CS 223 Final Project

CuckooRings: A Data Structure for Reducing Maximum Load in Consistent Hashing

Jonah Kallenbach and Ankit Gupta

May 2015

1 Introduction

Cuckoo hashing and consistent hashing are different hashing schemes for different use cases, both of which have been implemented in numerous recent systems [1, 2, 3, 4, 5]. Cuckoo hashing, originally developed by Rasmus Pagh and Flemming Friche Rodler in 2001 [6], is a hashing scheme which provides $O(1)$ worst case look-up time and $O(1)$ expected amortized insertion time, by allowing keys to be stored in multiple locations, so if one slot is full, the other can be filled with that key. If the second slot is full, we evict the key in that slot, and repeat the process, usually up to some predefined limit, where if we exceed that limit, we rehash everything. In the setting of a distributed system, this step, the expansion of the hash table (or picking of new hash functions) and rehashing of all keys, is often utterly impractical. For these settings, hashing structures have been invented that allow only small amounts of rehashing when the structure of the buckets changes. Here we consider the idea of consistent hashing that was introduced by Karger et al. in 1997 [7]. In consistent hashing, we define some key-space K in which the servers or buckets (we will use the two terms interchangeably here) will live. To find the server associated with a particular key, we hash the key into the ring, and walk clockwise or counter-clockwise (an implementation could pick either direction as long as it is that way everywhere) along the ring until the first bucket index we reach. This has the key benefit that if a particular server fails or a new server is added, we rehash roughly $1/N$ of the keys, where N is the number of servers, instead of all of them.

In this project, we sought to implement a new data structure, which is an application of some of the ideas from the field of cuckoo hashing to consistent hashing. Most crucially, we maintain two consistent hashing rings, and if a particular server has a load above some predefined threshold, we rehash its load into the other ring. This provides an interesting trade-off, one which usually resolves in favor of our data structure: if this threshold load is made too low, one can set off a series of rehashes back and forth (we simply stop after some predefined number of cycles), which significantly slows down insertion, but if it

is too high, we are not really helping relieve the max load on the servers at all. In the ideal case, with minimal slowdown, we dramatically improve the load distribution over the servers.

2 Terminology

- **Server**: To be used interchangeably with bucket. A major use-case of the proposed data structure will be for allocation of jobs to servers in systems applications. However, in the general case, this data structure can be applied to wider problems, and so servers should be thought of as buckets.
- **Job**: A job should be thought of as a key that is going to be hashed. Thus, jobs get put into servers (as keys get hashed to buckets).
- **N**: number of servers or buckets
- **M**: number of keys or jobs being hashed.
- **Load**: The load of the ith bucket is the number of keys hashed to the ith bucket.
- **RingHash**: Our implementation of consistent hashing.

3 Consistent Hashing

3.1 Theory

Consistent hashing is a formulation of hashing that allows for the removal or addition of buckets with only small amounts of rehashing and no need to change the hash function. Consider a ring, with points along the ring representing the integers from 0 to 2^{32} (where 0 and 2^{32} are both at the top of the circle). Buckets are located at (ideally evenly spread) points along the circle. When hashing a key, you hash it to number between 0 and 2^{32}, which corresponds to some point on the circle. Then, you follow the circle around until you reach the next bucket.

The next section will delve more deeply into the systems applications of this simple yet powerful algorithm. However, there are a number of key theoretical results:

Theorem 1. *For any set of N nodes and K keys, with high probability:*

1. *Each node is responsible for $O((1+\epsilon)\frac{K}{N})$ keys.*
2. *When the $(N+1)$st server joins or leaves the network, responsibilities for $O(K/N)$ keys change to or from the joining or leaving node.*

Note: ϵ was originally shown to be $O(\log N)$, but can be reduced by using more complex schemes. This is the critical benefit of consistent hashing: the removal of buckets leads to only a small fraction of the total number of keys needing to be rehashed, and they can be rehashed very easily (just to the next point on the circle). Similarly, it is easy to add buckets to the structure. This makes it effective for the allocation of jobs to servers, as it allows for changing numbers of servers (as they go online and offline), with only a limited amount of necessary rehashing when this happens.

One downside of consistent hashing, however, is that it is susceptible to an uneven distribution of keys over the buckets. While a good hash function will help to alleviate this, we can still get buckets that have many more keys than others, in part because of the rehashing of keys that occurs when a bucket (server) is removed.

Our data structure will incorporate aspects of another hashing strategy, developed around the same time as consistent hashing but for very different purposes, called cuckoo hashing, in order to address some of these problems. The results section contains various metrics about our implementation of consistent hashing (RingHash), compared to our proposed data structure.

3.2 Our implementation

As a precursor to implementing our proposed data structure, we began by implementing the Consistent Hashing data structure, called RingHash. In the later sections of this paper, we discuss the performance of this implementation relative to our proposed data structure.

We implemented RingHash in `C++`, using a variety of optimizations. For one, we used a `std::map` to implement the ring structure described above, and performed the clockwise movement described above by using the `lower_bound()` method of the `C++` `std::map`. This was an effective way to implement consistent hashing, since the `std::map` is implemented as a red-black tree, which allows for logarithmic (in the number of servers) insertions, lookup, and removal of key values.

Also, we investigated a number of hash functions for this project, and initially had poor results due to hash functions that were not effectively mapping numbers across the key space. While this is going to be an issue in general (making good hash functions is a field in itself), we found several functions that had been rigorously tested and been open-sourced, and those are included in our implementation – note that none of them are cryptographic hash functions for speed reasons, but they all have relatively nice distributive properties.

4 Applications of Consistent Hashing

Unsurprisingly, consistent hashing has been used in myriad distributed caching systems on the internet – conventional hashing is simply impractical given the

likelihood of server crashes or adding new servers in a distributed online setting [5, 8, 4].

Perhaps the canonical usage of consistent hashing today is the enormously popular peer-to-peer lookup service Chord, developed by Stoica et al. at MIT in 2001. The Chord protocol attempts to solve the problem of finding the node in a large network which stores a particular data item, and supports only a single operation: given a key, what node does that key belong to? This operation, in most implementations of Chord, takes a 160-bit key and returns the IP address of the node responsible for that key. The system differs from typical consistent hashing in that given an N node Chord system, each node maintains information about only $O(\log N)$ other nodes, and any lookups will take $O(\log N)$ messages to other nodes, instead of the centralized information which is the norm for consistent hashing systems but would not work in a peer-to-peer system. The overarching logic is nonetheless the same. Each node k maintains a so called 'finger table' with 160 entries. The ith entry in this table at node k contains the identity of the first node that succeeds n by at least 2^{i-1} on the ring hash, that is, $k' = \text{successor}(n + 2^{i-1})$, where i is in $[1, 160]$, and all computations are mod 2^{160}. So, for example, the first finger of k is its successor on the circle. In figure 1 we see an example state of the Chord ring. When a node n doesn't know the successor of a key q, and the successor of that key is not given directly by the node's finger table, it issues a request to an intermediate node closer to q to find q's successor. Let's say we are node 3, and want to find the successor of identifier 1. We see that 1 is on the ring interval [7,3), so we check the third entry in the finger table of 3, and see 0, so we go to node 0, which checks its finger table and sees that the successor of key 1 is node 1, and return this information to node 3. We see that in general, each hop will at least halve the remaining keyspace, which is where the $O(\log N)$ bound on lookup time comes from.

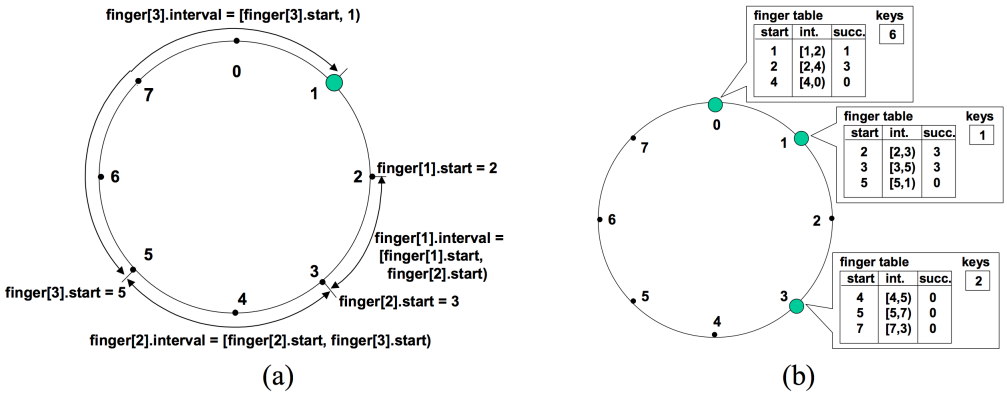


Figure 1: **Chord Finger Tables**
An example of a state the chord identifier ring could be in and how a lookup would work, taken directly from [4].

We see that, despite the local instead of global capturing of information about the ring, the chord protocol functions very similarly to a standard consistent hashing protocol, and our new data structure could certainly be applied to this setting.

In addition to heavy usage in internet protocols like Chord and like the routing servers at Akamai, consistent hashing is finding increasing popularity in the world of databases. For example, Amazon's massive, mission-critical Dynamo database uses consistent hashing to distribute load across its nodes. Another well-known recent example is the usage of consistent hashing in the FAWN (Fast Array of Wimpy Nodes) key value data store [3]. This system is a cluster architecture designed to serve massive loads while consuming very little power, through a combination of per-node flash storage (given the odd properties of flash writes, FAWN uses a log-structured datastore per node), balanced load distribution via a consistent hashing ring, and embedded CPUs. The authors report that their usage of consistent hashing, which they acknowledge to be similar to Chord, enabled FAWN to achieve fast failover and node insertion rates: critical properties for a massive key-value store like FAWN.

5 Cuckoo Hashing

Cuckoo Hashing was first developed by Rasmus Pagh and Flemming Friche Rodler [9]. The basic setup involves having two independent hash functions and two hash tables (one hash table per function). An element is hashed to one of the hash tables, and if that location is already occupied, the element there is rehashed into the other table, if that one is available. This is called a "cuckoo" and is done repeatedly until an element is successfully inserted into the table. Furthermore, it is possible that the sequence of insertions forms a cycle (where a particular key is hashed to the same location twice), in which case two new hash functions would need to be picked (or the number of buckets changed), and all of the keys would need to be rehashed.

Critically, Pagh and Rodler show that Cuckoo Hashing has amortized $O(1)$ insertion of elements, and $O(1)$ lookup and removal. We can attain this by viewing the structure of the cuckoo hash tables as a bipartite graph, as done in [10], where each of the bins are nodes in the graph, and an edge connects the two sides of the graph if there is a key that hashes to each of the vertices of that edge. Then, the cuckoo hash insertion can loop infinitely if there is a connected component with more than 1 cycle. This relies on the following theorem:

Theorem 2. *Inserting a key into a connected component with at most 1 cycle will not cause an infinite loop.*

This is proven in [10]. A simple way to consider the situation is that an infinite loop happens when a key tries to enter a particular bucket more than once. However, this can only happen if there are at least two cycles in the connected component, since that key has to be kicked out of both its initial spot and the other spot to attempt to join the original one again.

For the purposes of this explanation, we will be dealing primarily with the case where there is no such cycle, as our proposed data structure does not have rehashing (instead, it tolerates exceeding the maximum load of each bucket if enough rehashes have been tried).

If there is no cycle in any connected component, the expected time an insertion will take is proportional to the expected size of a connected component in the graph (since the number of edges in the connected component corresponds to how many times keys will need to be cuckooed before the process rests).

To analyze this, [10] shows that the bipartite graph has the following property

Theorem 3. *The expected size of a connected component in the bipartite graph is* $O(1 + \frac{1}{\epsilon})$.

This is an improvement over using just a single hash table, as it allows for hash collisions without needing chaining or linear probing (which both increase the lookup time). The proof for this theorem is given in [10]. As a brief overview, the proof relies on reducing the issue of connected components in a bipartite graph to a problem about subcritical Galton-Watson processes, on which bounds can be applied to show that the expected size of a connected component is $O(1 + \frac{1}{\epsilon})$.

6 CuckooRings: A Synthesis of Cuckoo and Consistent Hashing

6.1 Overview

In this section, we will describe our proposed data structure. This structure will combine aspects of Consistent Hashing and Cuckoo Hashing in a fast system for data allocation and load balancing problems.

As a summary, with N servers, our data structure will allow for $O(\log N)$ key insertions, server insertions, and server removals: these run time figures come from the fact that we use a red black tree to represent the sorted locations of the servers along the ring (using C++'s `std::map` implementation).

Each CuckooRings object contains two RingHash structures - which are an implementation of the aforementioned consistent hashing ring. In order to insert an item into the CuckooRings, we insert it into the RingHash that has fewer keys hashed to it. The RingHashs have a threshold number of keys per bucket. If the load on the RingHash bucket we insert into exceeds this number, we rehash the entire contents of that bucket into the other RingHash (see Figure 2), and then continue to do this recursively until either all of the buckets are within the threshold, or we pass some maximum recursion depth. If we exceed the maximum number of rehashes, we stop moving elements back and forth, which may allow the number of keys to exceed the threshold on some servers.

To look up an item in the CuckooRings, we determine which bucket of each of the RingHashs it is hashed to, as in Cuckoo Hashing, and do a linear search on those buckets to find the item (these keys were supposed to represent server jobs in which order doesn't matter and jobs will probably disappear with decent frequency anyway, but if we really wanted to make this as fast as possible we could keep a sorted data structure instead of a vector).

If a server is removed or added, we only have to rehash the part of the key-space affected by that one server, which is the same as our result for standard consistent hashing.

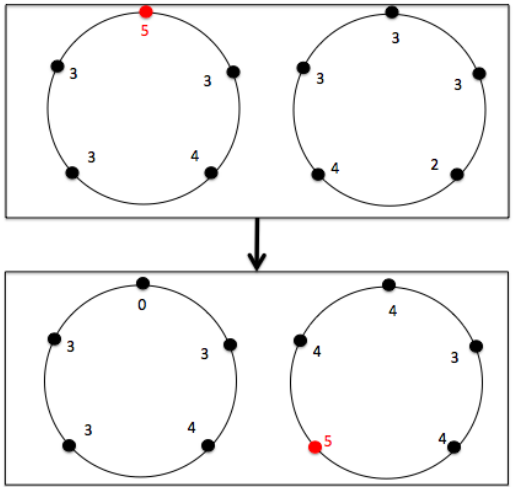


Figure 2: This shows one step of an insertion process where the threshold on a server is a 4. The rectangles are CuckooRings, and the circles inside them are RingHash structures. The points on the circles refer to servers, and the numbers are loads. The red servers have exceeded their threshold. In the next step of this process, we would rehash the keys in the server with too many keys in the right RingHash back into the left RingHash.

Why would we expect this to work well? In several ways it emulates the reasons that Cuckoo Hashing is effective. Namely, if several keys are hashing to the same bucket repeatedly, it allows us to rehash those keys using a second hash function, which may potentially perform better than the first one for that set of keys. Moreover, this makes the spread of keys across the buckets more uniform. If we set a small threshold on the number of keys per bucket (say $\epsilon(M/N)$, for $\epsilon = 2$, say, then we expect that most of buckets will be within that threshold (based on experimental, not theoretical analysis).

6.2 Code

All of our code is available on Github:
`https://github.com/jonahkall/CuckooConsistent`

7 Results: CuckooRings vs. Consistent Hashing

7.1 Criteria for Analysis

- **Time**: This is simply how much time it took the test to run in the CuckooRings versus in the standard RingHash. We aimed for a small or zero reduction in speed from the Ringhash.
- **Maximum Load**: This is defined as the maximum number of keys that any server has. This is important from a job-allocation standpoint because it can be used as a measure of the maximum latency that a client would experience.
- **Cost**: This is defined as $\frac{1}{n}\sum_i X_i^2$, where X_i is the load of the ith server. This cost function penalizes heavy load concentration on a small number of servers, and can serve a measure of the general effectiveness of a hashing scheme. This approach is also suggested by [11] as a good measure of clustering.

7.2 Testing Standards

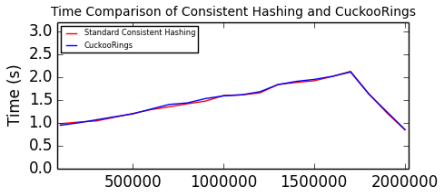
We evaluated the performance of our data structure through a series of tests, using the above criteria to distinguish between the CuckooRings and RingHash data structures. These are the tests that we constructed:

- **InsertKeys Test**: This test simply made an increasing number of insertions to a data structure of a fixed size, and recorded the above criteria for the two data structures.
- **RemoveServer Test**: This test began by inserting a constant number of elements into the data structures, and then removing an increasing number of elements in each sample. This allowed us to see how metrics like max load and cost would scale in conditions when there is widespread server failure (we tested up to the case were 10% of the total servers failed.
- **RandomActions Test**: This test was meant to mimic the time evolution of a real live distrbiuted system. Rather than simply adding or removing keys, we insert an increasing number of keys as well as randomly removing or adding servers. Furthermore, in this situation, we increased the size of the data structures (number of servers initially in the server) proportionally with the number of insertions we were making. This basically allows us to keep the ratio of initial keys to initial servers constant, and thus lets us see how the different metrics scale at different sizes. Having different server thresholds should particularly affect this test, since the ratio of number of keys to number of servers is roughly constant in these tests. In this case, $(M/N) \approx 1$.
- **Different Server Thresholds**: We changed the server thresholds between 2, 5, and 10. These thresholds correspond to the maximum load in

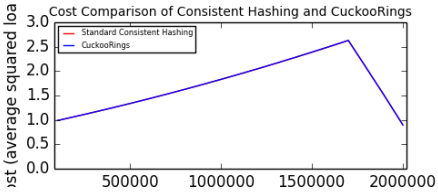
a server before the server begins cuckooing. Doing the tests at different server thresholds allowed us to see how the various metrics differed across different thresholds, which affect the amount of cuckooing that occurs during insertions.

7.3 Charts and Analysis

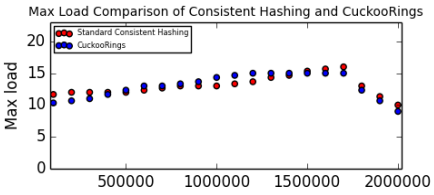
For this section, we tested the criteria above with different bucket capacity thresholds. Altering these thresholds effectively manages a trade-off between the expected amount of time that an insertion takes (which should be less for higher capacities, since there would be less cuckooing happening), with decreased benefits from the hash table structure, since there would be higher loads. In figure 3 we present the results from the tests with a threshold of 2, in figure 4 with a threshold of 5, and in figure 5 with a threshold of 10.



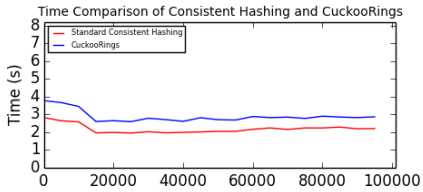
(a) InsertKeys Time



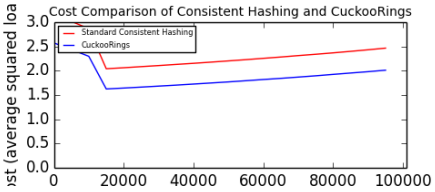
(b) InsertKeys Cost



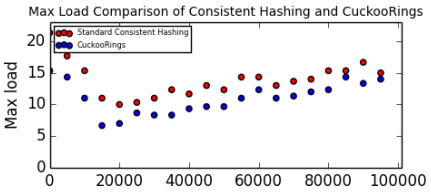
InsertKeys Max Load



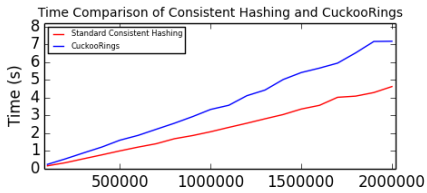
(a) RemoveServer Time



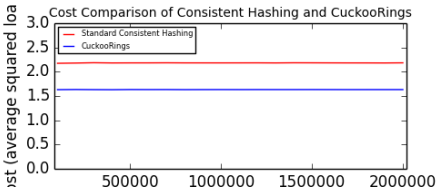
(b) RemoveServer Cost



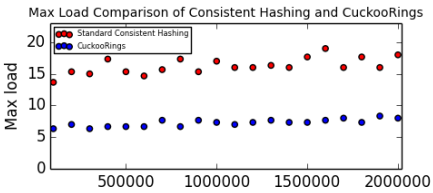
RemoveServer Max Load



(a) RandomActions Time

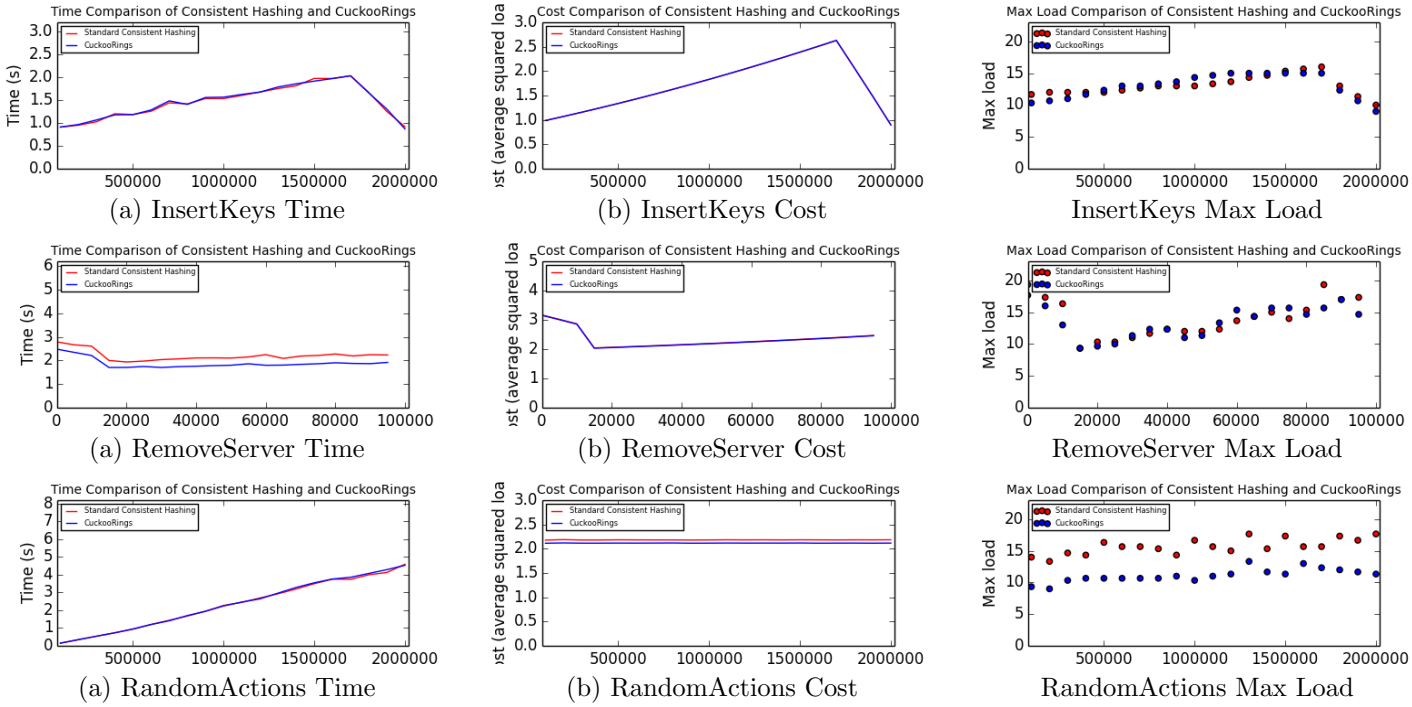


(b) RandomActions Cost



RandomActions Max Load

Figure 3: **Test Results for Threshold = 2**
This is the test case that we would expect to see the biggest benefits in terms of cost and max load, but the biggest drawbacks in terms of times. We would especially expect to see this in the RandomActions tests, since those involve many different kinds of actions. This hypothesis seems consistent with the data.



(a) InsertKeys Time (b) InsertKeys Cost InsertKeys Max Load

(a) RemoveServer Time (b) RemoveServer Cost RemoveServer Max Load

(a) RandomActions Time (b) RandomActions Cost RandomActions Max Load

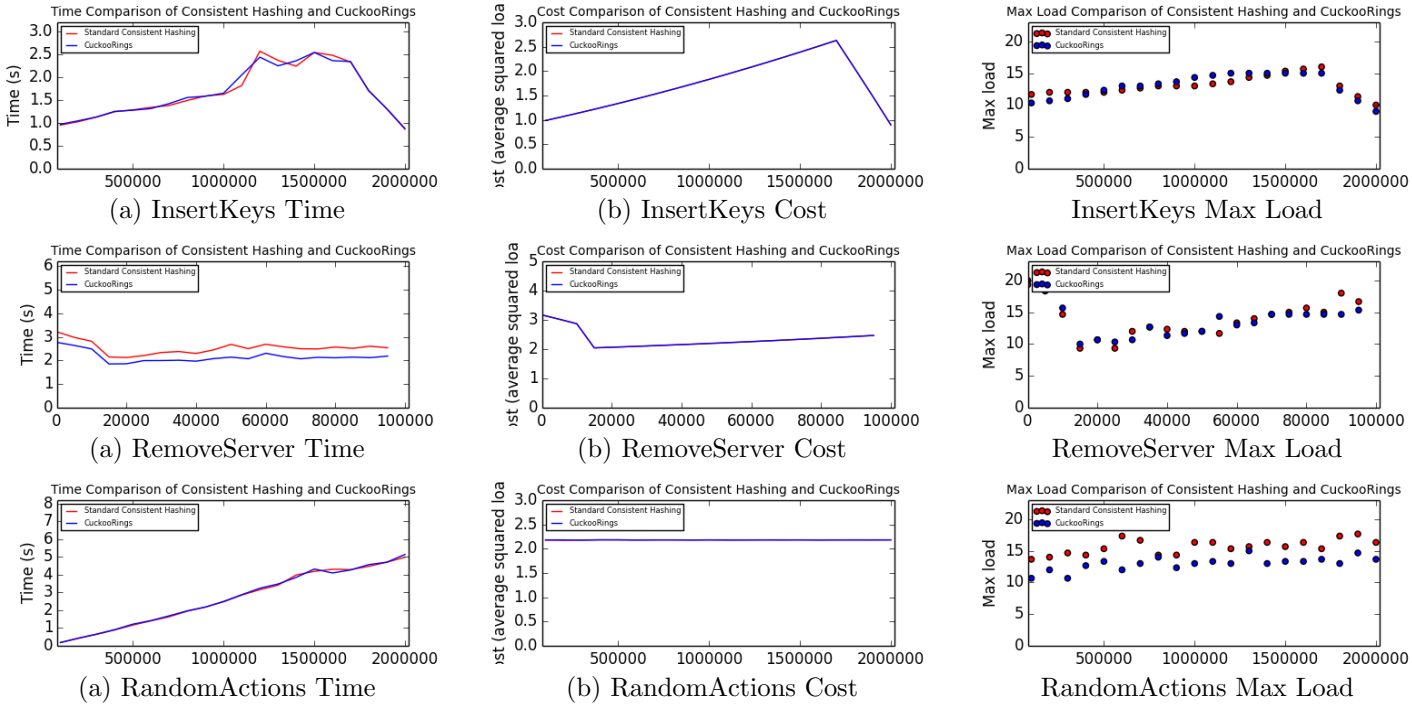
Figure 4: **Test Results for Threshold = 5**
In this case, we see less of a trade-off in terms of time, but also less benefits in terms of cost and max load reductions than the lower threshold.

There are several major takeaways from the data that we collected. For one, we found that across all metrics, the marginal time that it takes to use a CuckooRings structure is small or nonexistent: we often achieved very similar and occasionally even better runtimes. These cases are likely a result of reduced loads causing linear key lookups to take less time on average. The more interesting stories are in the max loads and the overall costs.

We see that there is a substantial reduction in the maximum load of the structure in most cases, particularly in the realistic RandomActions test. For the applications that we are dealing with, a potential reduction in latency by a factor of 2 (as we were seeing in terms of max loads), would lead to substantial benefits.

Also, we see that there are benefits in terms of the overall cost of the structure. This metric is important, as the maximum load metric only specifies what the improvement in the worst server is, while this metric assesses the general success of the dsitribution (while penalizing the especially high loads). We see that the costs associated with a CuckooRings structure are also substantially lower (and consistently so) than that of the RingHash structure.

It is interesting to see that the RandomActions test is what differentiates



(a) InsertKeys Time (b) InsertKeys Cost InsertKeys Max Load

(a) RemoveServer Time (b) RemoveServer Cost RemoveServer Max Load

(a) RandomActions Time (b) RandomActions Cost RandomActions Max Load

Figure 5: **Test Results for Threshold = 10**
With this relatively high threshold, we see very little difference in performance between the two data structures.

the two data structures the most, across all of the criteria. This is promising, as the random actions test is perhaps the most realistic representation of the distributed environment in which a CuckooRings structure would likely be deployed.

8 Some Open Theoretical Questions

We have achieved definite experimental benefits in load balancing using this new data structure, but what theoretical justification can we provide for these benefits? Unfortunately, analyzing this data structure's expected run time and max load is very difficult because of the complexity of its cuckoo graph. As above, we can construct a cuckoo graph. However, in our data structure, there is no longer a single element per bucket, meaning that the graph multiplies significantly in complexity: every key in a bucket needs to be connected to its corresponding buckets in the other ring. As in the standard cuckoo hashing analysis, we see that insertions only cause a problem if more than one cycle is generated in the cuckoo graph. At this point however, the analysis seems to

break down, because we no longer have the theorem that bounds the sizes of the connected components, as the proof assumes things which are no longer true in our situation (namely the cycles back and forth can contain multiple edges). We pose the following questions:
Question 1: *Bound the key insertion time into a CuckooRings structure.*
Question 2: *Bound the insertion time of a server into a CuckooRings structure (this could probably be at least trivially bounded because there are $O(K/N)$ expected rehashes, so this bound could be at least loosely derived with a union bound)*
Question 3: *Bound the expected max load given a number of keys, number of servers, and a cutoff.*

Our experimental results clearly demonstrated there is a trade-off between time, the cutoff, and the max load. Namely, reducing the cutoff leads to significantly lower max loads, but, because rehashes are going back and forth more frequently, also takes more time. We are curious about the best way to make this trade-off subject to some set of constraints.
Question 4: *Provide a theoretical analysis demonstrating how to optimize the cutoff given the desired trade-off between time and max load. For example, this could be specified in the form of a cost function.*

9 Future Work

There are substantial opportunities for future work in this area. For one, we have not substantially explored what happens when one adds replication of the servers into the structures. Replication works by having several virtual servers on the circle corresponding to real servers. Keys are hashed to virtual servers, which get sent through an intermediate mapping to the actual server which can handle the request. In particular, this approach should somewhat decrease the maximum load on the servers, but this does add costs in terms of insertion time. So, it would be interesting to see what benefits the proposed data structure has over this approach, if any, and whether the proposed structure can be further optimized using replication.

As mentioned above, this data structure is difficult to analyze theoretically, and so any kind of comprehensive analysis of its expected behavior would be interesting. There are a number of other areas we can explore. For one, we have developed a fairly free-form API that allows us to experiment with different configurations of the data structure, and have made it easy to change this configuration dynamically. We have explored to some degree what real world workloads look like, based on literature review and conversations with researchers in the field, but it would be interesting to try to run our system on an actual, real-live workload (perhaps one recorded from a real world system like Dynamo). Given our system's load balancing properties, it would also be interesting to explore whether our system could alleviate some of the known problems with the Chord distributed protocol, many of which stem from concentrated load at certain points in the ring.

Finally, it would be interesting to see if using more than 2 rings could produce even better results, and more generally, what the number of rings vs. cutoff vs. number of keys vs. number of servers trade-off space looks like.

References

[1] Tran Ngoc Thinh, Surin Kittitornkun, and Shigenori Tomiyama. Applying cuckoo hashing for fpga-based pattern matching in nids/nips. In *Field-Programmable Technology, 2007. ICFPT 2007. International Conference on*, pages 121–128. IEEE, 2007.

[2] Marcin Zukowski, Sándor Héman, and Peter Boncz. Architecture-conscious hashing. In *Proceedings of the 2nd international workshop on Data management on new hardware*, page 6. ACM, 2006.

[3] David G Andersen, Jason Franklin, Michael Kaminsky, Amar Phanishayee, Lawrence Tan, and Vijay Vasudevan. Fawn: A fast array of wimpy nodes. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 1–14. ACM, 2009.

[4] Ion Stoica, Robert Morris, David Karger, M Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. *ACM SIGCOMM Computer Communication Review*, 31(4):149–160, 2001.

[5] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: amazon's highly available key-value store. In *ACM SIGOPS Operating Systems Review*, volume 41, pages 205–220. ACM, 2007.

[6] Rasmus Pagh and Flemming Friche Rodler. *Cuckoo hashing*. Springer, 2001.

[7] David Karger, Eric Lehman, Tom Leighton, Rina Panigrahy, Matthew Levine, and Daniel Lewin. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, pages 654–663. ACM, 1997.

[8] Avinash Lakshman and Prashant Malik. Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review*, 44(2):35–40, 2010.

[9] Rasmus Pagh and Flemming Friche Rodler. *Cuckoo hashing*. Springer, 2001.

[10] Lecture notes on cuckoo hashing, stanford university. May 2014. http://web.stanford.edu/class/archive/cs/cs166/cs166.1146/lectures/13/Slides13.pdf.

[11] Lecture notes on hash functions, cornell university. 2014. http://www.cs.cornell.edu/courses/cs312/2008sp/lectures/lec21.html.