compare it to three well known hashing methods, as described in [20, Section 6.4]: Chained Hashing (with separate chaining), Linear Probing and Double Hashing. We also consider Two-Way Chaining [2].

The first three methods all attempt to store a key x at position $h(x)$ in a hash table. They differ in the way collisions are resolved, i.e., in what happens when two or more keys hash to the same location.

Chained Hashing. A linked list is used to store all keys hashing to a given location.

Linear Probing. A key is stored in the next empty table entry. Lookup of key x is done by scanning the table beginning at $h(x)$ and ending when either x or an empty table entry is found. When deleting, some keys may have to be moved back in order to fill the hole in the lookup sequence, see [20, Algorithm R] for details.

Double Hashing. Insertion and lookup are similar to Linear Probing, but instead of searching for the next position one step at a time, a second hash function value is used to determine the step size. Deletions are handled by putting a special "deleted" marker in the cell of the deleted key. Lookups skip over deleted cells, while insertions overwrite them.

The fourth method, Two-Way Chaining, can be described as two instances of Chained Hashing. A key is inserted in one of the two hash tables, namely the one where it hashes to the shorter chain. A cache-friendly implementation, as recently suggested in [6], is to simply make each linked list a short, fixed size array. If a longer list is needed, a rehash must be performed.

4.1 Previous Experimental Results

Although the dictionaries with worst case constant lookup time surveyed in Section 3 leave little to improve from a theoretical point of view, large constant factors and complicated implementation hinder their direct practical use. For example, in the "dynamic perfect hashing" scheme of [10] the upper bound on space is $35n$ words. The authors of [10] refer to a more practical variant due to Wenzel that uses space comparable to that of binary search trees.

According to [19] the implementation of this variant in the LEDA library [25], described in [39], 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. (On a Linux PC with an Intel® Pentium® 120 MHz processor.) The experi-

mental results listed in [25, 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. (On a 300 MHz SUN ULTRA SPARC.)

Silverstein [36] reports that the space upper bound of the dynamic perfect hashing scheme of [10] is quite pessimistic compared to what can be observed when run on a subset of the DIMACS dictionary tests [24]. He goes on to explore ways of improving space as well as time, 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 an implementation of linear probing to achieve similar time per operation. Silverstein also considers versions of the data structures with packed representations of the hash tables. In this setting the dynamic perfect hashing scheme was more than 50% slower than linear probing, using roughly the same amount of space.

Is seems that recent experimental work on "classical" dictionaries (that do not have worst case constant lookup time) is quite limited. In [19] it is reported that chained hashing is superior to an implementation of dynamic perfect hashing in terms of both memory usage and speed.

4.2 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. During rehashes this is known not to occur and slightly faster versions of the insertion procedure are used.

Our focus is on minimizing the time for dictionary operations under the constraint that space usage should be reasonable. By the *load factor* of a dictionary we will understand the size of the set relative to the memory used. (For Chained Hashing, the notion of load factor traditionally disregards the space used for linked lists, but we desire equal load factors to imply equal memory usage.) As seen in [20, Figure 44] the speed of Linear Probing and Double Hashing degrades rapidly for load factors above 1/2. On the other hand, none of the schemes improve much for load factors below 1/4. 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, causing 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 impractical to realize. However, experience shows that rather simple and efficient hash function families yield performance close to that predicted under stronger randomness assumptions. We use a function family from [9] with range $\{0,1\}^q$ for positive integer q. For every odd a, $0 < a < 2^w$, the family contains the function $h_a(x) = (ax \bmod 2^w) \operatorname{div} 2^{w-q}$. Note that evaluation can be done very efficiently by a 32 bit multiplication and a shift. However, this choice of hash function restricts us to consider hash tables whose sizes are powers of two. A random function from the family (chosen using C's `rand` function) appears to work fine with all schemes except Cuckoo Hashing. For Cuckoo Hashing we experimented with various hash functions and found that Cuckoo Hashing was rather sensitive to the choice of hash function. It turned out that the exclusive or of three independently chosen functions from the family of [9] was fast and worked well. We have no good explanation for this phenomenon. For all schemes, various alternative hash 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. Specific choices made and details differing from the references are:

Chained Hashing. C's `malloc` and `free` functions were found to be a performance bottleneck, so a simple "freelist" memory allocation scheme is used. Half of 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. We store the first key of each linked list directly in the hash table, as this often saves one cache miss. Having the first key in the hash table also slightly improves memory utilization, in the expected sense. This is because every non-empty linked list is one element shorter and because we expect more than half of the hash table cells to contain a linked list for the load factors considered here.

Double Hashing. To prevent the tables from clogging up with deleted cells, resulting in poor performance for unsuccessful lookups, all keys are rehashed when 2/3 of the hash table is occupied by keys and "deleted" markers. The fraction 2/3 was found to give a good tradeoff between the time for insertion and unsuccessful lookups.

Linear Probing. Our first implementation, like that in [36], employed deletion markers. However, we found that using the deletion method described in [20, Algorithm R] was considerably faster, as far fewer rehashes were needed.

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 [6]. For larger collections of keys one should allow more keys in each bucket, resulting in general performance degradation.

Cuckoo Hashing. The architecture on which we experimented could not parallelize the two memory accesses in lookups. Therefore we only evaluate the second hash function after the first memory lookup has shown unsuccessful.

For all schemes, rehashing was implemented as repeated insertion of all keys into a newly allocated hash table. For efficiency we used special insertion procedures without a check of whether keys were already inserted.

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 was no significant slowdown for other operations. We will describe the results for Asymmetric Cuckoo when they differ significantly from those of Cuckoo Hashing.

4.3 Setup

Our experiments were performed 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. Our results nicely fit a simple model parameterized by the cost of a cache miss and the expected number of probes to "random" locations (see the technical report [28] for details). They are thus believed to have significance for other hardware 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 algorithms, and allows us to discard the time used by the program issuing the calls to the Cuckoo Hashing data structure. 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 on a dictionary operation 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 machine with no irrelevant user processes, so disturbances should be minimal. On our machine it took 32 clock cycles to call the `rdtsc` instruction. These clock cycles have been subtracted from the results.

4.4 Results

Our main experiment 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 Internet [23]. 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 Figures 3 and 4, which show an average over 10 runs. We ran experiments with up to $2^{24}/3$ keys. 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. The results had a large variance, over the 10 runs, for sets of size 2^{12} to 2^{16}. Outside this range the extreme values deviated from the average by less than about 7%. The large variance sets in when the data structure starts to fill the level 2 cache. We believe this is caused by our test program reading data from disk and thus sometimes evicting parts of the data structure from cache.

As can be seen, the time for lookups is almost identical for all schemes as long as the entire data structure fits in level 2 cache, i.e., for $n < 2^{16}/3$. After this the average number of accesses to a random memory cell (with the probability of a cache miss approaching 1) shows up. The shape of the curves reflect the increasing probability of a cache miss for an access to a random memory cell (see Section 5 of the technical report [28] for details). This makes linear probing an average case winner, with Cuckoo Hashing and Two-Way Chaining following about 40 clock cycles behind. For insertion the number of accesses to a random memory cell 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, slower for small sets. The cost of forced rehashes sets in for Two-Way Chaining for sets of more than

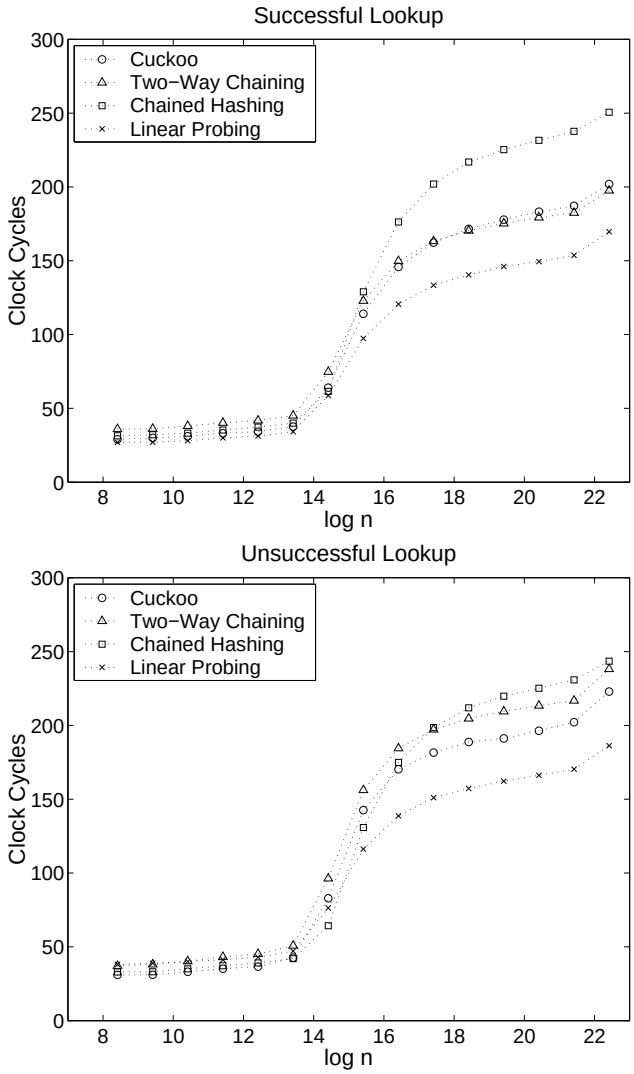

Fig. 3. The average time per lookup operation in equilibrium for load factor 1/3.

a million elements, at which point better results may have been obtained by a larger bucket size. For deletion Chained Hashing lags behind for large sets due to accesses to a random memory cell when freeing list elements, while the simplicity of Cuckoo Hashing makes it the fastest scheme. We note that, for dictionaries that fit in cache, the total time for an insertion and a deletion is smallest for Cuckoo Hashing among the four schemes.

At this point we should mention that the good cache utilization of Linear Probing and Two-Way Chaining depends on the cache lines being considerably larger than keys (and any associated information placed together with keys). If this is not the case, it causes the number of cache misses to rise significantly. The other schemes discussed here do not deteriorate in this way.

We made additional experiments concerning the cost of insertions in growing dictionaries and deletions in shrinking dictionaries, which takes into account

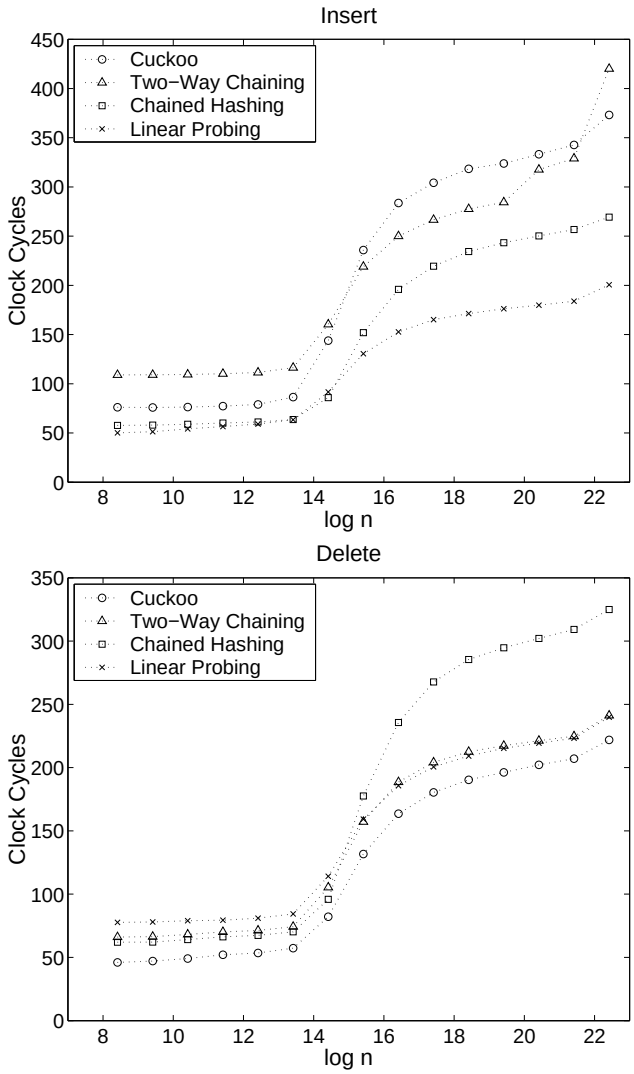

Fig. 4. The average time per update operation in equilibrium for load factor 1/3.

the cost of rehashes needed to keep space utilization around 1/3. The interested reader can find the results of these tests in the technical report [28].

DIMACS Tests

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 deletion of short-lived keys. 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 [24]. The tests involving string keys were preprocessed by hashing strings to 32 bit integers, as described in

	Joyce		Eddington	
Linear	42 - 45	(.35)	26 - 27	(.40)
Double	48 - 53	(.35)	32 - 35	(.40)
Chained	49 - 52	(.31)	36 - 38	(.28)
A.Cuckoo	47 - 50	(.33)	37 - 39	(.32)
Cuckoo	57 - 63	(.35)	41 - 45	(.40)
Two-Way	82 - 84	(.34)	51 - 53	(.40)

Fig. 5. Average clock cycles per operation and load factors for two DIMACS string tests.

	3.11-Q-1		Smalltalk-2		3.2-Y-1	
Linear	99 - 103	(.30)	68 - 72	(.29)	85 - 88	(.32)
Double	116 - 142	(.30)	77 - 79	(.29)	98 - 102	(.32)
Chained	113 - 121	(.30)	78 - 82	(.29)	90 - 93	(.31)
A.Cuckoo	166 - 168	(.29)	87 - 95	(.29)	95 - 96	(.32)
Cuckoo	139 - 143	(.30)	90 - 96	(.29)	104 - 108	(.32)
Two-Way	159 - 199	(.30)	111 - 113	(.29)	133 - 138	(.32)

Fig. 6. Average clock cycles per operation and load factors for three DIMACS integer tests.

Appendix A. This preserves, with high probability, the access pattern to keys. For each test we recorded the average time per operation, not including the time used for preprocessing. The minimum and maximum of six runs can be found in Tables 5 and 6, which also lists the average load factor. Linear probing is again the fastest, but mostly just 20-30% faster than the Cuckoo schemes.

The Number of Cache Misses During Insertion

We have seen that the number of accesses to a random memory cell (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 assumptions), the analysis of the expected time for insertions in Section 2.3 is rather crude, establishing just a constant upper bound. One reason that our calculation does not give a very tight bound is that we use a pessimistic estimate on the number of key moves needed to accommodate a new element in the dictionary. Often a free cell will be found even though it *could* have been occupied by another key in the dictionary. We

also pessimistically assume that a large fraction of key moves will be spent backtracking from an unsuccessful attempt to place the new key in the first table.

Figure 7 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. We believe 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. The curve for Cuckoo Hashing seems to be $2 + 1/(4 + 8\alpha) \approx 2 + 1/(4\epsilon)$. This is in good correspondence with (3) of the analysis in Section 2.3. It should be remarked that the highest possible load factor for Two-Way Chaining is $O(1/\log\log n)$.

As noted in Section 2, the insertion algorithm of Cuckoo Hashing is biased towards inserting keys in T_1. If we instead of starting the insertion in T_1 choose the start table at random, the number of cache misses decreases slightly for insertion. This is because the number of free cells in T_1 increases as the load balance becomes even. However, this also means a slight increase in lookup time. Also note that since insertion checks if the element is already inserted, Cuckoo Hashing uses at least two cache misses. The initial lookup can be exploited to get a small improvement in insertion performance, by inserting right away when *either* cell $T_1[h_1(x)]$ or $T_2[h_2(x)]$ is vacant. For load factor 1/3 this places about 10% of newly inserted keys in T_2. The relatively low percentage is the reason why we found no advantage in performing the extra check in our implementation.

Since lookup is very similar to insertion in Chained Hashing, one could think that the number of cache misses would be equal for the two operations. However, in our implementation, obtaining a free cell from the freelist may result in an extra cache miss. This is the reason why the curve for Chained Hashing in the figure differs from a similar plot in Knuth [20, Figure 44].

5 Conclusion

We have presented a new dictionary with worst case constant lookup time. It 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 worse average case performance. Several challenges remain. First of all an explicit, truly practi-

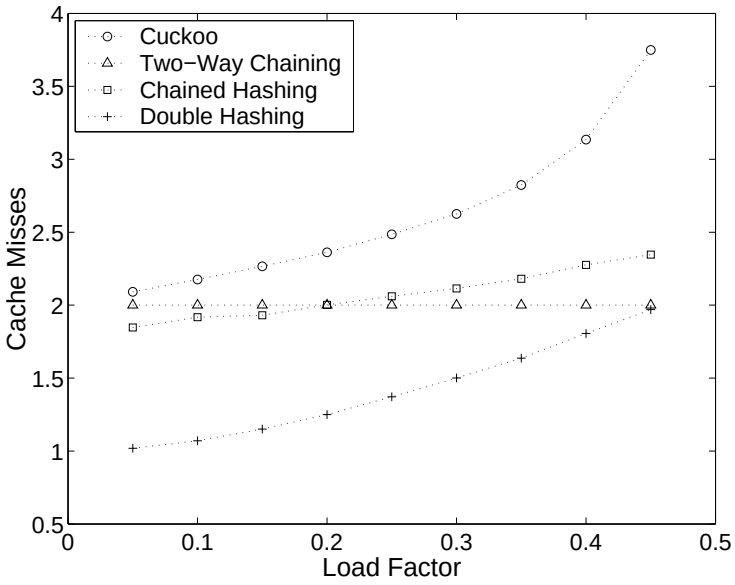

Fig. 7. The average number of accesses to a random memory cell for insertion.

cal hash function family that is provably good for the scheme has yet to be found. One step in this direction was recently taken by Dietzfelbinger and Woelfel [12], but their hash functions still require a relatively large amount of space. Secondly, we lack a precise understanding of why the scheme exhibits low constant factors. In particular, the curve of Figure 7 needs to be explained.

Acknowledgements. The authors would like to thank Andrei Broder, Martin Dietzfelbinger, Rolf Fagerberg, Peter Sanders, John Tromp, and Berthold Vöcking for useful comments and discussions on this paper and Cuckoo Hashing in general.

References

[1] Alfred V. Aho and David Lee. Storing a dynamic sparse table. In *Proceedings of the 27th Annual Symposium on Foundations of Computer Science (FOCS '86)*, pages 55–60. IEEE Comput. Soc. Press, 1986.

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

[3] Petra Berenbrink, Artur Czumaj, Angelika Steger, and Berthold Vöcking. Balanced allocations: the heavily loaded case. In *Proceedings of the 32nd Annual ACM Symposium on Theory of Computing (STOC '00)*, pages 745–754. ACM Press, 2000.

[4] Richard P. Brent. Reducing the retrieval time of scatter storage techniques. *Communications of the ACM*, 16(2):105–109, 1973.

[5] Andrei Z. Broder and Anna R. Karlin. Multilevel adaptive hashing. In *Proceedings of the 1st Annual ACM-SIAM Symposium on Discrete Algorithms (SODA '90)*, pages 43–53. ACM Press, 1990.

[6] Andrei Z. Broder and Michael Mitzenmacher. Using multiple hash functions to improve IP lookups. In *Proceedings of the Twentieth Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM 2001)*, volume 3, pages 1454–1463. IEEE Comput. Soc. Press, 2001.

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

[8] Martin Dietzfelbinger, Joseph Gil, Yossi Matias, and Nicholas Pippenger. Polynomial hash functions are reliable (extended abstract). In *Proceedings of the 19th International Colloquium on Automata, Languages and Programming (ICALP '92)*, volume 623 of *Lecture Notes in Computer Science*, pages 235–246. Springer-Verlag, 1992.

[9] 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.

[10] 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.

[11] Martin Dietzfelbinger and Friedhelm Meyer auf der Heide. A new universal class of hash functions and dynamic hashing in real time. In *Proceedings of the 17th International Colloquium on Automata, Languages and Programming (ICALP '90)*, volume 443 of *Lecture Notes in Computer Science*, pages 6–19. Springer-Verlag, 1990.

[12] Martin Dietzfelbinger and Philipp Woelfel. Almost random graphs with simple hash functions. In *Proceedings of the 35th Annual ACM Symposium on Theory of Computing (STOC '03)*, pages 629–638, 2003.

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

[14] Dimitris Fotakis, Rasmus Pagh, Peter Sanders, and Paul Spirakis. Space efficient hash tables with worst case constant access time. In *Proceedings of the 20th Symposium on Theoretical Aspects of Computer Science (STACS '03)*, volume 2607 of *Lecture Notes in Computer Science*, pages 271–282. Springer-Verlag, 2003.

[15] 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.

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

[17] Gaston H. Gonnet and J. Ian Munro. Efficient ordering of hash tables. *SIAM J. Comput.*, 8(3):463–478, 1979.

[18] Richard M. Karp, Michael Luby, and Friedhelm Meyer auf der Heide. Efficient PRAM simulation on a distributed memory machine. *Algorithmica*, 16(4-5):517–542, 1996.

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

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

[21] J. A. T. Maddison. Fast lookup in hash tables with direct rehashing. *The Computer Journal*, 23(2):188–189, May 1980.

[22] Efrem G. Mallach. Scatter storage techniques: A uniform viewpoint and a method for reducing retrieval times. *The Computer Journal*, 20(2):137–140, May 1977.

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

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

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

[26] Rasmus Pagh. On the Cell Probe Complexity of Membership and Perfect Hashing. In *Proceedings of the 33rd Annual ACM Symposium on Theory of Computing (STOC '01)*, pages 425–432. ACM Press, 2001.

[27] Rasmus Pagh and Flemming Friche Rodler. Cuckoo hashing. In *Proceedings of the 9th European Symposium on Algorithms (ESA '01)*, volume 2161 of *Lecture Notes in Computer Science*, pages 121–133. Springer-Verlag, 2001.

[28] Rasmus Pagh and Flemming Friche Rodler. Cuckoo hashing. Research Series RS-01-32, BRICS, Department of Computer Science, University of Aarhus, August 2001. 21 pp.

[29] Patricio V. Poblete and J. Ian Munro. Last-come-first-served hashing. *J. Algorithms*, 10(2):228–248, 1989.

[30] Rajeev Raman and S. Srinivasa Rao. Succinct dynamic dictionaries and trees. In *Proceedings of the 30th International Colloquium on Automata, Languages and Programming (ICALP '03)*, volume 2719 of *Lecture Notes in Computer Science*, pages 345–356. Springer-Verlag, 2003.

[31] Ronald L. Rivest. Optimal arrangement of keys in a hash table. *J. Assoc. Comput. Mach.*, 25(2):200–209, 1978.

[32] Peter Sanders and Berthold Vöcking, 2001. Personal communication.

[33] Jeanette P. Schmidt and Alan Siegel. On aspects of universality and performance for closed hashing (extended abstract). In *Proceedings of the 21st Annual ACM Symposium on Theory of Computing (STOC '89)*, pages 355–366. ACM Press, 1989.

[34] Jeanette P. Schmidt and Alan Siegel. The analysis of closed hashing under limited randomness (extended abstract). In *Proceedings of the 22nd Annual ACM Symposium on Theory of Computing (STOC '90)*, pages 224–234. ACM Press, 1990.

[35] Alan Siegel. On universal classes of fast high performance hash functions, their time-space tradeoff, and their applications. In *Proceedings of the 30th Annual Symposium on Foundations of Computer Science (FOCS '89)*, pages 20–25. IEEE Comput. Soc. Press, 1989.

[36] Craig Silverstein. A practical perfect hashing algorithm. In *Data Structures, Near Neighbor Searches, and Methodology: Fifth and Sixth DIMACS Implementation Challenges*, volume 59 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 23–48. American Mathematical Society, 2002.

[37] Mikkel Thorup. Even strongly universal hashing is pretty fast. In *Proceedings of the 11th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA '00)*, pages 496–497. ACM Press, 2000.

[38] John Tromp, 2003. Personal communication.

[39] Michael Wenzel. Wörterbücher für ein beschränktes Universum. Diplomarbeit, Fachbereich Informatik, Universität des Saarlandes, 1992.

A Constructions and properties of universal hash functions

A.1 Universal hash function families

As a simple example of a universal family, the family of all functions from U to some codomain is $(1, |U|)$-universal. However, for implementation purposes one needs families with much more succinct memory representations. A standard construction of a $(2, k)$-universal family for range $R = \{0, \ldots, r-1\}$ and prime $p > \max(2^w, r)$ is

$$\{x \mapsto ((\sum_{l=0}^{k-1} a_l x^l) \bmod p) \bmod r \quad | \quad 0 \leq a_0, a_1, \ldots, a_{k-1} < p\} \ . \tag{A.1}$$

This paper uses a hash function construction due to Siegel [35] that has *constant* evaluation time (however, the constant is not small). Its properties are captured by the following theorem, which can be derived from Siegel's paper by using a universe collapse function, as described below.

Theorem 1 (Siegel) *Let γ and $\delta > 0$ be constants, and take any set $X \subseteq U$. Using space and initialization time $O(|X|^\delta)$ it is possible to construct a family of functions such that, for some constant $\delta' > 0$:*

- *With probability at least $1 - |X|^{-\gamma}$ the family is $(1, |X|^{\delta'})$-universal when restricted to X.*
- *Furthermore, functions from the family can be evaluated in* constant *time, and a random function can be picked using time and space $O(|X|^\delta)$.*

A.2 Collapsing the universe

The restriction that keys are single words is not a serious one, as longer keys can be handled using the standard technique of *collapsing* the universe. Specifically, long keys can be mapped to keys of $O(1)$ words by applying a random function ρ from a $(O(1), 2)$-universal family. There is such a family whose functions can be evaluated in time linear in the number of words in a key [7]. It works by evaluating a function from a $(O(1), 2)$-universal family on each word of the key, computing the bitwise exclusive or of the function values. (See [37] for an efficient implementation.) Such a function ρ with range $\{0,1\}^{2\log(n)+c}$ will, with probability $1 - O(2^{-c})$, be injective on S. In fact, with constant probability ρ is injective on a given *sequence* of $\Omega(2^{c/2}n)$ consecutive sets in a dictionary of initial size n (see [10]). When a collision for ρ between two elements of S is detected in the dictionary, everything is rehashed, i.e., ρ is chosen anew and the whole data structure is rebuilt. If a rehash can be done in expected $O(n)$ time, the amortized expected cost of this is $O(2^{-c/2})$ per insertion. In this way we can effectively reduce the universe size to $O(n^2)$, though the full keys still need to be stored to decide membership.