# Cuckoo Hashing

## Overview of performance improvements in real-world, hardware applications

Katharina Fey

August, 2016

Humboldt-Universität zu Berlin

## Abstract

Cuckoo hashing is a highly efficient multiple-choice hashing data structure with amortised constant insertion and lookup times, good space efficency and low chance of overflow. However it has a non-neglectable chance that insertion will take $\Omega(log(n))$ instead of $\Theta(1)$. This makes it unsuitable for certain real-time applications such as network routers or RAID controllers.

Since it's introduction in 2001 by Pagh and Rodler many researchers have published attempts at de-amortising the performance of cuckoo hashing. This includes different implementations and additions to make it a viable choice when dealing with real-time applications where a (even if polynomially small) chance of $\Omega(log(n))$ is unacceptable.

In this paper we will give an overview of existing research on cuckoo hashing and in detail examine an attempt made by Kirsch and Mitzenmacher which was later refined by Arbitman, Naor, and Segev to a guaranteed $\Theta(1)$ insert time which makes it viable for certain low-latency harware routing tasks.

## 1 Introduction

A dictionary is an abstract data type widely used throughout computer science. It keeps a set of keys $K$ in relation to a set of values $V$ in a finite collection called $C$. Queries such as "$k \in K$?" provide access to the respective values of $V$ for insertion, lookup, deletion and update.

To understand how to most efficiently implement a dictionary we first need to understand some basic properties of hash functions. A hash function is always deterministic so that a function $H$ with input $k$ always yields the same $H(k) \rightarrow k'$.

A hash function also always has a predetermined output range of values. A very simple example of that would be the hash function $H(k) = k \mod m$ where $m$ is a scaling factor that can adjust the output range. With that in mind the output range is $H(k) = n$ with $n \in \{0, ..., 5\}$ for $m = 6$.

In addition to that a good hash function will also provide uniformity across its possible outputs. This means that every value in a possible output range should

have an eqaul chance of being generated by an input. What that often also includes is that input values that are similar don't map to similar hash output values. And while to some extent this is an idealised assumption, there are hash functions that attempt to aproximate this behaviour.[OP03]

On these principles we define a hash function $H$ which takes a variable length string input so that $H(k) \to k'$ is deterministic and $k'$ is a number of maximum size $L_k$. Suppose $C$ is stored in a table (of maximum size $L_k$) with $m$ entries numbered $0, 1, ..., m-1$ then an item $(k, v)$ can be inserted by taking the value $v$ and storing it at a position in the table $m$ determined by the output of $H(k) \to k'$ so that $m(k') \to v$.

In section 1.1 and 1.2 we discuss performance parameters that are important to consider when implementing a dictionary through hash functions.

In section 2 we introduce cuckoo hashing, showing to have the properties of the perfect dynamic hashing scheme described in [Die+94] by Dietzfelbinger et al.

Finally in section 3 we examine a method first introduced by Kirsch and Mitzenmacher in [KM07] and later on refined by Arbitman, Naor, and Segev in [ANS09] solving these problems and analyse their performance with real world data (in use cases outside those of classic cuckoo hashing) to hightlight the performance improvements and compare them against regular cuckoo hashing in certain environments.

## 1.1 Performance

The performance of a hashtable is dependant on a few factors. The three main parameters for performance analysis are lookup time, update time and space complexity. Because an update operation to the table is merely a successful lookup, followed by a simple write operation and an insertion is an unsuccessful lookup, followed by another write operation we can reduce the significant performance factors to be lookup time and space complexity.

In theory a perfect hashtable utilising a perfect hash function has an insert and lookup time complexity of $\mathcal{O}(1)$. However it would require enough space to store all possible results (called the Universe) of hash function $H(k)$ where $k$ is an arbitrary piece of data. If there are two $k$ that yield the same $k'$ under the hash function $H$ we encounter a collision. Limitation in space but also imperfections (non uniformity or limitation of output range) of the chosen hash function $H$ lead to collisions. These increase the time complexity of lookup operations and thus any operation performed on a hash table.

When encountering a collision a "collision resolution scheme" or "collision resolution strategy" needs to be applied. Because hash collisions are deemed unavoidable and are responsible for performance bottlenecks in hash-based dictionaries, these strategies are a major focus point of research.

A hashmap trades-off time and space complexity through the selection of a collision resolution strategy. Ideally we would be able to handle colliding items in a way that both items are still accessable in $\mathcal{O}(1)$ while not using more than $\mathcal{O}(n)$ space. Such a strategy would be called the perfect collision resolution strategy and can very often only be approximated.[PR04].

## 1.2 Collision Resolution Strategies

Before we consider cuckoo hashing as a collision resolution strategy we will high-

light other strategies due to their popularity [SG09].

1. Chained Hashing
2. Linear Probing
3. Double Hashing

Early theoretical analysis of hashing schemes was typically done under the assumption that hash function values were uniformly random and independent. Precise analyses of the average and expected worst case behaviors of the above mentioned schemes have been made, see e.g. [PR04]

Linear probing is a strategy where for a colliding key $k$ with chosen index value $k'$, the next slot in the table $k'+1$ is chosen if the original space was filled. If this next space is also occupied $k'+2$ will be considered.

This strategy utilises the localisation principle of hash values that (due to their uniformity) will form groups of items in the table which means that blank spaces will be left between groups to be filled.

When utilising the Linear Probing aproach, we can expect the number of probes for a successful probe to be $\frac{1}{2}(1+\frac{1}{(1-d)})$ and for an unsuccessful probe to be $\frac{1}{2}(1+\frac{1}{(1-d)^2})$, where $d$ is the occupancy factor of the hash table we are probing.

Double Hashing is a strategy where for a colliding key $k$ with chosen index $H_1(k) \rightarrow k_1'$ the next slot in the table will be determined by using a different hash function $H_2$ for which a new $k_2'$ will be calculated. If the strategy runs out of hash functions to use or when an upper limit is reached, the table is rehashed completely.

With this strategy the expected costs are $\frac{\ln \frac{1}{1-d}}{d}$ for a successful lookup with the longest sequence being $\mathcal{O}(\log n)$ and $\frac{1}{1-d}$ for an unsuccessful lookup with the longest sequence being $\Theta(\frac{\log n}{\log \log n})$.

Chained Hashing is a strategy that doesn't consider buckets in the table of a dictionary to be of size 1. Instead of storing the actual items (a value $v$ and a key $k$ as satellite data), we store the head to a linked list of entries in the bucket.

When inserting an item $i = (k, v)$, the key $k$ is hashed and added to the tail of the linked list found in the bucket at $m(k')$. On retrieval the satelite data (a copy of the key) is required to find the correct item in the linked list again.

When using Chained Hashing the number of probes on successful lookup is $1+\frac{1}{d}$ and for an unsuccessful lookup $1+\frac{d^2}{2}$ with the chain length being relevant in the lookup times. The maximum chain length would average out to $\Theta(\frac{\log n}{\log \log n})$.

From the above data we can derive that chained hashing is better for lookups than double hashing with double hashing being better than linear probing [PR04][GS78].

# 2 Cuckoo Hashing

Cuckoo hashing was first introduced by Pagh and Rodler in 2004 and describes a highly efficient multiple choice hashing dictionary built as the dynamisation of the static dictionary described in [Pag01] and later in [Pag06].

A cuckoo dictionary consists of multiple tables and associates multiple hash functions with these tables. Assume we have two tables $m_1$ and $m_2$ to store a series of items from a collection $C$. We in-

sert items into $m_1$ by determining $k'_1$ with $H_1(k) \rightarrow k'_1$ and $m_2$ by determining $k'_2$ with $H_2(k) \rightarrow k'_2$.

When inserting an item $i_1$, we first consider table $m_1$. If $m_1(H_1(k)$ is free, we are done. If the bucket is filled by a vallue $i_2$, we remove $i_2$ from it and insert $i_1$ instead. Afterwards we rehash $i_2$ with $H_2$ to insert into table $m_2$. This process is repeated until no more collisions occur.

The following pseudo code describes the insert operation in more detail.

```
function insert(x)
    if lookup(x) then return
    loop MaxLoop times
        if T1[h1(x)] = ⊥ then T1[h1(x)] ← x; return
        x ↔ T1[h1(x)]
        if T2[h2(x)] = ⊥ then T2[h2(x)] ← x; return
        x ↔ T2[h2(x)]
    end loop
    rehash(); insert(x);
end
```

Retrieving an item from the dictionary is done by hashing a key with multiple functions and checking different locations in all of the tables. In our example an item $i = (k, v)$ can either be found at $m_1(k')$ with $k' = H_1(k)$ or $m_2(k')$ with $k' = H_2(k)$ but never in both [PR04].

The following pseudo code describes the lookup operation.

```
function lookup(x)
    return T1[h1(x)] = x  ∨  T2[h2(x)] = x
end
```

[PR04].

In that process loops can occur which means that each insertion needs to be monitored for a maximum depth of inserts. If said depth is reached new hash functions are chosen, and both $m_1$ and $m_2$ are rehashed with $H_3$ and $H_4$ respectively.

Figure 2.1 highlights how multiple item keys hash with different hash functions and how some values have either looping or colliding hash values. Loops in particular mean that some insertions may be impossible to resolve.

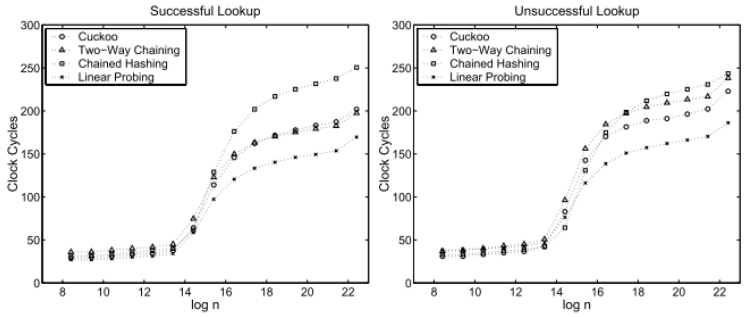


Figure 1: Time graph for insert and delete operations [PR04]

Additionally a test may be conducted to estimate the population density of both tables to increase their size to allow for less collisions during future insertions [PR04].

Figure 1 show some experimental data collected by Pagh and Rodler. It highlights the performance of cuckoo hashing under lookup, insert and deletion in comparison to several different collision resolution strategies over a series of insert operations $n$ ranging up to $2^22$ elements.

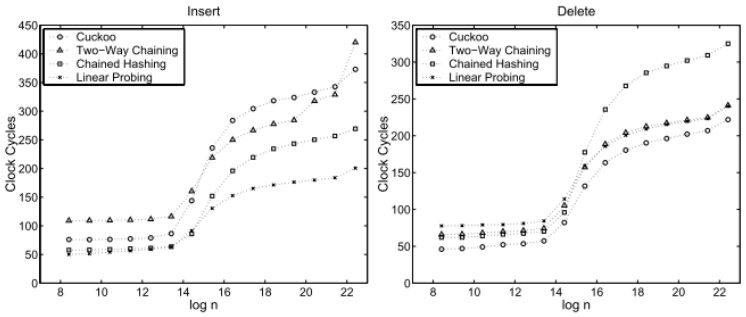


Figure 2: Time graph for insert and delete operations [PR04]

The experiments were conducted on an 800MHz Intel® Pentium® III processor with 16KB of level 1 cache and 256KB of level 2 "advanced transfer cache". The system had access to another 256MB of memory on PC100 extension cards.

Analysing the graphs in 1 and 2, it is clearly visible that all collision resolution strategies break down past $2^{14}$ elements in the collection. Pagh and Rodler speculate that this is due to the limitation of L1 and L2 cache of the CPU. Experimental data on more modern architectures is unfortunately lacking.

What is also visible from the graphs is that cuckoo hashing demonstrates the worst insert time performance of all measured strategies. When considering lookup, its performance sits between linear probing and double chained hashing.

However when doing delete operations, cuckoo hashing performs best of the selected strategies, across all data batch sizes.

The experimental results seem to confirm our assumption that insertions into a cuckoo dictionary can take considerable time due to "rehash-chains" where large parts of the tables will be moved to accomidate new data. Thus on average an insertion into a cuckoo dictionary takes $\mathcal{O}(1)$ but can require $\Omega(log(n))$ in the worst-case [KM07].

While the above demonstrated experimental results are excellent at keeping memory consumption low and guaranteeing $\mathcal{O}(1)$ lookup times which makes cuckoo hashing ideal for many software applications, there are certain (often hardware) low-latency applications where cuckoo hashing is unsuitable due to its non-predictable insert time [DM03].

# 3 Queued cuckoo hashing

Kirsch and Mitzenmacher presented a variant of cuckoo hashing centered around its application in routing hardware to reduce the amount of Content Addressable Memory (CAM) that is required. A CAM is a hardware circuit that uses data tagging to store limited length data entries in a cache that allows for constant time access.

In their proposal they at first focus on cuckoo hashing with $d > 2$ subtables, however later on switch to an implementation of $d > 4$. This has shown to be more practical and offer significant space savings [Fot+05].

In following examples we will consider algorithms to use an arbitrary number of tables to demonstrate the principles independently of implementational differences. In addition to that we have a hardware content addressable memory of size $S$. An item is either found in one of the tables or the CAM but never both. And the lookup code from the previous section can be adjusted to allow for checking inside the CAM.

An insertion can be described with a series of sub-operations consisting of the item $i$ that is to be inserted as well as some metadata that describes what previous insertion failed and what future insertion should be considered. This metadata comes in form of a table number.

Assume we want to insert an item $i = (k, v)$ into this modified cuckoo dictionary. We attempt to insert it into $m_1$ by computing an index in the table with $H_1(k) \rightarrow k'$. If this slot is taken, we add the item to a sub-operation in addition to some metadata which describes that it was insertion 1 and next up to try is table $m_2$.

This insertion sub-operation is then added to the CAM. After this we are ready for the next insertion.

This small change gives us the ability to finish insertions quickly (within a single step) while also giving us the memory efficiency of cuckoo hashing as well as the ability to keep the hardware costs on the CAM down.

What this means however is that many items will end up in the CAM that might be insertable. And while the CAM will give us quick access times to our data, it is desirable to keep the number of items in it as low as possible. What this means is that we need to apply a strategy at removing items from the CAM.

Deletions are more difficult in this modified cuckoo dictionary as operations are always split into sub–operations and stored in the queue. A deletion will also be stored in a queue before it is handled.

It might therefore be easier to keep a second list of items that have been marked for deletion until the sub-operation can be picked from the CAM.

Keeping the number of items in the CAM low can be accomplished by one of two strategies. The first requires a deletion in the system to occur which could potentially free up a slot that blocked us previously from inserting something into our tables. This is easily done by checking items in the CAM for insertability after a deletion has occured.

The other strategy requires access to an asyncrhonous runtime environment where insert operations can be tried continously. Some of these sub-operations will fail and add new sub-operations to the queue which will then be handled at a later time. It means however that some sub-operations will be inserted and thus keep the occupancy of the CAM down [KM07]

## 3.1 Queue policies

Kirsch and Mitzenmacher introduced a few queue policies with examples which have a considerable impact on the performance of queued cuckoo hashing. These policies can be applied to the CAM to allow the sub-operations stored in it to be ordered in a way that is most likely to yield positive insertions at the beginning as iterating over the entire CAM might not always be possible.

The three base queue policies that were introduced in [KM07] are "naive", "rotating" and "PQage". We will quickly elaborate these policies and show experimental data with them.

First we define an *age* parameter. This parameter is set when an insertion lands in the CAM. This way the items in the CAM can be sorted by insert-time.

In the standard cuckoo hashing we process an iteration sequentially, only terminating when we have successfully inserted an item.. With the modifications made to the dictionary breaking up insertions into sub-operations and saving them in the queue this corresponds to placing new items at the back of the queue while new sub-operations created by processing an existing item in the queue yields new items at the front of the queue.

This approach can give rise to large queue sizes as a difficult insertion can create loops or block other operations at the back of the queue without resolving the insertion. Because this approach doesn't use the ability to re-order items in the CAM, it is called the native policy.

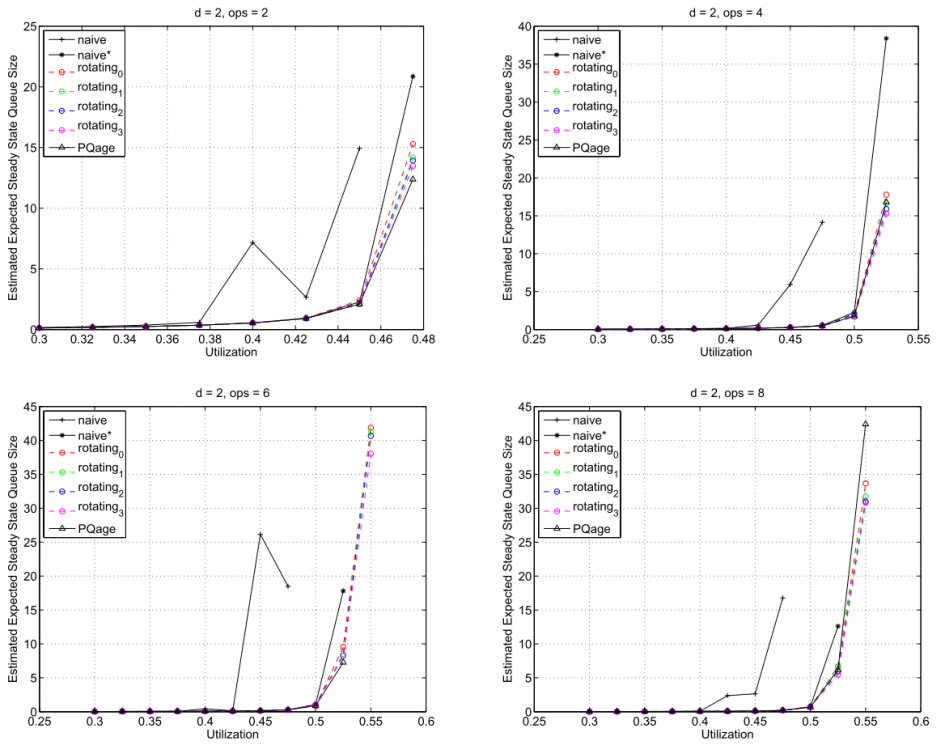


Figure 3: Expected queue sizes for different policies [KM07]

Using our previously defined *age* parameter, we can start to re-order items in the queue. It seems reasonable to assume that newer items in the CAM have a higher chance of being inserted than older items. Because this is not possible in our applications, we will need to let these items remain in CAM until elements are removed from the tables. This way we increase the chance of resolving new operations quickly while letting old operations rest in the CAM until new places in the tables are cleared up. We shall call this policy PQage because it implements a "Priority Queue" over the age of the elements.

While this approach does ideal, it is however difficult to implement and includes moving vast amounts of items in the CAM which might require too many resources in terms of time and space in the CAM.

The goal is to try to aproximate the behaviour of PQage without having to constantly re-order items in the queue. Kirsch and Mitzenmacher describe a queue policy they call rotating where we place a new sub-operation $o$ at the front of the queue if it is created by some other sub-operation and its age is at most $I$. Otherwise we place it at the back of the queue. Therefore we priorities items with ages smaller than $I$.

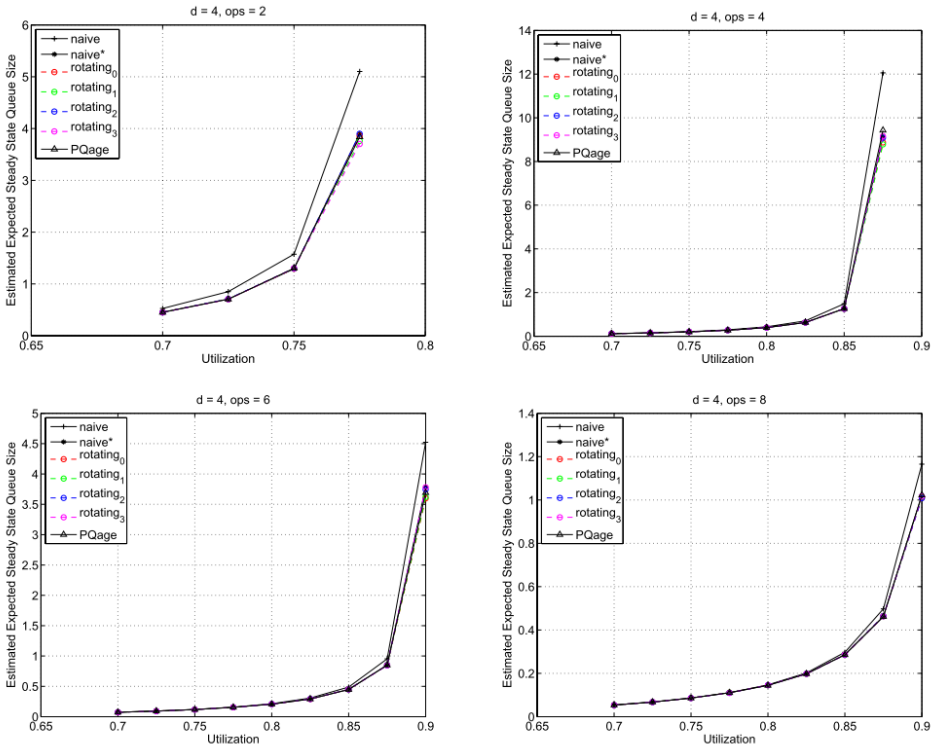


Figure 4: Expected queue sizes for different policies [KM07]

What this allows is to rotate through the sub-operations to avoid getting stuck processing a single failing operation. This policy is also simpler to implement than PQage [KM07].

Some experimental data can be found in Figures 3 and 4 which is from a more realistic set of experiments for $d = 2$ and $d = 4$ respectively. The experiments focus on the average size of the queue if the table is designed for a specific utilisation factor $u$ where 1 represents a completely full and 0 a completely empty table. Values above 50 items in the queue have been removed for readability.

Immediately visible is that PQage yields the best results across both sets of experiments, although the advantage is less visible in the second set of graphs where four tables were used. The rotating strategies with different offsets yield simmilar results, in some scenarios better and some worse than QPage.

The naive approach yields the lowest results, with being over 50 items and thus not showing up for higher utilisation factors. What this shows is that there is a definate advantage of giving priority to newer elements while potentially ignoring older ones in the CAM [KM07].

## 3.2 Provable Worst-Case Performance

While the policies described in the previous section highlight that adding a hard-

ware queue as a second data storage can greatly increase the performance of cuckoo hashing, it does not fully describe an insert operation.

Arbitman, Naor, and Segev propose an alteration to the previously described scheme in an attempt to provide a provable worst case performance while keeping the number of elements in the queue as low as possible without the need of a complicated queue policy.

They follow the same approach made by Kirsch and Mitzenmacher and use a hardware CAM as a storage buffer, using a very simple queue policy to determine which element will be processed next.

The lookup procedure is unchanged from the one described by Kirsch and Mitzenmacher. What is worth noting here is that Arbitman, Naor, and Segev focus on cuckoo dictionaries with only $d = 2$ tables while most of the experimental data from [KM07] was done with 4.

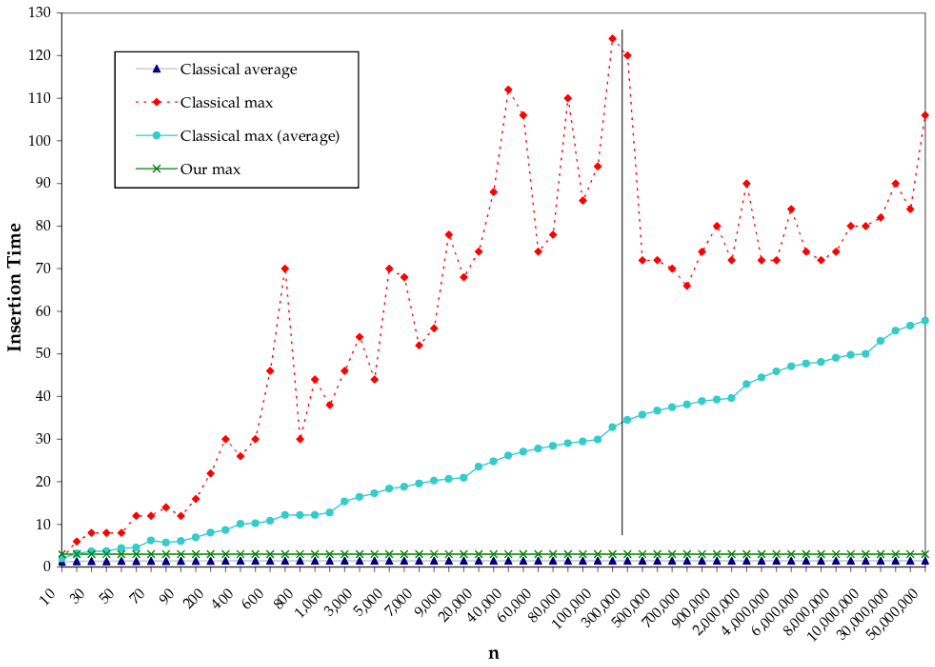


Figure 5: Comparison between traditional cuckoo hashing and augmented deamortisation [ANS09]

An insertion into the in [ANS09] proposed structure is done as follows. The entire process is done under a globally defined parameter $L$. An item $i_1 = (k_1, v_1)$ is to be inserted. Instead of placing it in one of the tables, we bundle the item, together with an additional piece of information and then insert this pair $p_1 = (i_1, 1)$ at the back of the queue.

We then procede to take elements of the

head of the queue and attempt to insert them into our tables labled $m_1$ and $m_2$ when dealing with $d = 2$. Assume the element at the head of the queue is labled $p_{head} = (i_{head}, 2)$ we attempt to insert $i_{head}$ in the index $H_2(k_{head}) \rightarrow k'_{head}$ as with a normal cuckoo dictionary insertion. If this space is empty, we successfully insert $i_{head}$ and procede by taking the next element off the head of the queue for insertion. If however $m_2[k'_{head}]$ is occupied, we displace the item and insert $i_{head}$ instead. We then procede to insert the displaced element as usual in cuckoo hashing by moving it into a different table.

This process is repeated until $L$ elements have been moved either from the queue to either $m_1$ or $m_2$ or internally between $m_1$ and $m_2$. When reaching $L$ steps, the currently nestless element is placed at the head of the queue, with an extra bit of information indicating which table it should be inserted into next.

The deletion and lookup functions are defined by the property that any element $x$ is either stored in $m_1(H_1(k_x))$ or $m_2(H_2(k_x))$ or the queue.

This modified variant of cuckoo hashing in a hardware context guarantees constant lookup and deletion times with a searchable CAM. It is however possible to use this variant purely in software at which point it is no longer guaranteed that deletion and lookup will run in constant time as the queue might hold a significant amount of items [ANS09].

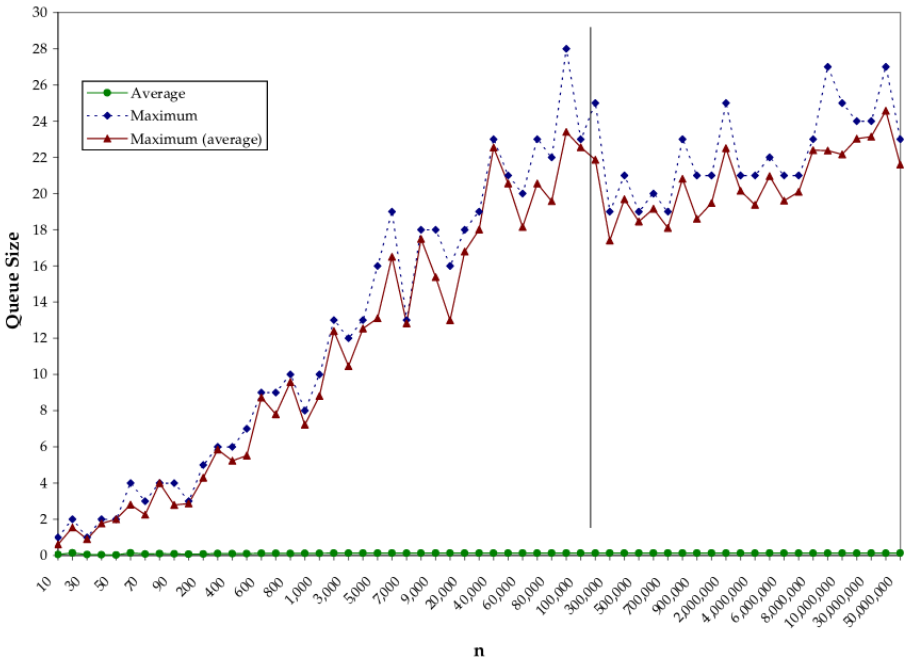


Figure 6: Comparison between traditional cuckoo hashing and augmented deamortisation [ANS09]

Very expensive rehashing of all items can be avoided by leaving certain elements in the CAM forever.

In Figure 5 an experimental run of this augmented cuckoo hashing structure can be found. We are comparing it to the average insert time values of a traditional cuckoo dictionary and the respective maximum values. The graph doesn't show anything unexpected as we are artificially limiting the step-delay to $L$, which in this example was 3.

The experiment becomes interesting when we consider the graph in Figure 6 which highlights the required queue size during the experiment.

This shows that even with a queue of 50 elements, we are able to reliably and safely insert elements into the augmented dictionary, without showing a run-away effect on insertion times such as the traditional cuckoo dictionary.

This makes cuckoo hashing viable in a new segment of datastorage problems, including but not limited to hardware applications for routers and storage controllers.

## 4 Conclusion

We examined cuckoo hashing as an alternative to established strategies to manage data collisions in hash based dictionaries. We also analysed performance bottlenecks and their impact on certain applications.

While some of the problems around cuckoo hashing could be solved by Kirsch and Mitzenmacher, Arbitman, Naor, and Segev and others, there are still many open questions about cuckoo hashing and potential performance improvements.

Some of these open questions have been discussed recently by Mitzenmacher and showcase important research opportunities in order to make cuckoo hashing faster, safer and potentially viable in more applications.

These include a more in-depth analysis of the use of queues and stashes to aid cuckoo hashing in buffering difficult items as well as more experimental data with more than 2 tables.

What is also currently still unclear is how randomness can impact the insertion times of a cuckoo hashing dictionary.

When considering the de-amortization of cuckoo hashing in hardware applications done by [KM07] and [ANS09], one prospect was to allow an augmented dictionary to be viable for routing tasks with the aspect of clocked adversaries. While it is assumed that the modifications done are sufficient, there is a lack of security analysis of these modifications.

What also seems to be lacking are some modern performance analysis with larger CPU cache sizes and different applications. Many more open questions are included by Mitzenmacher in [Mit09].

2

## References

[ANS09] Yuriy Arbitman, Moni Naor, and Gil Segev. "De-amortized cuckoo hashing: Provable worst-case performance and experimental results". In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 5555 LNCS.PART 1

(2009), pp. 107–118. ISSN: 03029743. DOI: 10.1007/978-3-642-02927-1_11. arXiv: 0903.0391.

[Die+94] Martin Dietzfelbinger et al. "Dynamic Perfect Hashing: Upper and Lower Bounds". In: *SIAM Journal on Computing* 23.4 (1994), pp. 738–761. ISSN: 0097-5397. DOI: 10.1137/S0097539791194094. URL: http://epubs.siam.org/doi/abs/10.1137/S0097539791194094.

[DM03] Luc Devroye and Pat Morin. "Cuckoo hashing: Further analysis". In: *Information Processing Letters* 86.4 (2003), pp. 215–219. ISSN: 00200190. DOI: 10.1016/S0020-0190(02)00500-8.

[Fot+05] Dimitris Fotakis et al. "Space efficient hash tables with worst case constant access time". In: *Theory of Computing Systems*. Vol. 38. 2. 2005, pp. 229–248. ISBN: 0302-9743. DOI: 10.1007/s00224-004-1195-x.

[GS78] Leo J. Guibas and Endre Szemeredi. "The analysis of double hashing". In: *Journal of Computer and System Sciences* 16.2 (1978), pp. 226–274. ISSN: 10902724. DOI: 10.1016/0022-0000(78)90046-6.

[KM07] Adam Kirsch and Michael Mitzenmacher. "Using a Queue to De-amortize Cuckoo Hashing in Hardware". In: (2007).

[Mit09] Michael Mitzenmacher. "Some open questions related to cuckoo hashing". In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. Vol. 5757 LNCS. 2009, pp. 1–10. ISBN: 3642041272. DOI: 10.1007/978-3-642-04128-0_1.

[OP03] Anna Ostlin and Rasmus Pagh. "Uniform hashing in constant time and linear space". In: *Proceedings of the thirty-fifth ACM symposium on Theory of computing - STOC '03*. 2003, p. 622. ISBN: 1581136749. DOI: 10.1145/780542.780633. URL: http://dl.acm.org/citation.cfm?id=780542.780633.

[Pag01] Rasmus Pagh. "On the cell probe complexity of membership and perfect hashing". In: *Proceedings of the thirty-third annual ACM symposium on Theory of computing - STOC '01* (2001), pp. 425–432. ISSN: 07349025. DOI: 10.1145/380752.380836. URL: http://dl.acm.org/citation.cfm?id=380752.380836.

[Pag06] Rasmus Pagh. "Cuckoo hashing for undergraduates". In: *IT University of Copenhagen* (2006), pp. 1–6. URL: http://www.itu.dk/people/pagh/papers/cuckoo-undergrad.pdf.

[PR04] Rasmus Pagh and Flemming Friche Rodler. "Cuckoo hashing". In: *Journal of Algorithms* 51.2 (2004), pp. 122–144. ISSN: 01966774. DOI: 10.1016/j.jalgor.2003.12.002.

[SG09] Mahima Singh and Deepak Garg. "Choosing Best Hashing Strategies and Hash Functions". In: *2009 IEEE International Advance Computing Conference, IACC 2009*. 2009, pp. 50–55. ISBN: 9781424429288. DOI: 10.1109/IADCC.2009.4808979.