Experiments with cuckoo hashing

Fredrik Bengtsson
Division of Computer Science and Networking
Department of Computer Science and Electrical Engineering
Luleå University of Technology

Abstract— **In this paper, a slight generalization of the cuckoo hashing scheme, first introduced by R. Pagh and F. F. Rodler, is presented, along with experimental results that shows that an increased load factor above 0.5 is possible in exchange for longer, but constant, lookup time. Several other improvements are also introduced as well as a few generalizations that does not lead to a better scheme.**

I. Introduction

The cuckoo hashing scheme, first introduced by Pagh and Rodler [PR01], has constant worst-case lookup and delete time and constant expected insertion time. Together with the fact that the scheme is very simple, this makes cuckoo hashing a very attractive hashing scheme in many applications.

This paper introduces a slight generalization and improvement over the original cuckoo hashing scheme. The improvements are verified by experiments (simulations), with a test implementation of the cuckoo hashing scheme.

In this section, hashing in general and cuckoo hashing in particular will be introduced. In Section II, the cuckoo hashing generalizations made will be introduced together with discussions of the experimental results. Section III concludes the paper and presents open problems.

A. Background

Hashing in general and cuckoo hashing in particular solves a problem commonly referred to as *the search problem*. The search problem is the problem of, given *key*, finding a pair, $(key, data)$, in a set of pairs of the same type.

There are two major approaches for solving the search problem; hashing based algorithms and tree based algorithms. Cormen, Leiserson, Rivest [CLR89] has an introduction to both. Tree-based algorithms requires a total order on the set of key and hashing requires the ability to perform arithmetic operations on the keys.

Sometimes, hashing based algorithms solve a special case of the search problem called the *dictionary problem*; in this case, the keys are required to be unique. The field *data* in the pair $(key, data)$ is often ignored in the design and analysis of search algorithms, because it is, in many algorithms, trivial to add the capability to handle this data (you can just store the data, or a reference to the data, together with the key).

The hashing approach for solving the search problem uses a *hash function* to map the universe of keys to the universe of positions in a *hash table*. The number of positions in the hash table is smaller than the universe of keys and larger than the number of keys that is supposed to be stored in the table. The keys are stored in the table at the positions generated by applying the hash function to the key. Since the number of positions in the table is smaller than the universe of keys, there can be collisions (two keys can map to the same position). There are several techniques for solving this problem and cuckoo hashing is one of them. We use the term *basic hash table* to refer to this hash table (without collision resolution).

B. The cuckoo hashing scheme

The cuckoo hashing scheme uses two basic hash tables. An element is inserted into the cuckoo hash table by inserting the element into the first basic hash table. If a collision occurs when inserting the element into the basic hash table, the old element in that position is removed and the new element is inserted. The removed element is by Pagh and Rodler termed the "kicked-out" element. The "kicked-out" element is then inserted into the second basic hash table. If the position in the second basic hash table was empty prior to the insertion of the "kicked-out" element, we are done. If not, the collision in the second basic hash table is handled in the same way as the collision in the first basic hash table: The old element from the second basic hash table is "kicked-out". This element is then inserted into the first basic hash table. Here, it is important to observe that the second "kicked-out" element does not, in general, hash to the same position in the first basic hash table as the original element to be inserted, and the first "kicked-out" element, did, because the second "kicked-out" element and the first "kicked-out" element are different elements and can, therefore, be expected to hash to different positions in the basic hash table. If, again, there is a collision (with another element) in the first basic hash table, the procedure is repeated until no collision occurs or until a preset number of collisions have occurred (determined by the collision limit function). When this happens, all elements are rehashed into a new cuckoo hash table. With this scheme, several (ultimately all) positions in both basic tables are probed for an empty position.

At this point, we observe that a given element can only be placed in one unique position in each basic hash table (determined by the hash function). This fact makes searching the table trivially constant time, because there is only one position to be probed in each table, and there are a constant number of tables.

The cuckoo hashing scheme, as described above, is a *static* scheme, which means that the size of the data struc-

ture (the cuckoo hash table in this case) cannot change. It is possible to make the scheme *dynamic* by creating a new, larger, table when the table becomes full and rehashing (inserting into the new table) all the elements into the new table.

Pagh and Rodler claims that the cuckoo hashing scheme is only able to handle a load factor of at most 0.5, which means the table can only be filled half full.

II. Generalizations and Simulations

This section will introduce the cuckoo hashing generalizations and discuss simulation results that is the basis for the conclusions of this report. The following generalizations/changes have been made to the original cuckoo hashing scheme:

- The use of more than two basic hash tables.
- The use of more than one hash function per basic hash table.
- Several strategies for changing the size of a cuckoo hash table.
- Several strategies for inserting an element into the cuckoo hash table.
- Several functions for determining the maximum collision limit.

All simulations were done on a test implementation of the cuckoo scheme made in Java (JDK 1.3.1). The implementation is not intended to be a high performance implementation, but to be a basis for extensions and generalizations of the original scheme by allowing easy implementation of new features. The implementation is available upon request.

The performance measure used in this report is the number of table updates performed. One table update is defined as the operation of assigning a slot in a table a new value. Since both the search and the delete operations are constant time by construction of the scheme, only the insert operation is studied in the experiments. This performance measure makes the simulations independent of machine, operating system and programming languages, because the number of table updates is only dependent on the algorithm.

Uniformly distributed random values are used for the test data and all random values, including the random values needed for the generation of hash functions, are extracted from George Marsaglias random bits collection [Mar].

The hash functions used are the exclusive or of three functions selected from the family $h(x) = (ax \bmod 2^w) \operatorname{div} 2^{w-q}$, as suggested by Pagh and Rodler. Here, a is the random number determining the function and w is the machine word length. The table size is 2^q. The main reason for choosing this hash function is that it can be evaluated very efficiently in a real implementation. If the multiplication ax is done with the precision of the machine word length, the mod operation becomes implicit. The div operation can be implemented with a shift operation.

A. Number of tables and functions

It is possible to increase the number of used tables above two, which is the number used in the original scheme. In this setup, we are required to perform a lookup in each table in order to find the element searched for. This will necessarily increase the lookup time. However, it may be possible to fill the tables more than half full as suggested for the original scheme.

To see this, consider the extreme case where we have equally many tables as elements to insert. In such case, the algorithm degenerates to an unsorted array; Each basic hash table has only one position and each hash function is the constant function that maps all keys to that position. When searching, it is necessary to linear search all the basic hash tables (equivalent to linear searching an array), which requires $O(n)$ time. Insertion can be done by inserting the element into any of the basic tables that does not yet contain a key. This way, it is possible to fill the cuckoo table completely. There may be a trade-off between the search time and the lookup time: By increasing the number of basic tables in the cuckoo table, it may be possible to fill the cuckoo table more than half full.

With this as background, we hope that the increased number of tables will allow us to utilize a larger load factor.

Another possibility is not only to use more tables, but to use several hash functions per table. In such case, each hash function is treated as a separate basic table and the lookup will have to test all functions in all tables, in the worst case, in order to find the element searched for.

A problem arises when using several functions for one table: Since there are many hash function for each basic table, we have lost information about which hash function was used to insert the elements into the table. This can be a problem when a collision occurs during insertion; we have to make sure that the same hash function is not used again for the "kicked out" element when it is inserted into the same table but with a different hash function. It is, however, possible to check which function was used by evaluating the functions on the element and see if it returns the same position. The drawback of this is, of course, that the number of hash function evaluations are increased. Another possible resolution for the problem is to store extra information about the function used associated with each position. This will, of course, require extra memory.

A.1 Simulation setup

In this test, the number of tables and hash functions per table is varied while the total table size is constant. The following cases have been tested:

- 2 tables and 1 function
- 4 tables and 1 function
- 8 tables and 1 function
- 2 tables and 2 function
- 4 tables and 2 function

More than 8 tables have not been tested, because the lookup time would be long and, as we will see, more than 4 tables will not give much improvement in the possible usable load factor.

The tests are performed by filling an initially empty table of fixed size up until the maximum load factor possible (approximately) for the particular configuration used. The result presented represents a mean value of 10000 operations created by applying 10000 insert-delete operations with elements from two pre-generated lists for insertions and deletions, respectively. That is, successively, a value from the insert-list is inserted and then a value from the delete-list is deleted until the insert and delete lists are empty. The delete operation is not counted. The operation count is then divided by the number of elements in the lists (10000). This way, it is possible to see how much an insert operation pays, at average, for the rehashes sometimes required. After this is performed, the table is filled up until the next sample point is reached and the process is repeated.

The tests were performed using three different table sizes: 2^{10}, 2^{15} and 2^{20} elements. Since the insertion time, according to Pagh and Rodler, should not be dependent on the number of elements in the table but only (possibly) on the load factor, the results should be almost identical for the three cases. The simulation verifies this claim and, therefore, only the results for 2^{20} elements are presented here.

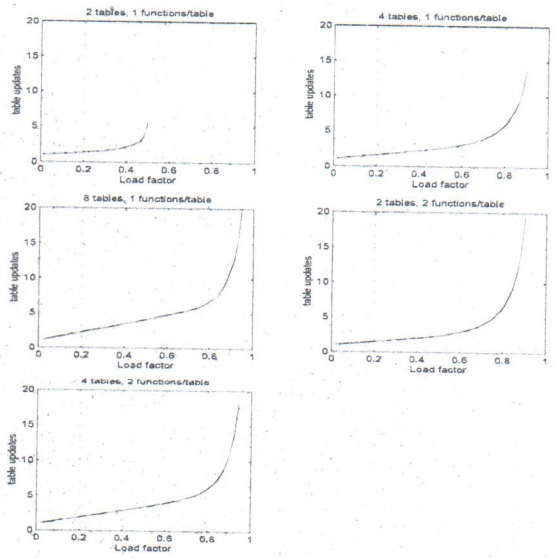

Fig. 1. Insertion of elements into a 2^{20} elements table.

The simulation plots (Fig. II-A.1) shows that with two tables, a load factor of at most 0.5 gives acceptable insertion cost (this is the original cuckoo hashing).

With four four tables, we observe that the load factor can be increased to 0.8 with an average of 6 table updates/insertion. At a load factor of 0.9, 14 table updates are required per insertion. This is an improvement over the case with two tables. With eight tables, a load factor of 0.9 requires 11 updates per insertion and a load factor of 0.95 requires approx. 20 updates per insertion.

The lookup time for four tables respective eight tables are two respective four times the time for a lookup with two tables, because of the fact that it is necessary to probe all tables (in the worst case) in order to find an element.

It is important to observe that the insertion cost for lightly loaded tables with four tables are slightly higher than with two tables. With eight tables, the phenomenon is even stronger. This makes perfect sense; The function converges to a linear function in the case of the number of tables equal the number of elements. In that case, we can have a load factor of 1.0, but the insertion cost will be linear. Thus, when using more tables, we get a slightly longer insertion time for lightly loaded tables, but it is, on the other hand, possible to utilize a large load factor.

The case with two tables and two functions per table are almost identical to the case with four tables and one function per table. We can observe the exact same behavior at load factors around 0.8 with an average insertion cost of 6 updates per insertion. In the same way, four tables and two functions per table is very similar to eight tables and one function per table.

The conclusion is that it does not matter if the number of tables is increased or the number of functions per table is increased; The results are the same. Since one function per table eliminates the extra memory (or extra comparison) needed to determine the hash function used, we suggest using one function per table.

B. Doubling technique

This section introduces three slightly different techniques for increasing (and decreasing) the tables in order to implement a fully dynamic version of the scheme.

B.1 Simulation setup

This test is a dynamic test, in the sense that the table size is not fixed. Elements are inserted into an initially empty table of smallest size (1 element per table). As the table gets full, it is rehashed according to the doubling technique tested. The plot (Fig. II-B.1) shows the total number of table updates (including the ones needed for rehash) divided by the number of inserted elements. In this simulation, no mean values are used.

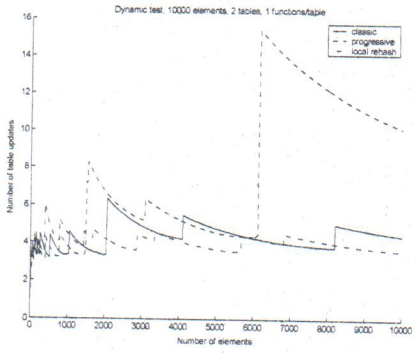

Fig. 2. Test of doubling technique.

B.2 Classic

One simple way of obtaining a fully dynamic scheme is to use the standard doubling/halving technique as suggested by Pagh and Rodler. This means that when the load factor increases above a predefined threshold, a new cuckoo table of twice the size is allocated and all elements are moved (rehashed) into the new table. The old table is then discarded. Similarly, when the load factor decreases below a predefined threshold, the table size is halved in the same manner. We will call this technique the "classic" technique. Observe that both the basic tables are increased/decreased equally much in this technique.

The plot (Fig. II-B.1) indicates that the classic doubling technique uses approximately 3 table updates/insertion. It can be observed that the plot has very distinct "steps" at integer powers of 2. This can be explained by that the hash table became full and needed to be rehashed into a larger (twice as large) table. Between the "steps", the number of table updates per insertion is constant, but the plot is decreasing. This is because the plot shows total number of table updates divided by total number of inserted elements. Since the total number of elements is increasing, the graph will be decreasing. It is possible to identify some additional "steps" in the plot; this is because the collision limit was reached and the table was rehashed into a table of the same size. The number of updates per insertion should be independent of the table size, which means the graph should be approximately constant. This seems reasonable.

B.3 Progressive

The progressive scheme uses a larger table size (twice as large) for the first tables and successively multiplicative smaller tables (table number k has size $m2^{-k}$, $k = 0\ldots$, m=number of positions/table).

This scheme is tightly associated with the sequential insertion technique. The motivation for using a larger table size for the first tables is that the probability that an element is placed in the first tables is larger, because the first table is probed first. This is only relevant if the sequential insertion technique is used.

This technique is used by Pagh and Rodler in their implementation, for that reason.

The plot (Fig. II-B.1) for the progressive technique shows very large "steps" which indicates that the rehash has failed and has to be restarted. An explanation for this phenomenon has not been found.

B.4 Local rehash

The local rehash doubling technique applies the ordinary doubling/halving technique to each individual basic table. When a basic table becomes full, only the size of that specific table is doubled; all other basic tables are untouched.

There appears to be two opportunities for placing the elements from the old (and full) basic table; either place all of them in the new basic table or insert them in the ordinary way into the cuckoo hash table, and they will be placed in any table (including the new table). The first alternative is not, however, attractive, because the elements from the old basic table were placed in that table because they happened to fit into that table with that particular hash function. If the hash function would have been different, they would have been distributed differently between the basic tables in the cuckoo table and other elements would probably have been placed in this particular basic table. Therefore, the new basic table with the new hash function cannot be expected to be able to accommodate the very same elements as the old basic table with the old function. Therefore, the elements are inserted into the cuckoo table in the usual way.

The local rehash technique shows a behavior very similar to the classical technique, although the "steps" are closer and smaller. This phenomenon makes perfect sense, since only one table at a time is rehashed and thus, we can expect the rehash to require fewer operations (table updates) and, therefore, smaller steps. Since the size of the cuckoo table is increased less when only one basic table is rehashed (compared to the classical technique, where all the basic tables are rehashed), the rehash has to be done more often. The local rehash can be a slight advantage over the classical technique, since the insertion that caused the table to rehash will be slightly faster (because the rehash was faster) than when all tables are rehashed. On the other hand, the rehash has to be done more often. The advantage is that the time for rehash is spread out over a larger number of insertions.

C. Insert heuristics

When inserting elements into the table, there are a few different ways of choosing which table to try to insert the element into. We call the way of choosing "insert heuristics". In this section, three different insert heuristics will be investigated.

This test was performed using the same test program as the "Number of tables" test (see Section II-A) with the same type of mean values.

C.1 Sequential

The sequential heuristics is the method proposed by Pagh and Rodler. The element is first inserted into the first table with the first function. If a collision occurs, the "kicked-out" element is inserted into the first table with the second function. The "kicked-out" elements are successively inserted with all functions associated with the first table, then all functions associated with the second table and, in this way, successively all tables and all functions in each table. The scheme is repeated until there is no collision or the collision limit is reached.

C.2 Random

The random insert heuristics chooses a table and a function by random (uniformly distributed) for each element inserted. If the same table and function happens to be selected twice, a new random table and function is selected. The "kicked-out" elements are inserted in the same way.

C.3 Least full

The least full insert heuristics is very similar to the sequential insert heuristics. The difference is that the sequence is started at the least full table with the first function. After that, all elements are inserted in the same way as for the sequential heuristics.

C.4 Simulation

The sequential insert heuristics is the one proposed by Pagh and Rodler. It can be seen from the plot (Fig. II-B.1) that it does not seem to make much difference between the different heuristics. This might be related to the input data that is perfectly random. Observe that there is a peak in the plot for the least load heuristics around a load factor of 0.45. It is possible that this behavior can be related to a large number of rehashes occurring at this point.

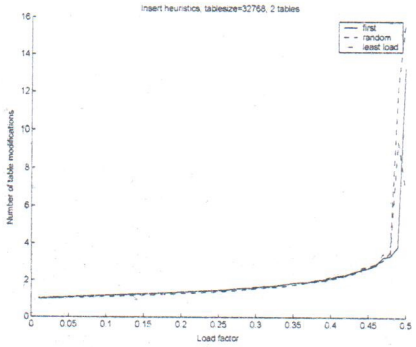

Fig. 3. Test of insert heuristics.

D. Collision limit

The collision limit suggested by Pagh and Rodler is $O(\log n)$. They also prove why this is a good limit. The functions tested in this paper are $O(\log n)$, $O(\log \log n)$ and $O(\log n / \log \log n)$. Both $O(\log \log n)$ and $O(\log n / \log \log n)$ will result in a smaller number of collisions before rehash than $O(\log n)$ (asymptotically). Since the collisions can loop before the whole cuckoo table is probed, we want to investigate if changing this function can improve performance.

D.1 Simulation

The maximum collision limit test was performed using the same test program as the "Number of tables" test (see section II-A) using the same type of mean values.

Elements are inserted into an initially empty table using different maximum collision functions. The number of performed table updates per insertion is measured.

We observe from the plot (Fig. II-D.1) that all three functions behave in almost exactly the same way for load factors below approximately 0.42 with an increasing number of table updates needed as the load factor increases. This makes sense since it can be expected to be harder to find a free slot when the tables are heavily loaded. The fact that all functions behave almost identical for lightly loaded

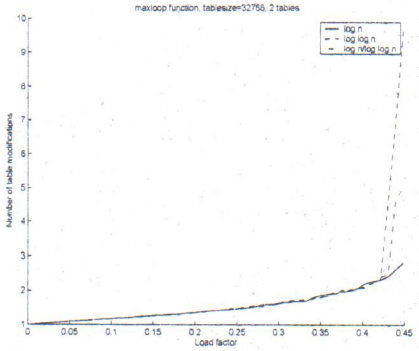

Fig. 4. Collision limiting function.

tables does probably indicate that the tables does not need to be rehashed (it is always possible to find an empty slot) or needs to be rehashed very seldom. For heavily loaded tables (above 0.42) we observe that the $O(\log \log n)$ function causes a rapidly increasing number of updates to be performed as the load factor increases. This is probably because the search for a free slot was aborted too early and the tables were rehashed. The same behavior (but not as strong) can be observed for the $O(\log n / \log \log n)$ function. We can conclude that the $O(\log n)$ function proposed by Pagh and Rodler behaves best, with a comparatively slow increase.

III. Conclusions

In this report, several generalizations of the cuckoo hashing scheme have been studied through experiments (simulations) on a test implementation of the scheme. Several of the generalizations does indeed lead to a better scheme, but a few of them does not.

The simulation for the number of tables shows that there exists a tradeoff between search time and insertion time. The search time is always, by construction of the scheme, constant, but the constant is higher when more tables are used. When using four tables, the search time is twice as large as when using two tables, but it is possible to fill the tables up to approximately 80% instead of 50%.

The use of more than one function per table improves the scheme in the same way as when using more tables, but has other disadvantages (longer insertion time) and is more complex to implement. Therefore, it is a better choice to use more tables and one function per table.

The best doubling technique seems to be the local rehash technique, because it increases the total table size in smaller steps than both the others. This effect becomes even stronger when more than two tables are used.

The collision limiting function does only seem important for heavily loaded tables. The function that did perform best in this test was the $O(\log n)$ function.

The insert heuristics does not seem to be very important and it is therefore reasonable to choose the simplest heuristics, namely the sequential heuristics.

We can conclude that local rehash, sequential insert and $O(\log n)$ collision limit is a good choice together with four

or eight tables, depending on application.

A. Further work

Theoretical results that verifies some of the simulations in this report remains to be discovered. This includes the exact dependence between the number of tables and performance (insert performance and load factor). It would also be interesting to investigate if it is possible to choose hash functions that perform better together than when they are chosen completely independently. A theoretical analysis of the hash functions also remains to be done. The progressive rehash technique did not behave well in the simulations and an explanation has not been found.

References

[CLR89] Thomas Cormen, Charles Leiserson, and Ronald Rivest. *Introduction to Algorithms*. The MIT Press, Cambridge, MA, 1989.

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

[PR01] 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, Berlin, 2001. Springer-Verlag.