Some Open Questions Related to Cuckoo Hashing

Michael Mitzenmacher*

School of Engineering and Applied Sciences
Harvard University, Cambridge, MA 02138
michaelm@eecs.harvard.edu

Abstract. The purpose of this brief note is to describe recent work in the area of cuckoo hashing, including a clear description of several open problems, with the hope of spurring further research.

1 Introduction

Hash-based data structures and algorithms are currently a booming industry in the Internet, particularly for applications related to measurement, monitoring, and security. Hash tables and related structures, such as Bloom filters and their derivatives, are used billions of times a day, and new uses keep proliferating. Indeed, one of the most remarkable trends of the last five years has been the growing prevalence of hash-based algorithms and data structures in networking and other areas. At the same time, the field of hashing, which has enjoyed a long and rich history in computer science (see e.g., [26]), has also enjoyed something of a theoretical renaissance. Arguably, this burst of activity began with the demonstration of the power of multiple choices: by giving each item multiple *possible* hash locations, and storing it in the least loaded, remarkably balanced loads can be obtained, yielding quite efficient lookup schemes [4, 7, 21, 28, 35]. An extension of this idea, cuckoo hashing, further allows items to be moved among its multiple choices to better avoid collisions, improving memory utilization even further.

In this brief note I plan to describe some recent work in the area of cuckoo hashing, providing some focus on several remaining open problems, with the hope of spurring further research. The presentation may admittedly be somewhat biased, focusing on my own recent research in the area: this is hopefully excused by the fact that this note is written in conjunction with an invited talk for the 2009 ESA conference in Denmark. The topic seems apropos; the paper introducing cuckoo hashing by Pagh and Rodler appeared in the 2001 ESA conference, also held in Denmark! [31, 32] Also for this reason, the focus here will be primarily on *theoretical* results and problems. There is of course also recently a great deal of interesting work in hashing combining theory and practice, as detailed for example in the survey [25].

* Supported in part by NSF grants CNS-0721491 and research grants from the Cisco University Research Program, Yahoo! University Research Program, and Google University Research Program.

2 Background : Multiple-choice Hashing and Cuckoo Hashing

The key result behind multiple choice hashing was presented in a seminal work by Azar, Broder, Karlin, and Upfal [4], who showed the following: suppose that n items[1] are hashed sequentially into n buckets by hashing each item d times to obtain d choices of a bucket for each item, and placing each item in the choice with the smallest current number of items (or *load*). When $d = 1$, which is standard hashing, then the maximum load grows like $(1 + o(1))) \log n / \log \log n)$ with high probability [20]; when $d \geq 2$, the maximum load grows like $\log \log n / \log d + O(1)$ with high probability, which even for 2 choices gives a maximum load of 6 in most practical scenarios. The upshot is that by giving items just a small amount of choice in where they are placed, the maximum load can be greatly reduced; the cost is that now d locations have to be checked when trying to look up the item, which is usually a small price to pay in systems where the d locations can be looked up in parallel. A variant later introduced by Vöcking [35], that we refer to as d-left hashing, both gives slightly improved performance and is particularly amenable to parallelization. The hash table is split into d equal-sized subtables; when inserting an item, one bucket is chosen uniformly and independently from each subtable as a possible location; the item is placed in the least loaded bucket, breaking ties to the left. This combination of splitting and tie-breaking reduces the maximum load to $\log \log n / d\phi_d + O(1)$, where ϕ_d is the asymptotic growth rate of the dth order Fibonacci numbers [35].

In practice, the $\log \log n$ terms are so small in the analysis above that one can generally assume that a suitably sized bucket will never overflow. As noted for example in [7], this effectively means that d-left hash tables can provide an "almost perfect" hash table in many settings, which can then be used to bootstrap further data structures. The hash table is only almost perfect in that technically there is some probability of failure, and of course it is not minimal in terms of size.

Cuckoo hashing [32] is a further variation on multiple choice hashing schemes. In the original description, an item can be placed in one of two possible buckets. But if on insertion there is no room for an item at any of its two choices, instead of this causing an overflow, we consider *moving the item* in one of those buckets to the other location consistent with its set of two choices. Such a move may require the move of yet another element in another bucket to prevent overflow, and so on until an empty spot for the current item is found (or until sufficiently many attempts have been made to declare a failure). An excellent picture and description is available on Wikipedia's entry for cuckoo hashing, and I encourage everyone who has not already read this entry to do so now. The name cuckoo hashing comes from the cuckoo bird in nature, which kick other birds out of their nest, much like the hashing scheme recursively kicks items out of their location as needed. Successfully placing an element corresponds to finding an augmenting

[1] We use the term items for the objects to be hashed, which are generally keys or key-value pairs; we assume throughout that items are a fixed size.

path in the underlying graph where buckets are nodes and elements correspond to edges between nodes. When there are n items to be placed in $2(1+\epsilon)n$ buckets, that is when the load of the table is less than $1/2$, all such augmenting paths are $O(\log n)$ in length with high probability. A failure occurs if an item can't be placed after $c \log n$ steps for an appropriately chosen constant c.

Although cuckoo hashing was originally introduced with just two choices per items and buckets of unit capacity, it was naturally generalized to situations with more than two choices per bucket and more than one item per bucket [15, 17]. These variations share the properties that they require checking only $O(1)$ memory locations even in the worst case. Hence, in general, we refer to the entire range of variations as cuckoo hashing, and clarify in context when necessary. For cuckoo hashing the case of $d = 2$ choices with one item per bucket is now well understood [32, 27], the cases with more choices and more items per bucket have left many remaining open questions [15, 17]. The case of $d = 2$ is so well understood because there is a direct correspondence to random graphs. We can think of buckets as vertices, and items as edges, where the edge for an item connects the two vertices corresponding to its two buckets. The choice of a bucket by an item corresponds naturally to an orientation of a directed edge. For $d > 2$, there is a correspondence to random hypergraphs, which are more technically challenging, and for more than one item in a bucket, the edge orientation problems become more technically challenging. These questions that remain for these variations are both theoretically interesting and potentially important practically, as these cuckoo hashing variants can allow very high memory utilizations, significantly higher than previous multiple choice hashing schemes.

3 Insertion Times for Random Walk Cuckoo Hashing

Let us consider the *online setting* for cuckoo hashing, where new items may arrive to be inserted and old items may be deleted. Note that this is in contrast to the offline setting, where all items are initially present and one merely wants to make a lookup table, without updates to the underlying set. When there are $d > 2$ choices or more than one item per bucket, the question of what to do when inserting a new item is more subtle than in the case with two choices. One approach is to do a breadth first search to find an augmenting path in the underlying graph structure, looking at all paths that require one move, then two moves, and so on. For constant d in both settings it is known that an insertion only takes constant expected time, although high probability bounds on the insertion time are generally very weak [15, 17]. Moreover, both because of memory and time requirements, this approach does not suitable for many practical implementations.

Let us describe an alternative approach generally much more amenable to practical implementation, is to at each step kick out a random item. Specifically let us consider the case of one item per bucket and $d > 2$ choices; in this case, we randomly kick out of the d choices the first time, and of the $d-1$ "other choices" after the first time. This avoids the storage required for the breadth first search

and is usually much faster. This approach gives a random walk on items being kicked out of their location, until an item that has an empty bucket to be placed in is found. Intuition suggests that this approach should also find an augmenting path in $O(\log n)$ steps with high probability, since at each step there seems to be a constant probability of finding an open space. While simulations suggest good, possibly logarithmic performance, the intuition is quite speculative, as it ignores dependencies in the placement of items that are troublesome for analysis.

Until recently, there was no proof of even polylogarithmic performance for the random walk cuckoo hashing approach. A current result of Frieze, Melsted, and Mitzenmacher shows that in fact with high probability over the choices of the cuckoo hashing algorithm any insertion will, with high probability, take polylogarithmic time under suitable loads for large enough numbers of choices d [19]. The argument breaks into a pair of steps: first, most buckets have an augmenting path of length at most $O(\log \log n)$ to an empty bucket; and second, the graph representing the cuckoo hashing process expands sufficiently so that, regardless of the starting point, the random walk cuckoo hashing process will find itself at a bucket with an augmenting path of length at most $O(\log \log n)$ to an empty bucket after only $O(\log n)$ steps. While this represents a significant step forward, the picture for random walk cucko hashing remains incomplete.

Open Question 1: Find tight bounds on the performance of random-walk cuckoo hashing in the online setting, for $d > 3$ choices and possibly more than one item per bucket.

4 Threshold Loads for Cuckoo Hashing

Cuckoo hashing schemes appear to have natural load thresholds. As the number of items approaches some constant c times the number of buckets (where c depends on the variant of cuckoo hashing), the time to find an augmenting path increases, and as one reaches the threshold collisions become unavoidable. Given the connection to random graphs, this behavior in unsurprising. Indeed, when $d = 2$ and there is just one item per bucket, it is known that cuckoo hash tables with load less than $1/2$ succeed with high probability, but fail when the load is larger than $1/2$. See [27] for more detailed analysis. There is a large jump in moving to $d = 3$ choices, where the threshold appears to be around a 91% load based on experiments.

When $d = 2$ and there is more than one choice per bucket, results are well understood for the offline case. Again thinking of buckets as vertices and items as edges, the problem in the offline case becomes how to orient each edge so that no vertex has degree more than k. Hence the problem corresponds to the threshold for k-orientability on random graphs, which provides a framework for finding the threshold [8, 16]. Because in the offline case there is no moving of items needed, as items are simply placed, whether these load can be achieved by a natural cuckoo hashing variant in the online setting remains open. Specifically, it would be intereseting to determine if the threshold is the same for random

walk cuckoo hashing, or for a different scheme with constant average time and logarithmic time with high probability per insertion and deletion.

When $d > 2$ choices (and one item per bucket), the threshold for the offline case is also nearly settled. Upper bounds on the theshold can found by again viewing the problem as an orientation problem on random hypergraphs, and while some additional considerations are needed, an upper bound can be calculated [5]. Lower bounds have been achieved, based on a new approach for designing dictionary and retrieval structures, based on matrix techniques [13]. (See also [33].) These techniques are quite interesting and highly recommended but a full description is beyond the scope of this short note; essentially, one utilizes a full-rank matrix with at most d ones per column derived from a hash function on the set of keys, and solves for a vector such that the multiplication of the matrix times the vector yields the value associated with each key. Storing the vector is then sufficient to generate the value associated with each key, and further requires just d lookups into the vector. As a specific example, for the important case of $d = 3$, there is an upper bound of 0.9183 for the threshold load [5], and a lower bound of 0.8894 [13]. Again, however, the question of bounds for efficient algorithms in the online setting remains more open.

Open Question 2: Tighten the bounds on the thresholds on the load capacity of cuckoo hashing schemes for $d > 2$ choices and 1 item per bucket for the offline setting.

Open Question 3: Prove bounds on thresholds for other settings, such as for cuckoo hashing with $d > 2$ choices and more than 1 item per buckets (offline or online), or for specific or general online schemes.

5 Using Stashes and Queues with Cuckoo Hashing

The failure rate of cuckoo hashing is surprisingly high. With standard cuckoo hashing using $d = 2$ choices, if n items are placed into $2(1 + \epsilon)$ buckets, the probability of a failure – that some item can't be placed or takes too long to place – is $\Theta(1/n)$, with the constant factor in the asymptotic notation depending on ϵ [27]. In theoretical papers the standard suggested response is to rehash everything in case of such a failure; this does not change the important fact that the expected average insertion time per item is constant. Rehashing, however, is unsuitable for many applications. The failure rate is smaller with more choices of items [17] or more items per bucket [15], but the high failure probability still remains a potential problem.

In [24] we show that one needs only a small, constant-sized *stash* to greatly reduce the probability of a failure. A stash should be thought of as a small, fully-associative memory, that allows an arbitrary lookup in a single time step. In hardware, this can be implemented as a content-addressable memory (CAM), as long as the size of the stash is small, since CAMs are expensive. In software, this can be implemented with a small number of dedicated cache lines. We show a stash of constant size s reduces the probability of any failure to fall from $\Theta(1/n)$ to $\Theta(1/n^{s+1})$ for the case of $d = 2$ choices case. Similar results hold for other

variants, in that the failure probability provably falls linearly by a factor of the stash size s in the exponent. Such a reduction is key for scaling to applications with millions of users. The original motivation was for potential applications to routers, and applications of this result to devices using history-independent hash tables have also been suggested [30].

This idea of allowing a *small* amount of additional space to handle collisions seems quite powerful, although it is not commonly studied in theoretical work. (Interestingly, though, one can think of the seminal work on perfect hashing of Fredman, Komlós, and Szemerédi [18] in this context.) The issue of the right scale of the additional space seems to be an interesting question. For example, in other work, we have alternatively suggested using a CAM as a *queue* for pending move operations in a cuckoo hash table [22]. The advantage of this approach is it gives an effective de-amortization of cuckoo hash inserts: by queueing operations, we can arrange for inserts to have worst-case constant time (corresponding to the average time for an insert in standard cuckoo hashing). This technique appears potentially useful as an approach for deamortizing other algorithms or data structures in hardware. We conjectured in this setting that the queue size is required to scale like $O(\log n)$, corresponding to a maximum size achieved by a queue over $O(n)$ steps. For the case of $d = 2$, this conjecture has recently been proven in [3] (see also the similar [12]).

Finally, in other work, we have considered variants that allow only one move of an item in a hash table on each insertion [23]. The motivation for this work was to consider the benefits of making the *minimum possible change* to multiple-choice hashing, which is already being used in some hardware solutions, in order to convince builders of devices to consider trying systems that allow items to move within the hash table. Besides showing significant gains, the authors were able to analyze several schemes using a fluid limit/differential equations analysis. Here, we require using a CAM that scales linearly in n. That is, we find such schemes require a CAM of size ϵn for a very small ϵ chosen by the designer (e.g., 0.2%). So now we have examples where the natural choice of a stash size is constant, logarithmic, and linear, depending on our overall goal.

Open Question 4: Extend the de-amortization analysis for cuckoo hashing to other variants, including the case of $d > 2$ choices. Can this de-amortization technique be applied to other related problems as well?

Open Question 5: Develop a more general theory of the power of stashes and appropriate scalings in the setting of hash tables.

6 Limited Randomness and Cuckoo Hashing

Even from the inception of cuckoo hashing, the question of how much randomness is required was considered a worthwhile question. While assuming perfectly random hash functions is useful for analysis, it is both theoretically and practically unappealing, since perfectly random hash functions are not readily available. From the connection with random graphs in the case of $d = 2$ choices, it

is apparent that if each hash function is independently chosen from a $c \log n$-wise independent family for an appropriate constant c, the analysis showing expected constant time per operation continues to hold. Pagh and Rodler [32] in fact showed that a hash function family derived from the work of Siegel [34] with limited independence suffices for cuckoo hashing in the case where $d = 2$. However, these hash functions still appear to be too complex to be utilized in practice. They also experimented with weaker hash functions.

Recent advances in the area include the work of [3], where a result by Braverman [6] is used to show that the analysis of cuckoo hashing with a queue holds even with only polylogarithmically-wise independent hash functions. Cohen and Kane [11] demonstrate that 5-independence (which is slightly different than but close to 5-wise independence) is insufficient for constant amortized cost per operation for cuckoo hashing with $d = 2$ choices, but also show that only one of the two hash functions needs to be $c \log n$-wise independent to obtain constant expected time per operation.

An alternative direction, taken by Mitzenmacher and Vadhan, started with the question of why simple hash functions work so well in practice [29]. As mentioned, when analyzing hash-related data structures such as cuckoo hashing, one commonly assumes that the underlying hash functions are completely random, even though this is unrealistic. But in practice, such analysis generally turns out to be accurate, even when weak hash functions, such as pairwise independent (or universal) hash functions [9], are used.

The proposed resolution was to model the data as coming from a random source, where the i'th item X_i has at least some k bits of entropy (specifically, Renyi entropy) conditioned on the previous items $X_1, \ldots, X_{i-1}$. Then results from the theory of randomness extraction imply that when a hash function H is chosen from even a pairwise independent family, the sequence $(H(X_1), \ldots, H(X_T))$ has small statistical difference from the distribution obtained if H were a perfect hash function. That is, a weak hash function is good enough, as long as there is sufficient randomness in the data. The implications of this model apply to cuckoo hashing as well as other hashing-based algorithms and data structures. Improvements on the bounds of [29] are developed in [10],

As shown by Dietzfelbinger and Schellbach, however, one cannot use this insight blindly. They demonstrate that natural families of universal hash functions, namely multiplicative hash functions and standard linear hash functions over a prime field, fail even for fully random key sets, when the key set is sufficiently dense over the universe of keys [14]. In such cases, there is *not* sufficient entropy for the results of [29] to hold, so there is no contradiction. The implications of these results to practical settings certainly appear to be a worthy of further study.

Open Question 6: Determine better bounds on the amount of randomness needed for cuckoo hashing to be effective, either in terms of the requirements of the underlying family of hash functions, the amount of randomness in the data, or both.

7 Parallelized Variations of Cuckoo Hashing

As a final area for future work, there appears to be renewed interest in parallel algorithms for constructing hash tables and related data structures, inspired by the development of multi-core processors and other mainstream hardware that allows parallelization, such as graphics processor units (GPUs). In [2], we design a practical parallel scheme for constructing hash tables on GPUs motivated in part by cuckoo hashing techniques. The setting is offline, with all items available. Essentially, items perform the random walk cuckoo hashing approach in parallel: each item tries to place itself in its first choice; each item that fails to capture its first choice location tries to place itself it its second choice, and then it third choice. (Three choices per item were used in this implementation.) Any unplaced item then tries to kick out the item placed at its first choice, and then its second choice, and so on. In order to ensure quick convergence, a two-level scheme was used, where items are first partitioned using a separate hash function, in order to give with high probability a bounded number of items (in this case 512) per partition. The parallel cuckoo hashing approach is then run in parallel on each partition. This random partitioning trades additional space for efficiency. For details, see [2].

While there is a fair amount of historical work on parallel hashing and load balancing schemes (see, for example, [1, 28]), the significant advances made in the last decade in terms of analysis and understanding of the power to move items suggests that we can obtain both stronger results and tighter analyses in theory for such parallel hashing schemes. Moreover, there may be significant opportunities for the design of efficient parallel hash table construction schemes for real hardware systems. Given the inherent potential for parallelization with multiple-choice hash tables in general and cuckoo hashing in particular, this appears to be an interesting area for future growth.

Open Question 7: Design and analyze efficient schemes for constructing and maintaining hash tables in parallel architectures, particularly modern multicore architectures.

8 Conclusion

This note provides a smattering of open questions related to the theme of cuckoo hashing. There are certainly others, and more waiting to found. Indeed, at this very conference, there are a number of papers specifically on the theme of cuckoo hashing and on the more general themes of dictionary data structures and hash-based data structures. There remains plenty of interesting work to do in this area, which offers both rich theory and practical payoff.

Acknowledgments

I thank Martin Dietzfelbinger and Rasmus Pagh for helpful discussions, assistance with references, and comments on an earlier draft of this work.