More Robust Hashing: Cuckoo Hashing with a Stash*

Adam Kirsch† Michael Mitzenmacher‡ Udi Wieder§

Abstract

Cuckoo hashing holds great potential as a high-performance hashing scheme for real applications. Up to this point, the greatest drawback of cuckoo hashing appears to be that there is a polynomially small but practically significant probability that a failure occurs during the insertion of an item, requiring an expensive rehashing of all items in the table. In this paper, we show that this failure probability can be dramatically reduced by the addition of a very small constant-sized *stash*. We demonstrate both analytically and through simulations that stashes of size equivalent to only three or four items yield tremendous improvements, enhancing cuckoo hashing's practical viability in both hardware and software. Our analysis naturally extends previous analyses of multiple cuckoo hashing variants, and the approach may prove useful in further related schemes.

1 Introduction

In a multiple choice hashing scheme, each item can reside in one of d possible locations in a hash table. Such schemes allow for simple $O(1)$ lookups, since there are only a small number of places where an item can be stored. *Cuckoo hashing* refers to a particular class of multiple choice hashing schemes, where one can resolve collisions among items in the hash table by moving items as needed, as long as each item resides in one of its corresponding locations. An abundace of collisions leading to an overlfow somewhere in the table, however, remains the bane of cuckoo hashing schemes and multiple choice hashing schemes in general. There is always some chance that on the insertion of a new item, none of the d choices are or can naturally be made empty to hold it, causing a failure. In the theory literature, the standard response to this difficulty is to perform a full rehash if this rare event occurs. Since a failure in such schemes generally occurs with low probability (e.g., $O(n^{-c})$ for n items and for some constant $c \geq 1$), these rehashings have very little impact on the average performance of the scheme, but they make for less than ideal probabilistic worst case guarantees. Moreover, for many schemes, the constant c in the $O(n^{-c})$ failure probability bound is smaller than one actually desires in practice; values of $c \leq 3$ arguably lead to failures at too high a rate for commercial applications (assuming that the hidden constants are not too small). In particular, in many applications, such as indexing, elements are inserted and deleted from the hash table over a long period of time, increasing the probability of failure at some point throughout the life of

*A conference version of this work appears in [8].

†School of Engineering and Applied Sciences, Harvard University. Supported in part by NSF grant CNS-0721491 and a grant from Cisco Systems. Email: kirsch@eecs.harvard.edu

‡School of Engineering and Applied Sciences, Harvard University. Supported in part by NSF grant CNS-0721491 and a grant from Cisco Systems. Email: michaelm@eecs.harvard.edu

§Microsoft Research Silicon Valley. Email: uwieder@microsoft.com

the table. Furthermore, if the hash table is required to be *history independent* then a failure may trigger a long series of rehashings. See [11] for details.

In this paper, we demonstrate that with standard cuckoo hashing variants, one can construct much more robust hashing schemes by utilizing very small amounts of memory outside the main table. Specifically, by storing a *constant* number of items outside the table in an area we call the *stash*, we can dramatically reduce the frequency with which full rehashing operations are necessary. A constant-sized stash is quite natural in most application settings. In software, one could use one or more cache lines for quick access to a small amount of such data; in hardware, one could effectively use content-addressable memories (CAMs), which are too expensive to store large tables but are cost-effective at smaller sizes. The intuition behind our approach is quite natural. If the items cause failures essentially independently, we should expect the number of items S that cause errors to satisfy $\mathbf{Pr}(S \geq s) = O(n^{-cs})$ for some constant $c > 0$ and every constant $s \geq 1$. In this case, if we can identify problematic items during the insertion procedure and store them in the stash, then we can dramatically reduce the failure probability bound.

Of course, failures do not happen independently, and formalizing our results requires revisiting and modifying the various analyses for the different variants of cuckoo hashing. We summarize our general approach. For many hashing schemes, it is natural to think of the hash functions as encoding a sample of a random graph G from some distribution. One can often show that the insertion procedure is guaranteed to be successful as long as G satisfies certain structural properties (e.g., expansion properties). The failure probability of the hashing scheme is then bounded by the probability that G does not satisfy these requirements. In this context, allowing a stash of constant size lessens these requirements, often dramatically reducing the corresponding failure probability. For example, if the properties of interest are expansion properties, then a stash effectively allows one to remove a subset of items of constant size when proving the expansion requirements hold for a set of items. In particular, when small (constant-sized) sets are the bottleneck in determining the failure probability, the stash allows dramatic improvements. Our work demonstrates that the technique of utilizing only a constant-sized stash is applicable to a number of interesting hashing schemes, and that one can often determine whether the technique is applicable by a careful examination of the original analysis. Furthermore, when the technique is applicable, the original analysis can often be modified in a fairly straightforward way.

Specifically, we first consider a variant of the cuckoo hashing scheme introduced by Pagh and Rodler [12], which uses two choices. We then consider variations proposed by Fotakis et al. [4], which utilizes more than two choices, and by Dietzfelbinger and Weidling [2], which allows buckets to hold more than one item. We verify the potential for this approach in practice via some simple simulations that demonstrate the power of a small stash.

Before continuing, we note that the idea of using a small amount of additional memory to store items that cannot easily be accommodated in the main hash table is not new to this work. For instance, Kirsch and Mitzenmacher [6, 7] examine hash table constructions designed for high-performance routers where a small number of items can be efficiently stored in a CAM of modest size. (In particular, [7] specifically considers improving the performance of cuckoo hashing variants by reordering hash table operations.) However, the constructions in [6] technically require a linear amount of CAM storage (although the hidden constant is very small), and the schemes in [7] are not formally analyzed. Our new constructions are superior in that they only require a small constant amount of additional memory and have provably good performance.

2 Standard Cuckoo Hashing

We start by examining the standard cuckoo hashing scheme proposed by Pagh and Rodler in [12]. Here we attempt to insert n items into a data structure consisting of two tables, T_1 and T_2, each with $m = (1+\epsilon)n$ buckets and one hash function (h_1 for T_1 and h_2 for T_2), where $\epsilon > 0$ is some fixed constant. Each bucket can store at most one item. To insert an item x, we place it in $T_1[h_1(x)]$ if that bucket is empty. Otherwise, we *evict* the item y in $T_1[h_1(x)]$, replace it with x, and attempt to insert y into $T_2[h_2(y)]$. If that location is free, then we are done, and if not, we evict the item z in that location and attempt to insert it into $T_1[h_1(z)]$, etc. Of course, this is just one variant of the insertion procedure; we could, in principle, attempt to place x in either of $T_1[h_1(x)]$ or $T_2[h_2(x)]$ before performing an eviction. We can also place an upper bound on the number of evictions that the insertion procedure can tolerate without generating some sort of failure. We find the first variant simplest to handle. Our analysis holds for all these variants with trivial modifications.

Pagh and Rodler [12] show that if the hash functions are chosen independently from an appropriate universal hash family, then with probability $1 - O(1/n)$, the insertion procedure successfully places all n items with at most $\alpha \log n$ evictions for the insertion of any particular item, for some sufficiently large constant α, and with expected constant insertion time per item. Furthermore, if the insertion procedure is modified so that, if inserting a particular item requires more than $\alpha \log n$ evictions, the hash functions are resampled and all items in the table are reinserted, then the expected time required to place all n items into the table is $O(n)$. Hence even with rehashings the expected insertion time per item remains constant because rehashings occur with small probability; however, in practice, a rehashing can be a very significant event. Finally, the analysis of [12] holds even when arbitrary keys have been deleted, as long as newly inserted keys are indeed new and not previously deleted items.

Devroye and Morin [1] show that the success of the cuckoo hashing insertion procedure can be interpreted in terms of a simple property of a random multi-graph that encodes the hash functions[1]. In particular, Kutzelnigg [9] uses this approach to show that, if the hash functions are (heuristically) assumed to be independent and fully random, then the probability that the hash functions admit *any* injective mapping of the items to the hash buckets such that every item x is either in $T_1[h_1(x)]$ or $T_2[h_2(x)]$ is $1 - \Theta(1/n)$. (In fact, [9] identifies the exact constant hidden in the Theta notation.)

In this section, we use the approach of Devroye and Morin [1] to show that if the hash functions are independent and fully random and items that are not successfully placed in $\alpha \log n$ evictions result in some (easily found) item being placed in the stash, then the size S of the stash after all items have been inserted satisfies $\mathbf{Pr}(S \geq s) = O(n^{-s})$ for every integer $s \geq 1$. Equivalently, the use of a stash of constant size allows us to drive down the failure probability of standard cuckoo hashing exponentially. We note that we assume independent and fully random hash functions throughout the paper.

We now proceed with the technical details. We view the hash functions h_1 and h_2 as defining a bipartite multi-graph with m vertices on each side, with the left and right vertices corresponding to the buckets in T_1 and T_2, respectively. For each of n items x, the hash values $h_1(x)$ and $h_2(x)$ are encoded as an instance of the edge $(h_1(x), h_2(x))$. Following [1], we call this multi-graph the *cuckoo graph*. Assuming the hash functions are fully random, the cuckoo graph is sampled from $\mathcal{G}(m, m, n)$ - the uniform distribution over bipartite multi-graphs with m vertices on each side and

[1]Some of the details in the proofs in [1] are not accurate and are corrected in part in this paper, as well as by Kutzelnigg [9].

exactly n edges.

The key observation in [1] is that the standard cuckoo hashing insertion procedure successfully places all n items if and only if no connected component in the cuckoo graph has more than one cycle. In this case, the number of evictions required to place any item can be essentially bounded by the size of the largest connected component, which can be bounded with high probability using standard techniques for analyzing random graphs. We call components with at least one cycle *cyclic components*.

The *Insert* Procedure We use the following modified insertion algorithm: As before item x is inserted by placing it in $T_1[h_1(x)]$, if that space is occupied by item y we evict y and insert it in $T_2[h_2(y)]$ and so on. However, once a *failure* is detected, i.e. the component of the cuckoo graph with the edge $(h_1(x), h_2(x))$ has more than one cycle, we put in the stash the current unplaced item. We call an edge whose insertion causes a failure a *failing edge*. The main property we use in the analysis is that a failing edge either closes a second cycle in a component or connects two cyclic components.

A few remarks are in place: The exact way in which the failure is detected does not matter and could be implemented in a variety of ways. For instance, one can run a depth first search on the observed portion of the component. Alternatively, we can observe that an insert fails only if some item is evicted three times. The complexity of these mechanisms, both in time and space, is bounded by the size of the connected component to which x belongs, and therefore they do not dominate the running time of the algorithm.

A different and natural criteria for failure is to impose an a-priori bound of $\alpha \log n$ on the number of evictions, declaring a failure if no empty slot had been found. We see later that this approach is also valid as long as α is chosen large enough. For the time being however we assume that an insertion fails if and only if the inserted element is a failing edge, i.e. it either closes a second cycle or connects two cyclic components.

Another source of variation is the identity of the item which is put in the stash. The analysis will show that any element whose edge belongs to the component of x observed by the algorithm could be put in the stash. In particular that item could be x itself or the item currently displaced when failure is detected. Thus, our analysis does not depend on the particular mechanism which is used to detect failures. Denote by G_x the cuckoo graph prior to the insertion of x, and by G the cuckoo graph with the edge associated with x. Similarly, S_x denotes the stash prior to x's insertion and S denotes the stash afterward. Denote by $G \setminus S$ the graph G without the edges associated with the stash items. The property we require from the insertion algorithm is encapsulated in the following lemma:

Lemma 2.1. *The insertion scheme described above has the property that if the insertion of x triggers a failure and C is a tree component of $G \setminus S$, then C is a tree component of $G_x \setminus S_x$.*

Proof. Assume that the insertion of x caused y to be put in the stash, and consider the graph $G \setminus S_x$. We claim that y belongs to the same component of x, that this component has more than one cycle, and that y either belongs to a cycle or is a bridge connecting two cyclic components. Recall that we call such an edge a failing edge. First we observe that if y is failing then the lemma follows. To see this consider a tree C in $G \setminus S$. Now add y and remove x to obtain $G_x \setminus S_x$. The fact that y is failing and belongs to the same component as x directly implies that C is still a tree in $G_x \setminus S_x$.

To see that y is indeed a failing edge observe that x itself is a failing edge, so if x is put in the stash we are done. If x evicts item z then z belongs to the same component as x, and furthermore, we can act as if z had been the original item which we wanted to insert, i.e. the same reasoning implies that if z is placed in the stash the lemma holds, and so on, until a failure is detected and the insertion algorithm decides to place an item in the stash. □

The following theorem is the main result of this section.

Theorem 2.1. *For every constant integer $s \geq 1$, for a sufficiently large constant α, the size S of the stash after all items have been inserted satisfies $\mathbf{Pr}(S \geq s) = O(n^{-s})$.*

The rest of this section is devoted to the proof of Theorem 2.1. Our first goal is to characterize the size the stash by the combinatorial properties of the cuckoo graph.

Let G be a graph on v vertices. The *Cyclomatic Number*[2] of G denoted by $\gamma(G)$ is defined to be the smallest number of graph edges which should be removed from G so that no cycle remains. Furthermore, such a set could be found by the following procedure: Order the edges by some arbitrary order and insert them into the graph one by one. We mark an edge if it closes a cycle upon insertion. When all edges are inserted exactly $\gamma(G)$ edges are marked. The removal of the marked edges eliminates all cycles from G. We call the marked edges *excess* edges. While the specific set of excess edges depends upon the ordering in which the edges were inserted, the number of excess edges is always $\gamma(G)$. It is well known that if G is connected and has $v + k$ edges then $\gamma(G) = k + 1$. See for instance [13, §6.1] for proof details. Denote by $T(G)$ the number of cyclic components in G. We prove below the size of the stash equals $\gamma(G) - T(G)$, which we also write as $(\gamma - T)(G)$ for convenience.

Lemma 2.2. *Let G be an instance of the cuckoo graph. Let the elements be inserted one by one in some arbitrary order. Then the size of the stash S is equal to $\gamma(G) - T(G)$.*

Proof. The proof is by induction on the number of edges in G. We maintain two invariants.

(1) The number of edges in the stash is $(\gamma - T)(G)$

(2) If C is a connected component of $G \setminus S$ and is a tree, then C is a connected component of G and is a tree.

The lemma is encapsulated in the first invariant. Both invariants hold trivially for the empty graph. Fix some ordering on the edges and let x be the last edge inserted. Denote by G_x the cuckoo graph without the edge associated with x. We assume the invariants hold for G_x and need to prove the invariants still hold after the insertion of x. We first deal with the case where the insertion of x does not cause a failure, so $S = S_x$. We first show invariant (2). Let C be a tree in $G \setminus S$. If $x \notin C$ then the induction hypothesis states that C is a component of G and a tree. If $x \in C$ then x must have connected two (possibly empty) trees in $G_x \setminus S_x$. The induction hypothesis states that these are also trees in G_x. It follows that C is a tree componenet of G.

We still assume that $S = S_x$ and show invariant (1) by considering three cases depending upon x's position in G:

[2]The Cyclomatic Number is sometimes also called the Cyclomatic Dimension or the Circuit Rank.

1. The connected component of x in $G \setminus S$ is a tree. In other words, x extends a tree in $G_x \setminus S_x$, or x connects two trees in $G_x \setminus S_x$. Since the insertion succeeds without failure, S_x does not increase in size. The induction hypothesis of the second invariant implies that x extends a tree, or connects two trees in G_x as well. Thus $\gamma(G) = \gamma(G_x)$ and $T(G) = T(G_x)$ which proves invariant (1).

2. Item x connects a tree and a cyclic component in $G_x \setminus S_x$. Denote the tree by C. By the induction hypothesis of invariant (2) it holds that C is also a tree in G_x, and thus $\gamma(G) = \gamma(G_x)$ and $T(G) = T(G_x)$.

3. Item x closes the first cycle in a component C of $G_x \setminus S_x$. Since x closes a cycle $\gamma(G) = \gamma(G_x) + 1$. The induction hypothesis states that C is a tree component in G_x, thus $T(G) = T(G_x) + 1$ and $(\gamma - T)(G) = (\gamma - T)(G_x)$ as required.

We now deal with the cases where x triggers a failure and an item y is put in the stash.

1. Item x closes a second cycle in a component of $G_x \setminus S_x$. In this case $\gamma(G) = \gamma(G_x) + 1$ and since no new cyclic components were created it also holds that $T(G) = T(G_x)$ which proves invariant (1). For Invariant (2) let C be a tree in $G \setminus S$. Lemma 2.1 implies that C is a tree component in $G_x \setminus S_x$. By the induction hypothesis C is also a tree in G_x. Now the fact that the insertion of x triggered a failure implies that x is not in C, so C is also a tree in G.

2. Item x connects two cyclic components of $G_x \setminus S_x$. Either these two components are contained in two separate components in G_x, in which case $\gamma(G) = \gamma(G_x)$ but $T(G) = T(G_x) - 1$, or these two components are connected by the edges in S_x, in which case $\gamma(G) = \gamma(G_x) + 1$ but $T(G) = T(G_x)$. Either way invariant (1) continues to hold. As before, invariant (2) follows directly from Lemma 2.1.

□

We have shown that S is distributed as $(\gamma - T)(G)$ where G is sampled from $\mathcal{G}(m, m, n)$, and we have reduced the problem of bounding the size of the stash to the problem of bounding this particular random variable. It turns out however that $\mathcal{G}(m, m, n)$ is not convenient to work with. The main obstacle is that the occurrences of different edges in the graph are not independent events. A common approach in such cases is to approximate the distribution by looking at the distribution in which each edge occurs independently with probability p, typically denoted by $\mathcal{G}(m, m, p)$. In our case however we need to take into account the multiplicity of edges. We take a similar approach but need to be more careful in our choice of distributions.

For a distribution D over the natural numbers, let $\mathcal{G}(m, m, D)$ denote the distribution over bipartite graphs with m nodes on each side, obtained by sampling $\ell \sim D$ and throwing ℓ edges independently at random, that is, each edge is put in the graph by uniformly and independently sampling its left node and its right node. Note that the cuckoo graph has distribution $\mathcal{G}(m, m, D)$ when D is concentrated at n. The distribution we will work with is $\mathcal{G}(m, m, \mathrm{Po}(\lambda))$ where $\mathrm{Po}(\lambda)$ denotes the Poisson distribution with parameter λ. Where the context is clear, we may write $\mathrm{Po}(\lambda)$ for a random variable with this distribution. The key advantage of introducing $\mathcal{G}(m, m, \mathrm{Po}(\lambda))$ is the "splitting" property of Poisson distributions used in the proof of Lemma 2.3 below: if $\mathrm{Po}(\lambda)$ balls are thrown randomly into k bins, the joint distribution of the number of balls in the bins is the same as k independent $\mathrm{Po}(\lambda/k)$ random variables. This property simplifies our analysis. But

first we need to quantify the relationship between $\mathcal{G}(m, m, \mathrm{Po}(\lambda))$ and $\mathcal{G}(m, m, n)$. To this end we use the machinery of stochastic dominance. We present it in the context of random graphs.

Definition 2.1. For two graphs G and G' with the same vertex set V, we say that $G \geq G'$ if the set of edges of G contains the set of edges of G'. Let g be a function from graphs to reals. We say g is *non-decreasing* if $g(G) \geq g(G')$ whenever $G \geq G'$.

Definition 2.2. Let μ and ν be two probability measures over graphs with some common vertex set V. We say that μ *stochastically dominates* ν, written $\mu \succeq \nu$, if for every non-decreasing function g, we have $\mathbf{E}_\mu[g(G)] \geq \mathbf{E}_\nu[g(G)]$.

Note that the function $(\gamma - T)$ is non-decreasing.

Lemma 2.3. *Fix any $\lambda > 0$. For $G \sim \mathcal{G}(m, m, \mathrm{Po}(\lambda))$, the conditional distribution of G given that G has at least n edges stochastically dominates $\mathcal{G}(m, m, n)$.*

Proof. For a left vertex u and a right vertex v, let $X(u, v)$ denote the multiplicity of the edge (u, v) in $\mathcal{G}(m, m, \mathrm{Po}(\lambda))$. By a standard property of Poisson random variables, the $X(u, v)$'s are independent, each distributed as $\mathrm{Po}(\lambda/m^2)$. Thus, for any $k \geq 0$, the conditional distribution of G given that G has exactly k edges is exactly the same as $\mathcal{G}(m, m, k)$ (see, e.g., [10, Theorem 5.6]). Since $\mathcal{G}(m, m, k_1) \succeq \mathcal{G}(m, m, k_2)$ for any $k_1 \geq k_2$, the result follows. □

We need to show that we can choose λ so that $\mathcal{G}(m, m, \mathrm{Po}(\lambda))$ has at least n edges with overwhelming probability for an appropriate choice of λ. This follows from a standard tail bound on Poisson random variables. Indeed, we choose λ such that $m(1 - \epsilon') \geq \lambda \geq (1 + \epsilon')n$, for some constant $\epsilon' > 0$. Standard tail inequalities for Poisson variables (c.f. [10, Theorem 5.4]) imply

$$\mathbf{Pr}(\mathrm{Po}(\lambda) < n) \leq e^{-\lambda} \left(\frac{e\lambda}{n}\right)^n = e^{-n[\epsilon' - \ln(1+\epsilon')]} = e^{-\Omega(n)},$$

where we used the fact that $\epsilon' > \ln(1 + \epsilon')$.

From now one we consider only graphs sampled from this distribution. Let C_v denote the (edges of) the connected component of node v. Now Lemma 2.2 implies

$$\begin{aligned}\mathbf{Pr}(S \geq s) \leq\ & \mathbf{Pr}(\max_v |C_v| > \alpha \log n) \\ & + \mathbf{Pr}(\gamma(G) - T(G) \geq s) + e^{-\Omega(n)}\end{aligned}$$

where the first term takes into account the fact that a large component in the cuckoo graph may also trigger a failure. So it suffices to show that for a sufficiently large constant α,

$$\mathbf{Pr}(\max_v |C_v| > \alpha \log n) = O(n^{-s}) \quad \text{and} \tag{1}$$

$$\mathbf{Pr}(\gamma(G)) - T(G) \geq s)) = O(n^{-s}). \tag{2}$$

Lemma 2.4. *There exists some constant $\beta \in (0, 1)$ such that for any fixed vertex v and integer $k \geq 0$, we have $\mathbf{Pr}(|C_v| \geq k) \leq \beta^k$.*

Proof. First we note that since $\lambda \leq (1-\epsilon')m$, the exponential decay of Poisson variables implies that the probability that $\mathrm{Po}(\lambda)$ is greater than $(1-\epsilon'/2)m$ is exponentially small. Next we need to show that if indeed the number of edges in the graph is such bounded, the size of a component is dominated by a process that has an exponential decay. Let $X_1, X_2, \ldots$ be independent $\mathrm{Bin}(n, 1/m)$ random variables. By standard arguments, (c.f [1, Lemma 1] and [5, §5.2]), we have that for any $k \geq 1$,

$$\mathbf{Pr}(|C_v| \geq k) \leq \mathbf{Pr}\left(\sum_{i=1}^{k} X_i \geq k\right) = \mathbf{Pr}(\mathrm{Bin}(nk, 1/m) \geq k).$$

For $\epsilon \leq 1$, writing $nk/m = k/(1+\epsilon)$ and applying a standard Chernoff bound gives

$$\mathbf{Pr}(\mathrm{Bin}(nk, 1/m) \geq k) \leq e^{-\epsilon^2 k/3(1+\epsilon)} = \beta^k$$

for $\beta = e^{-\epsilon^2/3(1+\epsilon)} < 1$. To extend the proof to all $\epsilon > 0$, we note that $\mathbf{Pr}(\mathrm{Bin}(nk, 1/m) \geq k)$ is decreasing in ϵ for any fixed k and n. □

Lemma 2.4 implies (1) for a sufficiently large constant α. Our approach for proving (2) is to first bound $\gamma(C_v)$, the cyclomatic number of a single connected component in G. We then use a stochastic dominance argument to obtain a result that holds for all connected components simultaneously.

Lemma 2.5. *For every vertex v and $t, k, n \geq 1$,*

$$\mathbf{Pr}(\gamma(C_v) \geq t \mid |C_v| = k) \leq \left(\frac{3e^5 k^3}{m}\right)^t.$$

Proof. We reveal the edges in C_v following a breadth-first search starting at v. That is, we first reveal all of the edges adjacent to v, then we reveal all of the edges of the form (u, w) where u is a neighbor of v, and so on, until all of C_v is revealed. Suppose that during this process, we discover that some node u is at distance i from v. Define $K(u)$ to be the number of edges that connect u to nodes at distance $i-1$ from v. In other words, $K(u)$ is the number of edges that connect u to the connected component containing v at the time that u is discovered by the breadth-first search. It is easy to see that $\gamma(C_v) = \sum_u \max\{0, K(u) - 1\}$. Again, we call the edges that contribute to $\gamma(C_v)$ excess edges. We bound $\gamma(C_v)$ by bounding $K(u)$ for each u.

Fix some $i \geq 1$, condition on the history of the breadth-first search until the point where all nodes with distance at most $i-1$ from v have been revealed, and suppose that the size of the connected component containing v at this point in time is at most k. Then any node u that is not currently in the connected component containing v could be reached from at most k distinct vertices in the component in the next step of the breadth-first search procedure. Thus $K(u)$ is stochastically dominated by the sum of k independent $\mathrm{Po}(\lambda/m^2)$ random variables, which has distribution $\mathrm{Po}(k\lambda/m^2)$. (Here we are using the property that throwing $\mathrm{Po}(\lambda)$ edges randomly into the bipartite graph is the same as setting the multiplicity of the edges to be independent $\mathrm{Po}(\lambda/m^2)$ random variables.) We chose λ such that $k\lambda/m^2 \leq k/m$, and so $\mathrm{Po}(k\lambda/m^2)$ is stochastically dominated by $\mathrm{Po}(k/m)$. We conclude that the number of excess edges incident on u that are revealed at this point in the breadth-first search and contribute to $\gamma(C_v)$ is stochastically dominated by the distribution $L(k) = \max(0, \mathrm{Po}(k/m) - 1)$. Furthermore, since the number of occurrences of each edge are independent, the joint distribution of the number of excess edges introduced at

this point in the breadth-first search is stochastically dominated by sum of m independent samples from $L(k)$. It follows that the distribution of $\gamma(C_v)$ given $|C_v| = k$ is stochastically dominated by the sum of mk independent samples from $L(k)$. We derive a tail bound on this distribution in Lemma 2.6 below, which yields the desired result.

Lemma 2.6. *Fix $k \leq 2m$, and let $X_1, \ldots, X_{mk}$ be independent random variables with common distribution $L(k)$, and let $X = \sum_{i=1}^{mk} X_i$. Then for every $t \geq 1$, we have $\mathbf{Pr}(X \geq t) \leq \left(\frac{3e^5k^3}{m}\right)^t$.*

Proof. First we bound the number of the X_i's that are greater than zero. For every i we have $\mathbf{Pr}(X_i > 0) \leq \mathbf{Pr}(\mathrm{Po}(k/m) \geq 2) \leq k^2/m^2$ (since for any $\mu > 0$, we have $\mathbf{Pr}(\mathrm{Po}(\mu) \leq 1) = e^{-\mu}(1+\mu) \geq (1-\mu)(1+\mu) = 1-\mu^2$). The number of positive X_i's is therefore stochastically dominated by the binomial distribution $\mathrm{Bin}(mk, k^2/m^2)$. Let $P = \{i : X_i > 0\}$ denote the set of positive X_i's. We have

$$\mathbf{Pr}(X \geq t) \leq \mathbf{Pr}[|P| > t] + \sum_{\ell=1}^{t} \mathbf{Pr}(|P| \geq \ell) \cdot \mathbf{Pr}\left(\sum_{i \in P} X_i \geq t \;\middle|\; |P| = \ell\right). \tag{3}$$

We bound the first term by

$$\mathbf{Pr}(|P| \geq t) \leq \binom{mk}{t}\left(\frac{k^2}{m^2}\right)^t \leq \left(\frac{mke}{t}\right)^t \left(\frac{k^2}{m^2}\right)^t = \left(\frac{ek^3}{mt}\right)^t. \tag{4}$$

For the second term, let $Y \sim \mathrm{Po}(k/m)$, and note that for every $j \geq 0$,

$$\begin{aligned}
\mathbf{Pr}(X_i = j+1 \mid i \in P) &= \mathbf{Pr}(Y = j+2 \mid Y \geq 2) \\
&\leq \frac{\mathbf{Pr}(Y = j+2)}{\mathbf{Pr}(Y = 2)} \\
&= \frac{2m^2e^{k/m}}{k^2}\mathbf{Pr}(Y = j+2) \\
&\leq \frac{2e^2m^2}{k^2}\mathbf{Pr}(Y = j+2).
\end{aligned}$$

Now let $Y_1 \dots Y_\ell$ be independent random variables with common distribution $\mathrm{Po}(k/m)$. Then

$$\begin{aligned}
\mathbf{Pr}\left(\sum_{i\in P} X_i \geq t \;\middle|\; |P| = \ell\right) &= \sum_{\substack{j_1,\dots,j_\ell \\ \sum_{i=1}^{\ell} j_i \geq t-\ell}} \prod_{i=1}^{\ell} \mathbf{Pr}(X_i = j_i + 1 \mid X_i > 0) \\
&\leq \left(\frac{2e^2m^2}{k^2}\right)^{\ell} \sum_{\substack{j_1,\dots,j_\ell \\ \sum_{i=1}^{\ell} j_i \geq t-\ell}} \prod_{i=1}^{\ell} \mathbf{Pr}(Y_i = j_i + 2) \\
&\leq \left(\frac{2e^2m^2}{k^2}\right)^{\ell} \mathbf{Pr}\left(\sum_{i=1}^{\ell} Y_i \geq t + \ell\right) \\
&= \left(\frac{2e^2m^2}{k^2}\right)^{\ell} \mathbf{Pr}(\mathrm{Po}(k\ell/m) \geq t + \ell) \\
&\leq \left(\frac{2e^2m^2}{k^2}\right)^{\ell} \left(\frac{ek\ell}{m(t+\ell)}\right)^{t+\ell} \\
&= \frac{1}{m^{t-\ell}} \left(\frac{2e^3\ell}{k(t+\ell)}\right)^{\ell} \left(\frac{ek\ell}{t+\ell}\right)^{t}, \qquad (5)
\end{aligned}$$

where we have used the tail bound [10, Theorem 5.4] in the fifth step.

Substituting (4) and (5) into (3) yields

$$\begin{aligned}
\mathbf{Pr}(X \geq t) &\leq \left(\frac{ek^3}{mt}\right)^{t} + \sum_{\ell=1}^{t} \left(\frac{ek^3}{m\ell}\right)^{\ell} \cdot \frac{1}{m^{t-\ell}} \left(\frac{2e^3\ell}{k(t+\ell)}\right)^{\ell} \left(\frac{ek\ell}{t+\ell}\right)^{t} \\
&= \left(\frac{ek^3}{mt}\right)^{t} + \left(\frac{ek}{m}\right)^{t} \sum_{\ell=1}^{t} \left(\frac{2e^4k^2}{t+\ell}\right)^{\ell} \left(\frac{\ell}{t+\ell}\right)^{t} \\
&\leq \left(\frac{ek^3}{mt}\right)^{t} + \left(\frac{ek}{m}\right)^{t} 2e^4k^2 \\
&\leq \left(\frac{ek^3}{m}\right)^{t} + \left(\frac{2e^5k^3}{m}\right)^{t} \\
&\leq \left(\frac{3e^5k^3}{m}\right)^{t},
\end{aligned}$$

completing the proof. □

Lemma 2.6 now implies Lemma 2.5. □

Combining Lemmas 2.5 and 2.4 we have that for any vertex v and constant $t \geq 1$,

$$\begin{aligned}
\mathbf{Pr}(\gamma(C_v) \geq t) &\leq \sum_{k=1}^{\infty} \mathbf{Pr}(\gamma(C_v) \geq t \mid |C_v| = k) \cdot \mathbf{Pr}(|C_v| \geq k) \\
&\leq \sum_{k=1}^{\infty} \left(\frac{3e^5k^3}{m}\right)^{t} \cdot \beta^k \\
&= O(n^{-t}) \quad \text{as } n \to \infty. \qquad (6)
\end{aligned}$$

Equation (6) gives a bound for the number of excess edges in a single connected component of $G(m, m, \mathrm{Po}(\lambda))$. We now extend this result to all connected components in order to show (2), which will complete the proof. The key idea is to show that instead of looking at all the connected components of G, we may take $2m$ *independent* samples of C_v. Intuitively one can see that the $\gamma(C_v)$ are negatively correlated across components and therefore should be dominated by independent copies. The reason is that due to the independence of edges in the Poisson distribution, if $u \notin C_v$ then the distribution of C_u conditioned on C_v is as if we sampled from a graph with the nodes of C_v removed, which is dominated by the distribution of C_u. We formalize this intuition in the following Lemma.

Lemma 2.7. *Fix some ordering $v_1, \ldots, v_{2m}$ of the vertices. For $i = 1, \ldots, 2m$, let $C'_{v_i} = C_{v_i}$ if v_i is the first vertex in the ordering to appear in C_v, and let C'_{v_i} be the empty graph on the $2m$ vertices otherwise. Let $C''_{v_1}, \ldots, C''_{v_m}$ be independent random variables such that each C''_{v_i} is distributed as C_{v_i}. Then $(C''_{v_1}, \ldots, C''_{v_{2m}})$ stochastically dominates $(C'_{v_i}, \ldots, C'_{2m})$.*

Proof. We prove the result via the technique of coupling (c.f. [10, §11]). We show there exists a coupling where the C''_{v_i}'s have the appropriate joint distribution and C'_{v_i} is a subgraph of C''_{v_i} for $i = 1, \ldots, 2m$. We show how to sample the relevant random variables so that all of the required properties are satisfied. First, we simply sample C_{v_1} and set $C''_{v_1} = C'_{v_1} = C_{v_1}$. Then, for the smallest i such that $v_i \notin C_{v_1}$, we set $C'_{v_2}, \ldots, C'_{v_{i-1}}$ to be the empty graph and sample $C'_{v_i} = C_{v_i}$ according to the appropriate conditional distribution. Note that this conditional distribution can be represented as the connected component containing v_i in a sample from a distribution over bipartite graphs with vertex set $\{v_1, \ldots, v_{2m}\} - C_{v_1}$, where for each left vertex u and right vertex v, the multiplicity of the edge (u, v) is a $\mathrm{Po}(\lambda/m^2)$ random variable, and these multiplicities are independent. This conditional distribution is stochastically dominated by C_{v_i}, and therefore we can sample C_{v_i} according to the correct conditional distribution and simultaneously ensure that C_{v_i} is a subgraph of C''_{v_i} and that the distribution of C''_{v_i} is not affected by conditioning on C''_{v_1}. We continue in this way, until we have sampled all of the C'_{v_i}'s. At this point, we have that C'_{v_i} is a subgraph of C''_{v_i} for every i such that C'_{v_i} is not the empty graph, and we have not yet sampled the C''_{v_i}'s for which C'_{v_i} is the empty graph. To complete the construction, we simply sample these remaining C''_{v_i}'s independently from the appropriate distributions. □

Now let $B_1, \ldots, B_{2m}$ be independent samples, each distributed as $\gamma(C_v)$. By Lemma 2.7, we have that $\gamma(G) - T(G)$ is stochastically dominated by $\sum_{i=1}^{2m} B_i - |\{i : B_i \geq 1\}|$. Applying (6) now

implies that there exists a constant $c \geq 1$ such that for sufficiently large n,

$$\begin{aligned}
\mathbf{Pr}((\gamma - T)(G) \geq s) &\leq \mathbf{Pr}\left(\sum_{i=1}^{2m} B_i \geq s + |\{i : B_i \geq 1\}|\right) \\
&\leq \sum_{\substack{j_1,\ldots,j_{2m} \\ \sum_{i=1}^{2m} j_i = s}} \prod_{\substack{i=1,\ldots,2m \\ j_i \geq 1}} \mathbf{Pr}(B_i \geq j_i + 1) \\
&= \sum_{\substack{j_1,\ldots,j_{2m} \\ \sum_{i=1}^{2m} j_i = s}} \prod_{\substack{i=1,\ldots,2m \\ j_i \geq 1}} cn^{-j_i - 1} \\
&\leq \sum_{\substack{j_1,\ldots,j_{2m} \\ \sum_{i=1}^{2m} j_i = s}} c^s n^{-s-|\{i : j_i \geq 1\}|} \\
&= \sum_{k=1}^{2m} \sum_{\substack{j_1,\ldots,j_{2m} \\ \sum_{i=1}^{2m} j_i = s \\ |\{i : j_i \geq 1\}| = k}} c^s n^{-s-k} \\
&\leq \sum_{k=1}^{2m} \binom{2m}{k} k^s c^s n^{-s-k} \\
&\leq \sum_{k=1}^{2m} \left(\frac{2(1+\epsilon)ne}{k}\right)^k k^s c^s n^{-s-k} \\
&= n^{-s} c^s \sum_{k=1}^{2m} \left(\frac{2e(1+\epsilon)}{k}\right)^k k^s \\
&= O(n^{-s}).
\end{aligned}$$

Here the final summation and the c^s term are all $O(1)$ when s is a constant. This establishes (2), completing the proof of Theorem 2.1.

A remark on deletions: The previous analysis handled insertions only. One of the attractive properties of cuckoo hashing is that deletions take $O(1)$ in the worst case, and require only that the item deleted be removed from its current position. Such a naive implementation is not possible when a stash is added. To see this consider a component C with two excess edges x and y, and assume that x is in the stash and y isn't. Now if y is naively deleted then $C \setminus S$ is a tree, but C is not a tree violating Lemma 2.2. One way to fix this is to try and insert all elements of the stash upon each deletion. This approach no longer ensures a constant-time deletion operation. One way to preserve worst-case constant-time deletion is to charge subsequent insertion operations for these attempts to place stash items back in the main hash table. In other words, deletions are done naïvely in constant worst case time, but when inserting a new item x we first try to insert all elements in the stash, and then insert x. Now, since the stash is likely to be small, in fact with probability $1 - \Theta(1/n)$ it is empty, the insertion still takes expected constant time. In particular, the additional expected time invested in inserting the items in the stash at each insertion is at most $\mathbf{E}[|S|] \cdot \alpha \log n < 1$. One can devise other similar methods to trade off the effort of placing stash items back in the hash table.

3 Generalized Cuckoo Hashing

We now turn our attention to the generalized cuckoo hashing scheme proposed by Fotakis et al. [4]. Here, we attempt to insert n items into a table with $(1+\epsilon)n$ buckets and d hash functions (assumed to be independent and fully random), for some constant $\epsilon > 0$. Increasing the number of hash functions increases the query time, but also reduces the space required. Thus, the main object of research in [4] is the tradeoff between ϵ and d.

We think of the hash functions as defining a distribution $\mathcal{G}(n, \epsilon, d)$ over bipartite multi-graphs, which is sampled by creating n left vertices representing items, denoted by L, and $(1+\epsilon)n$ right vertices representing memory slots, denoted by R. Each $v \in L$ samples uniformly and independently d right vertices to which it connects, which represents its d hash locations. We think of a partial placement of the items into the hash locations as a matching in the graph. An L-perfect matching represents a complete assignment of the items. In this setting, a stash of size s could be represented by adding s vertices to the right hand side, and connecting each left hand side vertex to all of the s stash vertices. We denote the distribution over such graphs as $\mathcal{G}(n, \epsilon, d, s)$. We start by proving that a constant-sized stash can dramatically decrease the probability the graph does not contain an L-perfect matching. Generalizing Lemma 1 from [4] we have:

Lemma 3.1. *Given a constant $\epsilon \in (0,1)$, for any integer $d \geq 2(1+\epsilon)\ln\left(\frac{e}{\epsilon}\right)$, and integer $s \geq 0$, the bipartite graph G sampled from $\mathcal{G}(n, \epsilon, d, s)$ contains an L-perfect matching with probability at least $1 - O(n^{(1-d)(s+1)})$ where the constant in the O-notation depends on s and d.*

We note that for a given hash table size, the probability a matching doesn't exist decreases rapidly with d. Increasing d comes at a cost of increasing the query complexity. The stash introduces an arguably cheaper and simpler way to decrease the error probability.

Proof. The proof follows closely the proof of Lemma 1 in [4]. By Hall's theorem it is enough to bound the probability each set $X \subseteq L$ has at least $|X|$ neighbors. We note that if $|X| \leq s$ then the stash nodes alone provide the necessary $|X|$ neighbors. Let $P(k)$ denote the probability there exists a set of size k in L with fewer than k neighbors. Of these k neighbors, s can correspond to the stash, leaving at most $k-s-1$ other vertices from R. Hence we can bound $P(k)$ by

$$P(k) \leq \binom{n}{k}\binom{(1+\epsilon)n}{k-s-1}\left(\frac{k-s-1}{(1+\epsilon)n}\right)^{dk} \leq \frac{e^{2k}k^{(d-2)k}}{n^{(d-2)k-(s+1)}}.$$

The individual terms correspond to the ways of choosing k vertices on the left, the ways of choosing $k-s-1$ vertices on the right, and the probability that the k vertices on the left have all their choices land on the $k-s-1$ vertices on the right.

It is easy to see that for $2 \leq k \leq \frac{n}{e^{d/(d-2)}}$ the right hand side is non-increasing in k, and further that it decreases by a factor of at least $1/2$ for the first $2\log n$ terms for sufficiently large n. Furthermore, following [4], we find that for $n \geq k \geq \frac{n}{e^{d/(d-2)}}$ and the above choice of d, it holds that $P(k)$ is exponentially small in n. The lemma is now obtained by taking the union bound for $s+1 \leq k \leq n$, and noting that the asymptotics are dominated by the term when $k = s+1$. □

The following lemma shows that the bound on d is essentially tight. It generalizes Lemma 2 in [4].

Lemma 3.2. *For every constant $\epsilon > 0$, $s \geq 0$, and $d \leq (1+\epsilon)\ln\left(\frac{1+\epsilon}{2(\epsilon+s/n)}\right)$, the probability that a sample G from $\mathcal{G}(n,\epsilon,d)$ contains a matching of size $n-s$ is $2^{-\Omega(n)}$.*

Proof. We bound the probability p that G has at most $\epsilon n+s$ isolated right vertices, since otherwise a maximal matching in G has size less than $n-s$. This probability p is the same as the probability that throwing nd balls randomly into $(1+\epsilon)n$ bins yields at most $\epsilon n+s$ empty bins. By a standard Poisson approximation lemma (e.g. [10, Theorem 5.10]), we have

$$\begin{aligned} p &\leq 2\mathbf{Pr}\left(\mathrm{Bin}\left((1+\epsilon)n, \mathbf{Pr}\left(\mathrm{Po}\left(\frac{nd}{(1+\epsilon)n}\right)=0\right)\right) \leq \epsilon n+s\right) \\ &= 2\mathbf{Pr}\left(\mathrm{Bin}\left((1+\epsilon)n, e^{-d/(1+\epsilon)}\right) \leq \epsilon n+s\right). \end{aligned}$$

Now,

$$\mathbf{E}\left[\mathrm{Bin}\left((1+\epsilon)n, e^{-d/(1+\epsilon)}\right)\right] = (1+\epsilon)ne^{-d/(1+\epsilon)} \geq 2(\epsilon n+s),$$

and so Hoeffding's inequality gives

$$p \leq 2\exp\left[-\frac{(\epsilon+s/n)^2 n}{2(1+\epsilon)}\right] = 2^{-\Omega(n)},$$

as required. □

3.1 The Insertion Algorithm

For a bipartite graph G and a matching M in G, we let G_M denote a directed version of G where an edge e is oriented from right to left if $e \in M$, and e is oriented from left to right if $e \notin M$. Let F denote the set of vertices in R that represent unoccupied memory locations. When inserting an element v to L, we need to find a free memory location $\ell \in F$ for which there is a path $v \rightsquigarrow \ell$ in G_M. The number of nodes scanned when finding such a path determines the complexity of the insertion algorithm. There may be many different ways to perform this search, resulting in different insertion procedures. Fotakis et al. analyze the case where the insertion procedure uses a Breadth First Search[3]. They show that assuming the hash functions are truly random, the expected time and space needed for insertion is constant. Furthermore, the probability that the search takes time larger than n^β, for $\beta \in (0,1)$ is bounded by roughly $n^{-\beta d}$. We show that when using a stash of size s, the probability an insertion takes more than n^β falls roughly like $n^{-ds\beta/\log d}$ where the logarithm is taken with base 2.

Let it be the case that the insertion procedure uses BFS and when inserting item x, x is put in the stash if more than cn^β nodes have been scanned, for some $c > 0$ to be chosen later and $\beta \in (0,1)$. We prove the following generalization of Theorem 1 in [4]:

Theorem 3.1. *For any positive $\epsilon < \frac{1}{5}$ and integer $d \geq 5 + 3\ln(1/\epsilon)$, the insertion algorithm takes $(1/e)^{O(\ln d)}$ expected time per left vertex to maintain an L-perfect matching. Moreover, the probability the stash size exceeds $2s$ is at most $O(n^{\frac{(1-d)(s+1)}{2}} + n^{-\frac{\beta ds}{\log d}+s})$.*

[3]We remark that in practice it is probably better to use some sort of random walk. Analyzing more practical insertion procedures is an interesting open problem.

The proof closely follows [4]. Here we use the stash both to store items that possibly could not otherwise be placed (improving the probability that a matching exists) and to ensure that fewer than cn^β nodes are scanned. We use s of the $2s$ slots of the stash for each task; while we do not do it here, the analysis could be easily optimized further to trade off these two effects as needed.

Proof. Let $F \subseteq R$ be the set of unoccupied slots. When inserting v, the depth of the BFS tree is determined by the distance between v and F in G_M. Denote by Y_i the set of vertices in R with distance at most $2i$ from F in G_M, thus $Y_0 = F$. We now quote Lemma 5 from [4]:

Lemma 3.3. *Let G be sampled from $\mathcal{G}(n, \epsilon, d)$ and let $\lambda^* = \lceil \log_{\frac{4}{3}} \left(\frac{1}{2\epsilon}\right) \rceil$. With probability $1 - 2^{-\Omega(n)}$ it holds that $|Y_{\lambda^*}| \geq (1/2 + \epsilon)n$.*

Note that adding s stash vertices to R may only increase $|Y_{\lambda^*}|$, thus the lemma also holds for graphs sampled from $\mathcal{G}(n, \epsilon, d, s)$. Note that Lemma 3.3 implies that the probability a node $v \in L$ is at distance more than $2\lambda^* + 1$ from Y_0 is at most 2^{-d}. In order to obtain a bound on higher distances we need to track the growth of Y_i for $i > \lambda^*$. This is done via the following lemma, which generalizes Lemma 6 in [4]:

Lemma 3.4. *Let $d \geq 8$ and $s \geq 0$ be integers and sample G from $\mathcal{G}(n, \epsilon, d, s)$. Then, with probability at least $1 - O(n^{\frac{(1-d)(s+1)}{2}})$, it holds that any set $Y \subseteq R$, $|Y| \geq (\frac{1}{2} + \epsilon)n$, has at least $\frac{(1+\epsilon)n - |Y|}{2}$ neighbors.*

Proof. The Lemma follows immediately from the following lemma, generalizing Lemma 7 in [4].

Lemma 3.5. *Let $d \geq 8$ and $s \geq 0$ be integers, and let G be sampled from $\mathcal{G}(n, \epsilon, d, s)$. Then, with probability at least $1 - O(n^{\frac{(1-d)(s+1)}{2}})$ it holds that any set $X \subseteq L$, $|X| \leq \frac{n}{4}$ has at least $2|X|$ neighbors.*

Proof. For a fixed integer $k \leq n/4$ let $P(k)$ be the probability that there exisits a set of left vertices of size k which does not expand by a factor of 2. Note that $P(k) = 0$ for $0 \leq k \leq \frac{s+1}{2}$ trivially. For $k \geq \frac{s+1}{2}$, we have

$$P(k) \leq \binom{n}{k}\binom{(1+\epsilon)n}{2k-s-1}\left(\frac{2k-s-1}{(1+\epsilon)n}\right)^{dk} \leq \left(\frac{e^3 2^{d-2} k^{d-3}}{(1+\epsilon)^{d-2} n^{d-3}}\right)^k n^{-s-1}$$

which is non-increasing for $k \leq n/\alpha$ for some α. The right-hand side depends only on ϵ and d, and further it decreases by a factor of at least $1/2$ for the first $2 \log n$ terms for sufficiently large n. In [4] it is shown that for the choice of d in the lemma, for every α independent of n, $P(k)$ is exponentially small in n for $k \geq n/\alpha$. The lemma is obtained by union bounding for $\frac{s+1}{2} < k \leq n/4$. The lemma is obtained by taking the union bound for $\frac{s+1}{2} < k \leq n/4$, and noting that the asymptotics are dominated by the term with the smallest value of k. □

Lemma 3.5 now implies Lemma 3.4. To reach a contradiction assume that for some integer $k \leq n/4$ there is a set $y \subseteq r$, $|y| = (1+\epsilon)n - 2k$ with less than $n - k$ neighbors. denote by $x \subseteq l$ the vertices outside y's neighborhood, so $|x| \geq k$. Now Lemma 3.5 implies that x has at least $2k$ neighbors which must be disjoint from y, which implies that $|r| > (1+\epsilon)n$, hence the contradiction. □

We can now bound $|Y_i|$ for larger values of i. We quote Lemma 8 from [4]:

Lemma 3.6. *Let $\lambda^* = \lceil \log_{\frac{4}{3}} \left(\frac{1}{2\epsilon}\right) \rceil$. If G satisfies the conclusions of Lemma 3.3 and Lemma 3.4, then for any integer $\lambda \geq 0$ it holds that $|Y_{\lambda^*+\lambda}| \geq n(1 - 2^{-(\lambda+1)})$.*

Let v be a newly arrived left vertex and denote by T_v the number of nodes in R scanned when inserting v. If G satisfies the conclusions of Lemma 3.3 and Lemma 3.6, then T_v is determined by the distance of v from nodes in F. In particular, if one of v's (assumed to be random) neighbors is in Y_i then at most d^{i+1} nodes are scanned before a node in F is discovered. We conclude that

$$\mathbf{Pr}(T_v \geq d^{i+1}) \leq \left(1 - \frac{|Y_i|}{(1+\epsilon)n}\right)^d.$$

Let $\lambda^* = \lceil \log_{\frac{4}{3}} \left(\frac{1}{2\epsilon}\right) \rceil$. A simple calculation shows that if G satisfies the conclusions of Lemma 3.3 and Lemma 3.6, then for any $\beta \in (0,1)$ it holds that

$$\begin{aligned}\mathbf{Pr}[T_v > d^{\lambda^*+1} n^\beta] &= \mathbf{Pr}[T_v > d^{\lambda^*+1+\beta \log_d n}] \\ &\leq \left(1 - \frac{|Y_{\lambda^*+\beta \log_d n}|}{(1+\epsilon)n}\right)^d \\ &\leq 2^{-\beta d \log_d n} = n^{-\beta d / \log d}.\end{aligned}$$

The probability that out of the n items there are s such nodes is at most

$$\binom{n}{s} n^{-\frac{\beta d s}{\log d}} \leq n^{-\frac{\beta d s}{\log d} + s}.$$

Now we sample G from $\mathcal{G}(n, \epsilon, d, s)$ and add a stash of size s thus obtaining a stash of total size $2s$. The union bound of the above with the error probabilities of Lemma 3.3 and Lemma 3.4 yields the high probability bound of Theorem 3.1. The bound on the expectation follows directly from [4]. □

4 A Variant with Multiple Items per Bucket

The last scheme we consider in this work is *blocked cuckoo hashing* proposed by Dietzfelbinger and Weidling [2]. Here we attempt to insert n items into a table with $m = (1+\epsilon)n/d$ buckets using two hash functions h_1 and h_2, for some constants $\epsilon > 0$ and integer $d > 0$. Each bucket can hold at most d items. Note that here, following the notation of [2], d is the capacity of a bucket, not the number of hash functions. Here also we assume that the hash functions are independent and fully random.

As before, we think of the hash functions as defining a multi-graph G with n left vertices representing the items and m right vertices representing buckets. As before we connect each left vertex to its two potential buckets. We say the hash functions are *s-suitable* if $n - s$ nodes could be matched to buckets such that no bucket is matched to more than d items.

The following proposition generalizes Theorem 1 from [2].

Proposition 4.1. *For any $\epsilon > 0$, $d \geq 1 + \frac{\ln(1/\epsilon)}{1-\ln 2}$, $s \geq 0$, the probability two uniform, independent hash function are s-suitable is $1 - O(m^{(s+1)(1-d)})$.*

Proof. Theorem 1 in [2] establishes the case where $s = 0$, and so we assume that $s \geq 1$. The proof is now obtained by some simple modifications to the proof in [2, Section 3.2]. First, we change the definition of F to $F = \mathbf{Pr}(\exists X \subseteq S, |X| \geq s+1 : \Gamma(X) \leq |X|/d)$. The inequality [2, (10)] then becomes $F \leq \sum_{s+1 \leq j \leq m/(1+\epsilon)} F(j)$. Case 1 is now no longer necessary, and the proofs of Case 3 and Case 4 can be left as they are. Case 2 becomes $s+1 \leq j \leq m/(2e^4)$. The inequality [2, (18)] then becomes

$$\sum_{s+1 \leq j \leq m/(2e^4)} F(j) = O(f(s+1,0)) = O\left(\left(\left(\frac{s+1}{m}\right)^{d-1} e^{d+1}\right)^{s+1}\right) = O(m^{(s+1)(1-d)}),$$

completing the proof. □

The following lower bound generalizes Proposition 1 in [2].

Proposition 4.2. *For sufficiently small $\epsilon > 0$ and any constant $s \geq 0$, if $6 + 5\ln d + d < \frac{\ln(1/\epsilon)}{1-\ln 2}$, then with high probability (as $n, m \to \infty$) two random hash functions are not $s-$suitable for the items.*

Proof. We observe that the proof of Proposition 1 in [2] suffices with the obvious modifications. For the convenience of the reader we sketch it here, quoting freely from [2]. For bucket y define p_{d-1} to be the probability that exactly $d-1$ of the $2n$ random hash values equals y. We are interested in the expected number of buckets with this property. Inequality [2, (7)] states that

$$mp_{d-1} \geq m\left((2/e)^{-d}/5.2\sqrt{d}\right)(1 - O(1/n)).$$

If this expected value is larger by a constant factor than $\epsilon n + s$ (e.g., $mp_{d-1} \geq 1.05(\epsilon n + s)$), then with probability $1 - 2^{-\Omega(n)}$ more than $\epsilon n + s$ buckets are hit by exactly $d-1$ hash values. (This follows by applying standard tail inequalities like the Azuma-Hoeffding inequality.) These buckets can not be full in any allocation of $n - s$ items to the buckets, implying that it is impossible to place $n - s$ items into the buckets.

By [2, (6)], a sufficient condition for this to happen is:

$$\frac{(2/e)^d}{5.2\sqrt{d}} \cdot m > 1.05(\epsilon n + s),$$

which, for sufficiently large n, is satisfied if

$$\frac{(2/e)^d}{5.2\sqrt{d}} \cdot m > 1.1\epsilon n.$$

The inequality above is identical to [2, (8)]. It is shown there that substituting $dm = (1+\epsilon)n$ and rearranging completes the proof. □

4.1 The Insertion Algorithm

In order to describe the insertion algorithm we modify the graph in two ways. We split the nodes that represent the buckets into their d slots. In other words, the graph now has n left nodes representing the items and $(1+\epsilon)n$ right nodes representing the $m \cdot d$ slots available. Each left

node is connected to its $2d$ potential locations. As in the previous section, the placement of the items is expressed as a matching M where the edges of M are oriented from right to left and all other edges are oriented from left to right.

The nodes are inserted one by one, so at time t there is a set L_t of t nodes on the left side and a matching M_t that connects some of the nodes in L_t to the right side. The nodes in L_t that are unmatched correspond to the items in the stash. Call the graph at this time G_t. The arrival of an item corresponds to the addition of a new node x to set on the left, with all its edges oriented to the right. The insertion algorithm searches for an empty slot by exploring the graph $G_t \cup \{x\}$ for a path from x to an empty slot. We call such a path an *augmenting path*. If an augmenting path $x \rightsquigarrow y$ is found then a new matching M_{t+1} which includes x is obtained by having all the edges along the path flip their orientation and $|M_{t+1}| = |M_t| + 1$. If there is no such path then x is placed in the stash and $M_{t+1} = M_t$. The exact algorithm by which the graph is explored is not important, it could be BFS or DFS or any other procedure that is guaranteed to find a path if one exists.

Theorem 4.1. *If for some $s > 0$ the hash functions are s-suitable for the set of the items, then the procedure above puts at most s items in the stash.*

Proof. We prove the theorem by induction on the time t. Denote by ℓ_t the minimum number such that L_t is ℓ_t-suitable. Denote by s_t the number of item in the stash at time t. It is enough to show that $\ell_t = s_t$.

For $t = 0$ this is trivially true. Inductively assume that $\ell_t = s_t$ and now a new item x arrived. If x is inserted successfully into the table then $s_{t+1} = s_t = \ell_t$. Now, since by definition $\ell_t \leq \ell_{t+1} \leq s_{t+1}$ it holds that $\ell_{t+1} = s_{t+1}$. Consider the case that x is not successfully inserted into the table and is put in the stash, we claim that $\ell_{t+1} = \ell_t + 1$. To see this consider a modified search algorithm which when searching for a location may also evict nodes from the stash. Such an algorithm could be modeled by modifying the graph as follows: s_t stash nodes are added to the right side of the graph and every node in L_t is connected to the stash nodes with the edges oriented accordingly. Call the matching in this graph M'_t. Observe that M'_t is the union of M_t with the edges corresponding to the items in the stash. Now we claim that if this modified insertion algorithm fails then $\ell_{t+1} = \ell_t + 1$. To see this observe that the modified graph is exactly the residual graph in the flow based maximum matching algorithm, so if there is no augmenting path in the graph then M'_t remains a maximum matching even after x was added.

The only thing that remains to show is that an augmenting path in the modified graph does not touch the stash nodes. To see this consider such an augmenting path $x \rightsquigarrow y$. Let z be the last stash node in this path. Observe that by flipping the edges in the portion of the path $z \rightsquigarrow y$ we obtain a matching in G_t with one less element in the stash violating the induction hypothesis that $s_t = \ell_t$. □

The Insert time: Our analysis shows that if the stash is of fixed size s, the probability of a failure that requires rehashing is $1 - O(m^{(s+1)(1-d)})$. We claim nothing regarding the running time of the insert algorithm itself. In [2] it is shown that when using BFS the expected insertion time is constant. Their analysis holds trivially in our case as long as the stash remains empty. Obtaining better bounds for the insertion time and analyzing other insertion algorithms remains open even without considering a stash.

5 Some Simple Experiments

In order to demonstrate the potential importance of our results in practical settings, we present some simple experiments. We emphasize that these experiments are not intended as a rigorous empirical study; they are intended only to be suggestive of the practical relevance of the general stash technique. First, we consider using a cuckoo hash table with $d = 2$ choices, consisting of two sub-tables of size 1200. We insert 1000 items, allowing up to 100 evictions before declaring a failure and putting the last item evicted into the stash. In this experiment we allow the stash to hold as many items as needed; the number of failures gives the size the stash would need to be to avoid rehashing or a similar failure mode. In our experiments, we use the standard Java pseudorandom number generator to obtain hash values.

The results from one million trials are presented in Table 1(a). As expected, in most cases, in fact over 99% of the time, no stash is needed. The simple expedient of including a stash that can hold just 4 items, however, appears to reduce the probability for a need to rehash to below 10^{-6}. A slightly larger stash would be sufficient for most industrial strength applications, requiring much less memory than expanding the hash table to achieve similar failure rates.

Stash Size	Number of Trials
0	992812
1	6834
2	338
3	17
4	1

(a) 1000 items

Stash Size	Number of Trials
0	9989861
1	10040
2	97
3	2
4	0

(b) 10000 items

Table 1: For $d = 2$ and for every stash size we count the number of trials for which the stash size was necessary and sufficient. Table 1(a) provides results for 10^6 trials for 1000 items where the maximum required stash size is four. Table 1(b) provides results for 10^7 trials for 10000 items where the maximum required stash size is three.

We show similar results for placing 10000 items using $d = 2$ choices with two sub-tables of size 12000 in Table 1(b). Here we use 10^7 trials in order to obtain a meaningful comparison. The overall message is the same: a very small stash greatly reduces the probability that some item cannot be placed effectively.

We also conduct some simple experiments for the case $d = 3$. Here, rather than considering the breadth-first search extension of cuckoo hashing analyzed in Section 3, we examine the more practical *random walk* variant introduced in [4]. We use d equally sized tables, which we think of as arranged from left to right, with one hash function per table. To insert an item, we check whether any of its hash locations are unoccupied, and in that case, we place it in its leftmost unoccupied hash location. Otherwise, we place it in a *randomly* chosen hash location and evict the item x in that place, repeat the same procedure for x and so on. We continue in this way until an item is successfully placed in the table or until some prespecified upper-bound on the number of evictions had been reached, in which case we place the last item evicted into the stash.

In each trial we insert 1000 items, each of the three tables is of size 400, and the maximum number of evictions is 100. As before, we perform 10^6 trials. Similarly, for 10000 items, we consider

Stash Size	Number of Trials
0	998452
1	1537
2	11

(a) 1000 items

Stash Size	Number of Trials
0	9964148
1	35769
2	83

(b) 10000 items

Table 2: For $d = 3$, successes at each stash size measured over 10^6 trials for 1000 items (a), and measured over 10^7 trials for 10000 items (b).

the same experiment with three tables of size 4000, and perform 10^7 trials. The results are displayed in Table 2, and are analogous to those in Table 1.

6 Conclusion

We have shown how to greatly improve the failure probability bounds for a large class of cuckoo hashing variants by using only a constant amount of additional space. Furthermore, our proof techniques naturally extend the analysis of the original schemes in a straightforward way, suggesting that our techniques will continue to be broadly applicable for future hashing schemes. Finally, we have also presented some simple experiments demonstrating that our improvements have real practical potential.

There remain several open questions. The analysis of random-walk variants when $d > 2$, in place of breadth-first-search variants, remains open both with and without a stash. A major open question is proving the above bounds for *explicit* hash families that can be represented, sampled, and evaluated efficiently. Such explicit constructions are provided for standard cuckoo hashing in [12] and [3]. It would be interesting to improve upon those constructions and extend them to the case of a stash.

Acknowledgment

We thank Martin Dietzfelbinger for pointing out some errors and simplifications in a previous version of the paper. We thank the reviewers for helpful suggestions and improvements. The last author is grateful to Thomas Holenstein for useful discussions.

References

[1] L. Devroye and P. Morin. Cuckoo Hashing: Further Analysis. *Information Processing Letters*, 86(4):215-219, 2003.

[2] M. Dietzfelbinger and C. Weidling. Balanced Allocation and Dictionaries with Tightly Packed Constant Size Bins. *Theoretical Computer Science*, 380(1-2):47-68, 2007.

[3] M. Dietzfelbinger and P. Woelfel. Almost Random Graphs with Simple Hash Functions. In *Proceedings of the Thirty-Fifth Annual ACM Symposium on Theory of Computing (STOC)*, pp. 629-638, 2003.

[4] D. Fotakis, R. Pagh, P. Sanders, and P. Spirakis. Space Efficient Hash Tables With Worst Case Constant Access Time. *Theory of Computing Systems*, 38(2):229-248, 2005.

[5] Svante Janson, Tomasz Łuczak and Andrzej Ruciński. *Random Graphs*. Wiley, 2000.

[6] A. Kirsch and M. Mitzenmacher. The Power of One Move: Hashing Schemes for Hardware. To appear in *Proceedings of the 27th IEEE International Conference on Computer Communications* (INFOCOM), 2008. Temporary version available at `http://www.eecs.harvard.edu/~kirsch/pubs/`.

[7] A. Kirsch and M. Mitzenmacher. Using a Queue to De-amortize Cuckoo Hashing in Hardware. In *Proceedings of the Forty-Fifth Annual Allerton Conference on Communication, Control, and Computing*, 2007.

[8] A. Kirsch, M. Mitzenmacher, and U. Wieder. More Robust Hashing: Cuckoo Hashing with a Stash. To appear in *Proceedings of the 16th Annual European Symposium on Algorithms* (ESA), 2008. Temporary version available at `http://www.eecs.harvard.edu/~kirsch/pubs/`.

[9] R. Kutzelnigg. Bipartite Random Graphs and Cuckoo Hashing. In *Proceedings of the Fourth Colloquium on Mathematics and Computer Science*, 2006.

[10] M. Mitzenmacher and E. Upfal. *Probability and Computing: Randomized Algorithms and Probabilistic Analysis*. Cambridge University Press, 2005.

[11] M. Naor, G. Segev, and U. Wieder. History Independent Cuckoo Hashing. Manuscript, 2008.

[12] R. Pagh and F. Rodler. Cuckoo Hashing. *Journal of Algorithms*, 51(2):122-144, 2004.

[13] Vijay V. Vazirani. *Approximation Algorithms*. Springer, 2004.