Multi-copy Cuckoo Hashing

Dagang Li*†, Rong Du*
*SECE, Shenzhen Graduate School, Peking University
†Shenzhen Key Lab of Info. Theory&Future Net. Arch.
Shenzhen, China
dagang.li@ieee.org, duroong@163.com

Ziheng Liu, Tong Yang, Bin Cui
Department of Computer Science
Peking University
Beijing, China
liuziheng, yant.tong, bin.cui@pku.edu.cn

***Abstract*—Cuckoo hashing is widely used for its worst-case constant lookup performance even at very high load. However at high load, collision resolution will involve lots of probes on item relocation and may still fail in the end. To address the problem, we propose an efficient Cuckoo hashing scheme called Multi-copy Cuckoo or McCuckoo. Different from the blind kick-outs of standard Cuckoo hashing during a collision, we can foresee which way to successfully kick items by using multiple copies. Furthermore, with the knowledge of how many copies each item has in the table, we can identify impossible buckets and skip them during a lookup. In order to avoid expensive rehashing during insertion failures, McCuckoo also supports more efficient stash strategy that minimizes stash checking. McCuckoo uses simple logic and simple data structure, so it is suitable for both software and hardware implementation on platforms where intensive access to the slow and bandwidth limited off-chip external memory is the main bottleneck.**

Keywords—Cuckoo Hashing, Hashing algorithm, Multi-copy

I. INTRODUCTION

Hash tables [1] are basic data structures that are widely used in various fields such as database, networking, storage, security, etc. When working at low load, hash collisions rarely happen; but when load increases, collisions will happen frequently. Traditional solutions such as chaining methods and linear probing [2] need extra time and resource to resolve the collisions, affecting both insertion and lookup and losing the favorable performance bound.

Unlike traditional hash structures that only provide one candidate location for each item, Cuckoo hashing [3] uses d hash functions to provide multiple candidate buckets for each item to choose so as to reduce collision. Most importantly, existing items can be "kicked out" and relocated to another candidate bucket if necessary to make room in search of an overall arrangement for all items to settle down in their own buckets. This flexibility provided by multiple hashing and relocation helps Cuckoo hashing achieve very high *load ratio* (more items under the same table size) while still keeping worst-case constant lookup, during which at most d buckets may need to be checked to find the item. Because of its capability to provide worst-case $O(1)$ lookup at very high load, Cuckoo hashing has been the preferred hash technique in many fields such as storage systems[5], databases[6], privacy & security[4][7][14], networking[8][10], architecture [9][11], data processing [12][13], and so on.

The favorable lookup performance comes at the cost of the hardship to resolve collisions during insertion. At high load ratios, the cost to resolve increased collisions inevitably becomes high, since now more item rearrangements may have to be probed before a working one can be finally found. This greatly affects the insertion delay, and the collision resolution may still fail even after the costly relocating effort. Generally there are three main factors that determine the performance of Cuckoo hashing.

1) The recursive kick-outs during an insertion. At high load ratio, the probability of collision becomes high when a new item is inserted, and the probability of finding a solution only after a few round of kick-outs also decreases [9]. Moreover, because standard Cuckoo hashing cannot forsee which item has empty alternative buckets, it can only probe for one in BFS order or in a random fashion. The blindness in these trial-and-error probing approaches may take too much time to find a resolution or even end up in an endless loop [3], which is the main cause of insertion failures. A good strategy should find a solution fast if such solution exists.

2) The cost to handle insertion/lookup failures. When a collision resolution cannot be reached during insertion, the traditional Cuckoo hashing suggests a costly rehashing solution, reading out all inserted items and using a different set of hash functions to put them into a bigger table, during which the hash table is completely unusable. A more practical remedy is to allocate some small additional space to store all the items that fail in insertion [22], but then if an item is not found in the main table during a lookup, the stash needs to be double checked. This extra checking not only affects lookup performance, but also limits the number of items it can hold to handle very high load. To maximize the benefit of a stash, we should minimize its search cost, improve its scalability and reduce unnecessary checkings.

3) Multiple bucket checking for a single lookup. When looking up an item in a Cuckoo hash table, multiple buckets may need to be visited because the item can be in any of them. The additional visits affect the lookup performance and may be a big drawback when the table is too large and need to be put in slow and bandwidth limited off-chip external memory. If we can narrow down the subset of buckets that may contain the item beforehand and optimize the accessing pattern, we may increase the possibility of finding it with fewer visits and improve the lookup performance.

For the first problem, normally a *maxloop* is defined to draw the insertion procedure out of an endless loop, but still time and resource so far are already wasted. Proposals such as SmartCuckoo [15] and Necklace [16] tried to identify loops beforehand, so we won't run into an endless loop situation in

the first place. MinCounter [17] tries to reduce repetitions in the recursive kick-out process, so more buckets will be searched and the possibility of finding an empty one will increase. Other work such as blocked Cuckoo hash tables (BCHT)[18][19][20][21] allocate multiple slots within each bucket, and the set-associativeness among these slots provides another level of flexibility which help to reduce collisions and reach an even higher load ratio. As long as the whole bucket can be retrieved in one memory access [33], there will be no sacrifice on lookup performance.

For the second problem, most stash-based solutions such as Cuckoo hashing with a stash (CHS)[22] propose to put the stash on-chip to minimize its impact on performance[7][23] [13][24]. When the stash itself is full, items stored in it will take a try to the main table until some space is freed. A small stash of size 4 is regarded as enough to achieve rather high load (for example 95% in [24]) with high probability.

Checking multiple locations for every single lookup is more of a problem if accessing the table buckets is slow or expensive, which is the case when platforms with only limited fast on-chip memory need to handle very large lookup tables (for example the ASIC/FPGA/SOC based packet processing devices). A common practice is to use compact helping structures such as Bloom filters that can fit in the on-chip memory to do pre-screening, so as to minimize unnecessary visit to the main table in the slower off-chip memory, such as DEHT [25] and EMOMA [24].

In this paper, we propose a simple and effective Cuckoo hash mechanism to address all the three problems discussed above. The main idea is to store *multiple copies* of item in the table, so we don't have to rashly choose one at insertion time when more candidate buckets are available, so as to keep the flexibility on placement as much and as long as possible. The level of redundancy provides explicit clue to the choice of replacement target during collisions, therefore not only accelerates the insertion speed but can also help avoid endless loop of kick-outs. On the contrary, in traditional single-copy Cuckoo hashing the placement flexibility is immediately consumed when the inserted item settles down, which not only might be sub-optimal and has to be corrected by relocation later on, the relation among candidate locations is also lost that adds to the blindness in the resultant kick-outs. Keeping copies in all the available candidate buckets will maintain the flexibility and avoid entering the sub-optimal situation early: the optimal placement will come out naturally later on when the other occupied buckets are appropriately given away as per request to new items, who turn out to be the better owners of these buckets in an overall optimal arrangement.

McCuckoo assign a counter for each bucket and use them to track the number of copies of the stored items. Buckets with counter value larger than 1 can be readily overwritten when necessary because we know the item still has other redundant copies in the table. Since all the copies of the same item are each others' redundant, a table only containing multi-copy items actually has the same *bucket availability* as an empty table to accommodate new items anywhere in the table.

Regarding to lookups, since all buckets containing the same item should have the same counter value, this fact can also be used to tell which candidate buckets do not contain a an item for sure, so we can skip checking them during a lookup; and a counter value 0 from any candidate bucket can save us from checking the table at all (similar to a Bloom filter). Furthermore, because an item can always overwrite a redundant copy to settle down, if a lookup fails with any candidate bucket having counter value larger than 1, we know that item must have not been inserted before and skip checking the stash. These and other further observations on the behavior of the counters can enrich the operation rules to improve the effectiveness and performance of McCuckoo.

McCuckoo is mostly suitable for platforms that have a hierarchical memory structure where the main table can only be put to the abundant but slower second layer memory due to the large size, so all the three problems mentioned earlier will become equally apparent and McCuckoo can handle them altogether in an unified framework. In order to maximize the benefit of the counters, the counters need to fit in the on-chip embedded memory that is order-of-magnitude faster. We propose a compact on-chip counter array and specifically designed counter operations that only involve very simple logic, so that McCuckoo can be easily implemented in both software and hardware.

The contributions of this paper are as follows:

1) Introducing the idea of multi-copy into Cuckoo-based hashing architecture, which helps minimize the blindness in item relocation and improve bucket availability, improving insertion speed and success rate.

2) A new compact on-chip helping structure is proposed that can minimize unnecessary off-chip memory access with less on-chip memory cost than current solutions.

3) An efficient pre-screening mechanism is proposed to support a large off-chip stash by minimizing stash checking during failed lookups.

The rest of the paper is organized as follows. In Section II we will discuss related work and background information. The design considerations as well as details and extensions of McCuckoo will be introduced in Section III. Experimental results are presented and analyzed in Section IV, and Section V concludes the paper.

II. Background and Related Work

A. Standard Cuckoo hashing

Cuckoo hashing was first proposed in [3] as a dynamic variation on multi-choice hashing dictionary. It contains d hash tables of length n (T_1,T_2,...,T_d) and d hash functions h_1,h_2,h_3,...,h_d:$S \rightarrow \{0,...,n-1\}$. Any item $x \in U$ will be stored in one of its d candidate locations (called buckets) from these tables, which are $h_1(x)$ in T_1, $h_2(x)$ in T_2, ... $h_d(x)$ in T_d as determined by the d hash functions, therefore when querying for an item we only need to check these d buckets. However if all the candidate buckets are already occupied during an insertion, we need to "kick" one of the occupants away to make room. The evicted item needs to check if some of its other candidate buckets is empty, or else the "kick-out" will continue until every item finds a bucket to settle down. The capability of kicking-out old items wins Cuckoo hashing its

name as well as more flexibility to resolve collision than the other deterministic hashing algorithms.

Three situations may happen during insertion of Cuckoo hashing. An example of a d=2 Cuckoo hash table is given in Fig.1, where items are represented by an arrow starting from the bucket it occupies pointing to its alternative bucket. Now let's say item x is to be inserted and its 2 candidate buckets are $T_1[2]$ and $T_2[5]$. Fig.1(a) through 1(c) demonstrate how these situations are handled by Cuckoo hashing.

In Fig.1(a), item x is simply put to the empty candidate bucket $T_2[5]$ and the insertion is completed in $O(1)$ time. If all candidate buckets are occupied, one of the existing items needs to be relocated to make room for x. In Fig. 1(b), item a is kicked out, who recursively kicks item b out and so on, until item d is relocated to $T_2[3]$. The red letters in brackets show the final locations of each item. This situation will become more frequent when the table starts to fill up. We may run into a third situation in the recursive kick-out that no empty bucket can be found. In Fig. 1(c), the kick-outs form an loop and will never end. In practice a threshold called *maxloop* is used to quit from such situation and claim a failure [3]. The value of *maxloop* determines the trade-off between wasted kicking attempts and the possibility of a premature false alarm.

B. Cuckoo hashing Variants

Since Cuckoo hashing was first proposed, it has attracted much interests trying to improve it or make use of it in various applications. *d*-ary Cuckoo [27] and blocked Cuckoo [20] hash tables extend the original Cuckoo hashing from its 2-hash one-item-per-bucket simple design to using *d* hash tables or storing *l* items in each bucket, so the achievable load ratio can be improved to more than 90%. Since the 2 approaches are both "multiplication" extensions for similar objectives, later work very often combine the two to be more flexible in the data structure to meet specific design considerations [18][19]. Furthermore, in order to alleviate the increased hash calculation, hardware solutions using GPU [31] or software solutions using double hashing [21] are also developed.

To avoid being trapped in an endless loop, SmartCuckoo [15] uses directed pseudo forest to efficiently predetermine endless loops without paying the high cost of step-by-step probing, but it only works with 2 hash functions. Necklace [16] tries to maintain the relationship among alternative buckets in an auxiliary record to increase the chance of finding the minimum path during kick-outs, but the auxiliary record takes large space and visiting cost. MinCounter [17] allocates a 5-bit counter for each bucket to record the kick-out history of that bucket, and the bucket with minimum counter that is not so "hot" will be chosen during the kick-outs to reduce the total number of kick-outs in the long run. These mechanisms achieve good improvement over standard Cuckoo hashing but at the cost of additional space and more computations.

Another approach called Random-walk[28][29][30][20] is also proposed in the literature to reduce the insertion time in case of collisions. The original breadth-first search (BFS) strategy is inefficient in practice and only has a polynomial upper bound on the insertion time that holds with high probability (whp). Random-walk requires no additional data structure and can achieve poly-logarithmic insertion time by randomly selecting an item to kick-out when all candidate buckets are occupied. However, the improvement is also limited comparing to those that keep additional records.

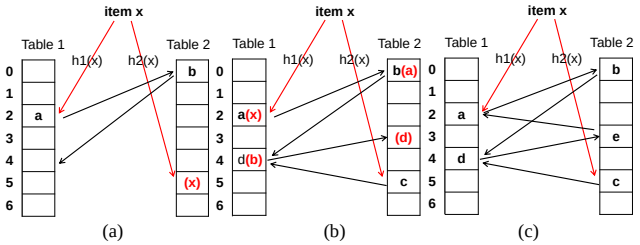

Figure 1. Insertion to a Cuckoo hash table.

Both rehashing and stash-based mechanisms [7][13][22][23][24] are used as the solution to insertion failures in various Cuckoo-based hash tables. When the table is loaded within the theoretical upper bound, a small stash of size *s* can improve the probability of rehashing from $O(1/n)$ down to $O(1/n^{s+1})$, therefore it is normally put on-chip because it is small in size but frequently visited during lookups. If a surge of insertion happens (which is possible in a very dynamic environment) that temporarily overruns the safe margin provided by the small stash, rehashing will still be inevitable.

Another category of work focuses on very big Cuckoo tables that need to be put off-chip, so that checking many buckets becomes a problem as off-chip memory access is generally much more costly. DEHT [25] and EMOMA[24] chose to use some on-chip helping structure to identify the real location of each item to achieve one-access lookups. However, the on-chip structure is rather big in DEHT, and EMOMA sacrifices some placement flexibility in the main table to battle the false positive error of the on-chip Bloom filter.

Latest advance on hash techniques in general such as AMAC[34] can also be incorporated in Cuckoo hashing to improve its lookup performance, but they are orthogonal to our work and out of the scope of this paper.

III. Design and Implementation Details

A. General idea

Comparing to all the existing Cuckoo based hash tables that try to optimize the arrangement of the sole copy of the inserted items, the biggest difference of McCuckoo is the idea of occupying all free candidate buckets with redundant copies, so as to circumvent the rash decision of picking a bucket in a hurry. Therefore existing methods can be generally regarded as reactive because they focus more on "kick-out" algorithms to resolve collisions caused by sub-optimal placement from the past; McCuckoo takes a proactive approach by keeping the decision on placement open until a more suitable item later on claims one of the buckets and replace the copy in it.

If all the non-empty buckets are occupied by multi-copy items, new incoming items can choose any bucket at will without evicting existing items. As a result, although at the same **load ratio** (number of *distinct* items against table size) McCuckoo will be filled with more redundant copies, the effective "**collision**" that leads to costly item relocation is actually less frequent. In order to maximally maintain this high insertion efficiency, one design principle of McCuckoo is to keep redundancy over all items for as long as possible.

In order to make redundant copies traceable across table updates, each bucket has a counter to record the total number of copies its occupying item current has in the table. Although the value of the counters is associated with items not the buckets, for the ease of wording we will just say "**bucket of value *x***" in the rest of the paper instead of "bucket whose associated counter has value *x*". With the counters we can make sure that an item won't be accidentally removed because all its copies are carelessly overwritten. Counter space are associated with buckets instead of items so we can have deterministic mapping and addressing which is good for easy logic implementation. Since the number of copies each item has cannot exceed *d*, only a few (2 in this paper when d=3) bits will be needed for each counter. And as d=3 is sufficient for a Cuckoo hash table to reach load ratio well over 90%, we won't see much larger d in practice.

When even more items are inserted to the table, sooner or later we may run into the situation that all the candidate buckets are occupied by items with only one copy left. At that time any existing collision resolving mechanisms such as random-walk or MinCounter can be used to start relocating items, and using these counters in each iteration to quickly find usable buckets (empty or occupied by redundant copy) or as tie breaker. In this paper random-walk is used as example.

The counters maintained in the fast on-chip memory can be used for multiple purposes. With very simple logic we can decide which buckets to use or overwrite during an insertion before accessing them (because we know which buckets are empty or only contain redundant copies), exclude impossible buckets from checking during lookup (exploit the fact that valid candidate buckets should have the same counter value), and mark a deletion without actually removing the item from the table, so as to minimize unnecessary operations to the main table in the slow off-chip memory.

B. Design Principles

1) Insertion

Unlike single-copy Cuckoo hashing that simply tries to find one empty bucket for the inserted item, McCuckoo needs to decide how many copies to store and to which candidate buckets. To maximize the placement flexibility provided by multi-copy, the strategy is to store as many copies as possible (or equivalently occupy as many buckets as possible) for each item and for all the items as a whole. This means that we also need to keep balance amongst copies from different items, so that when a random item is inserted, the probability of running into a collision is minimized. Only when all the candidate buckets contain the sole copy of other items, will we run into a real collision, so we want the number of copies decrease to 1 as slow as possible for any item in the table. This actually postpone the first occasion of a real collision, so that the table can reach much higher load with minimal insertion time.

The principles for insertion can then be summarized as below for a McCuckoo hash table with d hash functions.

1) Occupy all the empty candidate buckets.
2) Never overwrite buckets of value 1.
3) Overwrite the rest in the decreasing order of their value, until the overwriting results in more copies for the inserted item than the overwritten one.

The third principle is better explained with an example. Let's say we have a candidate bucket of value 3 - meaning it currently holds an item *B* with 3 copies. If the inserted item *A* only has 1 copy in the table, we choose to overwrite *B* so both items now have 2 copies each. If item *A* already has 2 copies, overwriting will not happen because that only changes the number of copies from 2:3 to 3:2, which gains nothing with respect to the whole table.

Theorem 1. The insertion principles above achieve the best overall redundancy and redundancy balance among the affected items.

Proof. Let's say the values of the d candidate buckets are $V_1,...,V_p,...,V_d$ in decreasing order, where only p of them are non-empty. If we evaluate redundancy by the number of item copies, then the total redundancy is $\sum_1^p V_t$ before the insertion. Because overwriting exist copies will not change the total redundancy, letting the inserted item A occupy all the empty buckets will increase the total redundancy to the maximal achievable value at $\sum_1^p V_t + (d - p)$.

Now let's look at redundancy balance. Because during the insertion $V_1,...,V_p$ cannot increase but only decrease by 1 when the bucket is overwritten by the inserted item, therefore start overwriting from the largest V_1 will always improve balance among $V_1,...,V_p$. On the other hand, the inserted item A is the only one that can increase its redundancy during the insertion. When it overwrites one bucket of V_t, because all the other buckets are not involved, the overall balance is only affected by the decrease in V_t and the increase in V_A. If we stop overwriting before the condition in principle 3 is met, that means there is at least one item among $V_1,...,V_p$ that has at least 2 more copies than item A. Clearly replacing it by A will improve the redundancy balance. If we keep overwriting after the condition in principle 3 is met, we effectively push the following V_t and V_A apart which decreases the redundancy balance. Therefore we should stop as principle 3 dictates. ■

Theorem 2. The total number of proactive redundant writes will not exceed $1+\sum_3^d 1/t$ times the size of the table.

Proof. We follow a constructive approach to find the maximal number of proactive writes for redundant copies. Let's say the McCuckoo table has S buckets in total. When a new item arrives, the maximal achievable redundancy would be d when it shares no buckets with any existing items. This can continue until all the S buckets are occupied, where $S\cdot(d-1)/d$ writes are redundant ones. From then on, for each new inserted item the maximum achievable redundancy is reduced to (d-1) when (d-1) items from the first round give up one bucket to it. This can continue until all the S/d items each has one copy less, and the number of redundant writes is $(S/d)\cdot(d-2)/(d-1)$. Similarly in the next round the number is $S/(d-1)\cdot(d-3)/(d-2)$ and so on, until the second last round we have $S/3\cdot1/2$. Before the last round all the existing items have 2 copies each, so in the last round there will be no redundant writes. To sum up, in total we have at most

$$S\cdot\frac{d-1}{d}+\frac{S}{d}\cdot\frac{d-2}{d-1}+\cdots+\frac{S}{4}\cdot\frac{2}{3}+\frac{S}{3}\cdot\frac{1}{2}=S\left(\frac{d-1}{d}+\sum_3^d\frac{t-2}{t(t-1)}\right)$$
$$<S+\frac{S}{d}+\frac{S}{d-1}+\cdots+\frac{S}{4}+\frac{S}{3}=S\cdot\left(1+\sum_3^d\frac{1}{t}\right)$$ ■

In the case of d=3, the total number of redundant writes will never exceed $5/6$ of the table size, which is not too

excessive to trade for much faster insertion efficiency. This theorem also tells us that the majority of the redundant writing happens when the table is building up; when the table is loaded and in normal working condition, there will be much less overhead for redundancy.

2) Lookup

In McCuckoo, the on-chip counters are also used to minimize checking to the off-chip main table for lookups on both non-existing and existing items.

First, they can help identify non-existing items. During an insertion, all candidate buckets will be filled, either by the item itself or some earlier items, resulting in non-zero value for all the corresponding on-chip counters. This behavior is very much like Bloom filter [32] that sets all hashed locations to 1 for each inserted item. Indeed, if we look at the on-chip counters as zero or non-zero, they actually form a standard Bloom filter and collectively they can answer if an item has been inserted to the table or not, with some false positive error but no false negative. This will avoid most wasted access to the off-chip table for queries on non-existing items, and even if such query passes the filtering with false positive, the main table will still return the correct answer.

Second, they can also reduce memory access for existing items. For example, we can always skip a bucket of value 0 because it is empty. Because the buckets containing the same item should always have the same value, we can also skip safely all the buckets that have a value that cannot be matched by the number of buckets. Furthermore, for the buckets that share a common matching value, checking just one of them is sufficient to return the right answer.

For example, if there are less than 3 candidate buckets of value 3, all of them can be safely skipped, because there are not sufficient valid buckets to support so many copies. Those buckets should hold other items that have 2 more copies elsewhere in the table. On the other hand, if there are 3 candidate buckets of value 2, we cannot exclude any one of them because if the item really has two copies in the table, they can be in any two out of the three of them. However, we only need to check up to two of them because statistically we should run into at least one of the copies in these two attempts; if we can't find the item after these two attempts, it can't be in the third bucket, either. Out of observations like these, simple but effective logic can be extracted to narrow down the checking scope and minimize unnecessary access to the off-chip main table. In practice we can achieve zero or one access for a large portion of lookup queries, especially when the table is moderately loaded.

To sum up, the principles for lookup to a McCuckoo hash table with d hash functions can be summarized as below.

1) Skip all the buckets and return negative if any candidate bucket has a value 0.
2) Partition the non-zero candidate buckets according to their value, and skip those partitions whose size is smaller than the associated value.
3) For each of the remaining partitions, if the size is S and the associated value is V, check up to S-V+1 buckets in the partition. Return the queried item if it is found, otherwise return negative.

Theorem 3. The lookup principles above can always narrow down the checking scope unless all candidate buckets are of value 1.

Proof. All the buckets will land into the scope of one of the 3 principles. If any one of them lands into the first two, it can be skipped; if they fall into the last one, (V-1) buckets in each partition can be skipped. Only when all the buckets are together with V=1, none of them can be skipped. ■

Theorem 3 tells us that before the table is extremely full, McCuckoo can always reduce checking effort during lookups.

3) Deletion

Deletion is very much like Lookup but now we need to visit all the buckets that contain a copy of the item and remove them all from the table. In order to further reduce access to the off-chip main table, after finding all the copies (actually we don't have to visit all the candidate buckets because from the discussion in lookup we know that some buckets can be safely excluded, but still at least one of them need to be visited in case the item to be deleted is not in the table), we can choose not to physically remove them from the main table and just reset their corresponding counters to 0, which will save all the writes to the off-chip memory during a deletion.

Resetting counters of the buckets holding the deleted item to 0 has consequences on lookup, because these buckets can be candidates buckets of other items, and resetting them to 0 can bring false negative to these items when they are queried. One solution is to skip the first lookup principle when deletion functionality is required, while the remaining principles can still help in lookups. There is also a second solution that we do not reset these counters to 0 but mark them as "deleted", which will be treated as zero for insertion but as non-zero for lookups. When those buckets are populated later on by new items, the mark will be naturally replaced by the correct counter value. One drawback about this solution is that non-zero buckets will never return back to zero again, so when the table has worked for a long time with lots of insertions and deletions, the capability to filter out non-existing item will still gradually fade away. Therefore this second solution is more suitable for cases where deletions rarely happen.

The principles for deletion to a McCuckoo hash table with d hash functions can be summarized as below.

1) Skip all the buckets of value 0 or marked as "deleted".
2) Partition the non-zero candidate buckets according to their value, and skip those partitions whose size is smaller than the associated counter value.
3) For each of the remaining partitions, if the size is S and the associated value is V, check up to S-V+1 buckets in the partition. If the to-be-deleted item is found, continue until all V copies are found, and reset or mark their counters; otherwise return negative.

In the following subsections we will explain the data structure and discuss other aspects of McCuckoo in more details, based on the principles explained in this subsection. For the convenience of discussion, if necessary we choose d=3 as the example. The same principles can be easily instantiated and applied to other d values, but d=3 is actually sufficient for most practical scenarios.

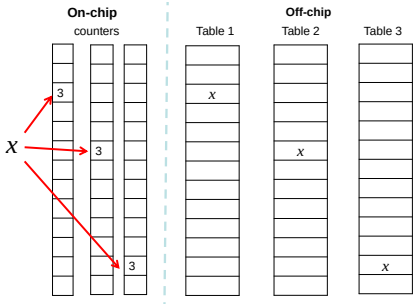

Figure 2. Item x inserted to a McCuckoo with 3 hash functions.

C. Data Structure

The data structure of McCuckoo can be seen in Fig.2 for the case of d=3 hash functions, which contains an on-chip part and an off-chip part. The off-chip part is the main Cuckoo hash table that stores the real items in the three sub-tables, and the on-chip part contains the counters that are one-to-one mapped to the off-chip buckets. For the case of d = 3, each counter costs only 2 bits. The counter array is initialized to 0 for an empty table. When the first item x is inserted, instead of choosing one out of the 3 candidate buckets to settle down, it will now occupy all 3 of them since they are all empty, and the corresponding counters will all be set to 3, as shown in the figure. Should one of the buckets be overwritten later on by some other item, the remaining buckets that still hold x will update their counters from 3 to 2 to reflex the change. The deterministic one-to-one mapping between on-chip counters and off-chip buckets makes it much easier and straightforward to determine the status of the buckets on-chip and execute the results from the counter logic off-chip.

D. Collision Resolution

When an inserted item finds all its candidate buckets are of value 1, collision occurs because it now cannot overwrite any of them, and a resolution routine should be followed to kick one occupying item out to make room. In McCuckoo any collision resolution algorithm can be used such as random-walk or MinCounter, as long as the affected counters are all correctly updated. Here in this paper we choose random-walk since it is simpler and easy to explain.

With random-walk, one bucket is selected at random and its current item will be kicked-out, who will re-check its alternative buckets to see if there is any counter that is bigger than 1 allowing an overwrite, or else another iteration of collision resolution routine is recursively called and another bucket is selected at random to evict. The whole procedure is actually equivalent to overwriting the selected bucket with the newly inserted item and then re-insert the evicted item back into the table. Comparing to the original random-walk, with the help of the counters we can pinpoint a solution right away in every round if one of the checked buckets has a counter larger than 1, whiles in the original case a new round of random kick-out will be carried out seeing all the candidate buckets occupied, risking missing the prompt resolution right there. Therefore in McCuckoo, the on-chip counters can help us find a collision resolution much faster.

However when McCuckoo table is really heavily loaded, it is still possible that the kick-out continues to see buckets of value 1 all the way and no resolution can be found. Indeed, the counters can help us find a usable bucket much faster but cannot create one if none exists. If an insertion failure happens we will resort to a failure handling routine. In this paper a stash-based approach is used and we will see later that the counters can also help to achieve an efficient and more capable stash solution.

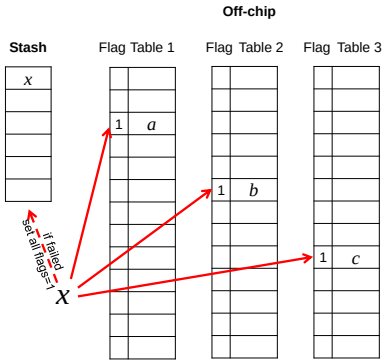

Fig. 3. Insertion routine with stash based collision resolution

E. Working with a Stash

In McCuckoo we choose to use a stash-based approach to store the items from failed insertions to avoid the costly rehashing. The main problem of the stash-based approach is that we always need to check the stash if an item is not found in the main table. Because a stash will be frequently visited, existing solutions put it in the on-chip memory that limits its size to hold only a couple of items. Even if stash is kept on-chip, keep checking it for any failed lookup to the main table is still a burden that costs time and CPU cycles, but adding another mechanism to filter out unnecessary checking on the stash may require additional on-chip space and calculation, so we may rather just check the small stash directly.

In McCuckoo we can put the stash in the off-chip memory to support a much bigger stash that can handle more insertion failures and keep the main hash table in working condition for longer time at high load ratio. We will see that with a small *off-chip* helping structure we can do so without sacrificing any lookup performance comparing to the on-chip solutions. The helping structure is a 1-bit flag that we put aside from the space of each off-chip bucket, as shown in Fig.3. They are initially set to 0 and work in the fashion of a Bloom filter: when an item fails in an insertion and put to the stash, the flags of its candidate buckets are set to 1, so at a later time when we want to decide whether to check the stash, we will see first if all the flags of the associated buckets are 1, if not we know for sure it is not there without accessing the stash.

This stash mechanism can be very efficient because of the following existing features of McCuckoo. First, the on-chip counters already filter out most lookups for non-existing items. These queries won't make it to the main table, let alone bothering the stash. Second, if an item were put to the stash during insertion, it must have run into a collision seeing counter value 1 for all its candidate buckets, and the attempt to resolve the collision ended in failure. Since the counters in McCuckoo will never increase, when an item in stash is queried, it should still see counter value 1 for all its candidate

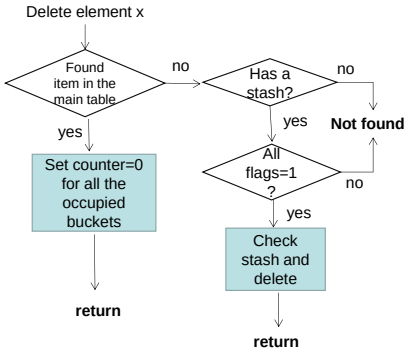

Fig. 4. Deletion routine with a stash

buckets, so for any failed lookup that involves counter value other than 1 we don't need to check the stash. On the other hand, if a lookup does fail with all counters equal to 1, next we need to check the flags to decide whether to go on and check the stash or not. Fortunately for us, the flags should have already been retrieved from the off-chip memory, because the lookup routine requires visiting all the buckets with counter value 1, during which the flags are read back as part of the bucket content. Therefore there will be no extra off-chip memory access dedicated to the stash filtering.

By working with the on-chip counters that are already there and the 1-bit off-chip flags that are negligible with respect to the size of item-storing buckets, and no extra access to the off-chip memory, we will have a stash that is only visited when the possibility that an item is really there is really high. Furthermore, since there is comparably abundant space in the off-chip memory that a stash can use, we can use more advanced hash techniques to construct the stash, so that checking it can be finished with minimal access.

F. Handle Deletion Aftermath

Following the mentality of McCuckoo, we should try to refill the empty buckets resulting from deletion as fast as possible to maximize redundancy. However, finding an appropriate item to occupy the freed bucket right after the deletion is difficult because we don't have that information even with the counters. We choose a casual update approach and probe (among originally empty ones) for newly freed buckets only during a later insertion. As soon as an empty bucket is found, either empty from the start or freed up later on, the inserted item will fill it up with a copy. This strategy is light-weighted but results in a slightly lower utilization of empty buckets which should be acceptable as long as deletions are not frequent or much less frequent than insertions.

Similar adaptation is also necessary to maintain the correctness when working with a stash. Because now buckets with value 0 might be from a deletion, and when that bucket is re-occupied the counter can be of any value, we cannot exclude the possibility of the item being stored in stash any more by exploring valid counter combinations. However, we can still check the flags that are already retrieved along with the items from the buckets during lookup (and neglect those skipped buckets) and decide not to check the stash if any of the flags is 0. Because the decision is now made with less flags, the possibility of false positive error may increase, but the false negative error will still be zero. Since comparing to

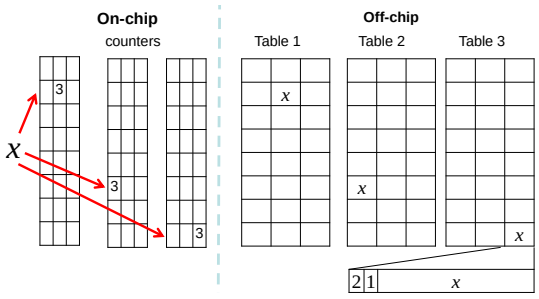

Fig. 5. Item x inserted to a d=3 l=3 multi-slot McCuckoo.

the items in the main table, the number of items in the stash is very small and most flags will be 0, therefore depending on less flags will have a higher rate of false positive error but can still screen out most of unnecessary access to the stash.

Another issue that needs to be taken care of is the deletion of stash items. Because the flags work in the fashion of a Bloom filter, they do not support deletion, either. In McCuckoo we choose not to update the flags when an item is deleted from the stash, so the false positive error rate will accumulate with each deletion of a stash item. But again, since the number of items in the stash is generally very small comparing to the number of items in the main table, the increased false positive is generally acceptable and the impact on screening performance is also small. After a series of deletions, we can choose to refresh the flags by resetting them to 0 and reinserting all the stashed items to the main table, so the status of the flags will be re-synchronized with the latest set of items in the stash.

G. Extension to multi-slot Cuckoo structure

One popular way to increase the load ratio even further is the blocked version of Cuckoo hashing that stores multiple items in one bucket that is divided into l slots. When the items are small in size and can be squeezed into the smaller slots, such "multi-slot" approach can achieve working load ratio not too far from 100%. The idea of McCuckoo can be extended to and benefit the multi-slot case as well. The example of 3-slot 3-hash (d =3, l =3) McCuckoo is show in Fig. 5. To accommodate the same number of n items, now the length of each table is reduced to roughly one third of the size as before, which is m/3. The on-chip structure is also adapted accordingly, making sure that each item still has an corresponding counter (now one counter for each slot).

However, because a new level of flexibility is introduced by the set-associativeness among slots from the same bucket, now there will be some more subtle details in item placement that cannot be fully tracked and described by just the counters. For example, with a "single-slot" McCuckoo, as one counter is associated with each bucket and the item stored in it, when we need to update the counter of a copy, we will know which exact counter to update solely on-chip as soon as we know the bucket. However in a multi-slot McCuckoo, knowing which bucket the copy sits in is not enough to identify the counter to update, because the copy can be in any slot of that bucket and we cannot track that placement details only with the on-chip structure we have. For the correctness we need to pay one access for each copy to read back the bucket and see in which slot it actually sits. One way to save those off-chip accesses is

to store which slot of bucket the *other* copies use along with the item itself in the off-chip main table as shown in Fig. 5, so when we want to update the copies of an item that we just retrieved from the main table, we know their slots in their other buckets as well. For a *d*-hash *l*-slot McCuckoo, the additional off-chip memory cost will be $(d\text{-}1)\cdot\log(l)$ bits per slot without compression.

There are also some other optimizations that work well with single-slot McCuckoo but cannot be simply extended to the multi-slot case, because for them adding full support to the set-associativeness requires the handling of some much more complicated details, for efficiency we choose not to handle all of them but the simpler ones. Although the improvement is more limited comparing to the simpler single-slot version that has been discussed extensively so far, the benefits of McCuckoo are still substantial considering the simplicity of the rules, especially for situations where the table is always moderately to highly loaded. Since the principles and considerations are basically the same as before, we will just show the full pseudo code here and discuss three of the more important changes in the multi-slot extension.

First, in many rules we still treat the bucket as a whole to decide on the actions, in which counters of all the slots in that bucket are considered together, whose sum is used as the measure of the availability (higher the better) of the bucket. Only when we see all nine counters with value 1 will we run into a collision, so the multi-slot McCuckoo is capable of sustaining really high load without caring about collision resolution. However for lookups, since there can be multiple items in one bucket and the combination of their status is difficult to trace with just the counters, we can't really exclude a full bucket solely on-chip, so the lookup routine is more like a traditional one that does not rely much on the counters. Third, the off-chip stash flags are still one-to-one assigned to each bucket not each slot, because we decide to do pre-screening at bucket level in consideration of the small number of items in the stash (than those in the main table), and doing so at bucket level is much simpler and faster.

Algorithm 1 Insertion (item *x*)

```
S ← candidate buckets of x
for each from S do
        if find one slot with counter=0
                do store a copy to the slot
                   remove bucket from S
if size(S) ≤ 1 return
sort S by sum(all counters of the bucket) descending
for each from sorted S do
        if find one slot with counter=3
                do store a copy to the slot
                   remove bucket from S
                if size(S) = 1 return
if size(S) ≤ 2 return
for each from sorted S do
        if find one slot with counter=2
                do store a copy to the slot
                   return
loop++
if loop ≤ maxloop do
        store x to a randomly picked slot from S
        Insertion (the evicted item y )
else do store x to stash
                set flags of S to 1
```

Fig. 6. Insertion routine of 3-hash 3-slot McCuckoo

H. Concurrency and multiset

Standard Cuckoo hashing is sequential in nature and does not support concurrent read/write access to the table, because during the kick-outs, not only the next kicking depends on the result of the previous one, evicted items will also become temporary unavailable from the table that may cause lookup errors. Since in practice there will be much more lookups than insertions and deletions, instead of supporting full-fledge concurrency, one-writer-many-reader concurrency will be much light weighted but sufficient for most practical schemes that are read-heavy. MemC3 [9] introduced the concept of *cuckoo path* and developed two simple modifications to the insertion order to realize one-writer-many-reader concurrency based on cuckoo path, however it did not develop efficient method to quickly find one. McCuckoo is good at quickly finding short cuckoo path for insertion, therefore combining the two will give McCuckoo efficient concurrency support.

McCuckoo can also support multiset, but not by distributing items of the same key among that key's multiple copies, because those redundant copies should always be identical; instead it can act as an indexing structure pointing to the address where all those items are actually stored.

IV. Experimental Evaluation

We have implemented McCuckoo in both *d*-ary and blocked form with an off-chip stash as the failure resolution method. We also implemented standard *d*-ary Cuckoo and the blocked Cuckoo Hash Table BCHT from [18] and used them as the baseline for comparison in the experiments. Since McCuckoo always tries to fill up all the empty candidate buckets, effectively it trades some earlier writes for less reads and writes during later kick-outs, and uses the additional copies to provide hints on the bucket status to the lookup and failure resolution procedure to minimize access to the slow external memory. We will show how such multi-copy strategy

Algorithm 2 lookup (item *x*)

```
S ← candidate buckets of x
for each from S do
        if sum(all counters of the bucket)=0
                do skip and remove bucket from S
        else search the bucket for x
        if found return x
if all flags of S = 1
        do check the stash
           return x | not_exist
else return not_exist
```

Fig. 7. Lookup routine of 3-hash 3-slot McCuckoo

Algorithm 3 delete (item *x*)

```
S ← candidate buckets of x
for each from S do
        if sum(all counters of the bucket)=0
                do skip and remove bucket from S
        else search the bucket for x
        if found F←slot(x)
if F ≠ Φ
        do set counters from F to 0
           return
if all flags of S = 1
        do check the stash
           delete x
```

Fig. 8. Deletion routine of 3-hash 3-slot McCuckoo

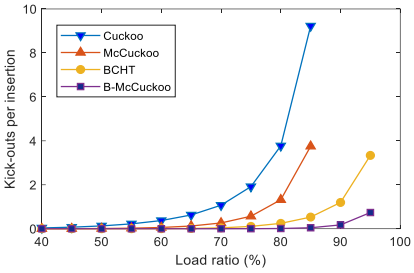

Fig. 9. Number of kick-outs per insertion

performs against their single-copy counterparts in different settings, especially under moderate to high load ratios.

Because the design goal of McCuckoo is very much to minimize the access to the slow off-chip memory, so for the first part of the evaluation we want to focus on showing how effective we are with regard to this goal through simulations. In order to evaluate the overhead of McCuckoo on counter manipulation and the additional logic on bucket selection, in the second part of the evaluation we also extend McCuckoo to a FPGA based platform to evaluate the insertion and lookup latency as well as throughput.

A. Exparimental Environments

1) Hardware Platform

All the simulations were carried out on a machine with 4-cores (8 threads, Intel Core i7@2.60 GHz) and with 8 GB of DRAM memory. The target platform is an Altera Stratix V GX FPGA from Intel with 4.5MB on-chip SRAM with support to external DDR3 SDRAM memory at 800Mhz. In our FPGA implementation, the logic runs at 333Mhz and the DDR3 memory controller runs at 200Mhz, respectively.

2) Dataset and Implementation

DocWords[a]**:** This dataset includes five text collections in the form of bag-of-words and we choose the one collected from NYTimes news articles. It contains approximately 70 million items in total. The DocID and WordID are combined to form the key of each item and inserted into the hash tables.

All the hash schemes were implemented in C++. The hash functions used in the experiments are BOB Hash[b].

For the FPGA implementation, due to the limit space of the on-chip SRAM, we extract a 6 million subset from the DocWords dataset to use for the evaluation, and a much simpler hash implementation that only involves modulo and bit operations is used instead of BOB Hash.

3) Experimental Settings

In the experiments we always use 3 hash functions to calculate the candidate locations, and 3 slots per bucket in the blocked schemes. For the convenience of discussion, we will call the ternary Cuckoo and the 3-hash 3-slot BCHT schemes as Cuckoo and BCHT for short, and their multi-copy counterparts as McCuckoo & B-McCuckoo. Each experiment repeated 10 times and the average is used as the final results.

B. Insertion Performance

We compared the number of kick-outs generated for each insertion under different load ratios, as well as the involved reads and writes to the memory. These measures can be used to evaluate the capability to quickly find a location for the

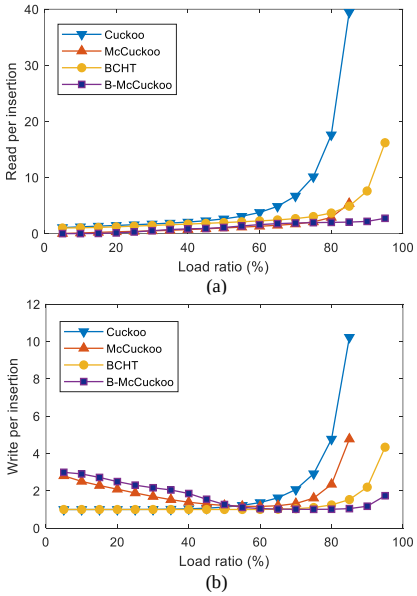

Fig. 10. Memory access per insertion

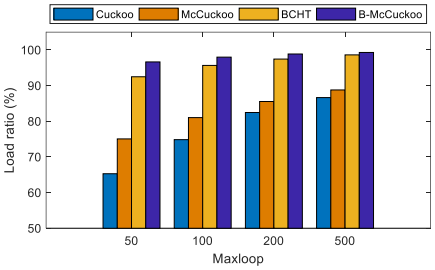

Fig. 11. Load ratio at first insertion failure

inserted item and the involved cost. Moreover, when a hash table is filling up with items, increased availability can defer the occurrence of collisions and insertion failure, therefore we also measured the load ratio when the first kick-out occurs and when the first insertion failure occurs. The results are shown in Fig. 9 through Fig. 11.

From Fig. 9 we can see that at very low load ratio almost all insertions can finish without kick-outs. At higher load ratio, it becomes more difficult to find an empty bucket right away therefore more kick-outs will be experienced. As expected, multi-copy schemes can help resolve collisions and reduce the average number of kick-outs, for example by 59.3% for ternary Cuckoo at load ratio of 85% and 77.9% for 3-way BCHT at load ratio of 95%.

Because McCuckoo needs to handle multiple copies that may need additional memory accesses during the insertion, we also measured the average number of reads and writes at different load ratio. We can see from Fig. 10a that the number of reads is much reduced and can be as low as 0 at low load ratio for the multi-copy schemes because in many cases with the on-chip counters we can already see which buckets are free. Furthermore, the "multiplier effect" is more severe for the single-copy schemes if we compare Fig.10a with Fig. 9, because they need to read back each candidate bucket to know

[a] "DocWords": http://archive.ics.uci.edu/ml/machine-learning-databases/bag-of-words/
[b] "bob hash website": http://burtleburtle.net/bob/hash/evahash.html

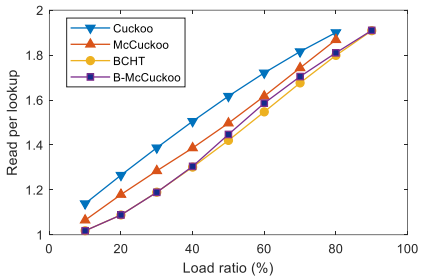

Fig. 12. Memory access per lookup for existing items

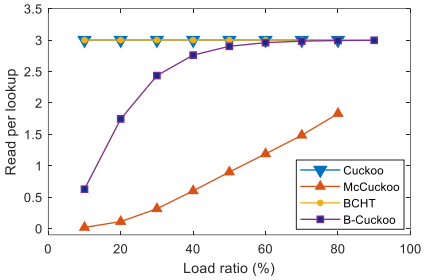

Fig. 13. Memory access per lookup for non-existing items

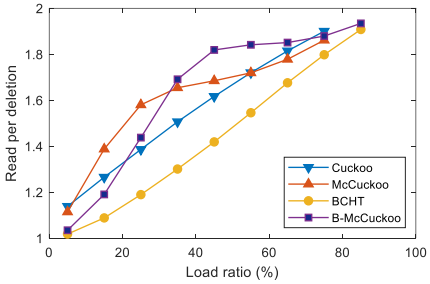

Fig. 14. Memory access per deletion

if they are empty or not during the kick-outs, while the multi-copy schemes can figure out the empty buckets with the on-chip counters.

The number of writes, on the other hand, is higher with the multi-copy schemes as shown in Fig.10b, but only at lower load ratio, because more copies are stored to the table during each insertion. At higher load ratio, the writes caused by writing multiple copies will decrease, while writes caused by kick-outs will increase. The cross-over happens at about half load for single-slot schemes and at a bit higher load for the multi-slot schemes, which means for the most likely working conditions of Cuckoo with the table moderately to heavily loaded, the number of writes is also lower with the multi-copy schemes. Since more reads will take place than writes during an insertion, the total number of accesses at higher load ratio is much reduced in the multi-copy schemes.

McCuckoo can maintain collision-free status for much longer because more buckets are kept available when they are just occupied by item copies. Table I shows that the first collision happens at a much later time with the multi-copy schemes when the table is much more filled up. The first occurrence of insertion failure is even more crucial because from then on, more insertions will stop at *maxloop* which costs heavily for us. The load ratio when the first insertion failure occurs is shown in Fig. 11, where *maxloop* is set from 50 to 500. Higher *maxloop* can help to reach higher load, but also induce heavier penalization if an insertion still fails after the lengthy trial. From the figure we can see that with multi-copy we can reach higher load ratio free of insertion failures with the same *maxloop*, or reach the same load ratio with smaller *maxloop* values than the single-copy schemes.

TABLE I. LOAD RATIO WHEN FIRST COLLISION OCCURS

Cuckoo	McCuckoo	BCHT	B-McCuckoo
9.27%	23.20%	46.03%	61.42%

C. *Lookups for Existing and No-existing Items*

The performance for lookup is shown in Fig.12 & Fig.13 for queries on existing and non-existing items respectively. Because with McCuckoo we can identify and skip the buckets that do not contain the target item for sure, the average memory access is lower than in the single-copy schemes.

For the lookup on non-existing items, the single-copy schemes always need to check all the buckets to be sure the item is not in the table, but with the pre-screening capability of the on-chip counters, we can recognize the non-existing item much faster, sometimes even without accessing the main table at all at lower load ratio when more empty buckets and more copies exist. The average lookup access increases fast for B-McCuckoo because when more buckets start to contain items, in order to really differentiate items from the same bucket, the on-chip counters are not enough and you may need to read them out to be sure. This means that at very high load ratio, for blocked McCuckoo it may be a good idea just to do the lookup the old way, since almost all the slots are occupied by the sole copy of different items.

D. *Deletion*

The performance of deletion is shown in Fig.14. in terms of deletion. Because items may have multiple copies in the table, on average more read is required to confirm all the existing copies. Fortunately deletions are not frequently executed comparing to lookups and it still possesses the worst-case constant bound guarantee. In the meantime we are still refining our algorithm to improve the performance.

The number of writes during a deletion will always be one for the single-copy schemes and zero for the multi-copy schemes, because in the latter only the on-chip counters are updated (reset to 0), so we didn't show them in a figure.

E. *Stash at High Load Ratio*

Since McCuckoo is the first hash scheme that uses an off-chip stash structure, we will not evaluate its performance through comparison; instead, we want to show the necessity of a bigger stash and the feasibility of putting one off-chip.

Table II and Table III present the experimental results that simulate a McCuckoo table and a blocked McCuckoo table that works very close to the maximum load so the main table is really crowded. Each row shows five parameters, which are the current load ratio, the value of the threshold *maxloop*, the number of items in the stash, its percentage against all the inserted items, and the percentage of queries for non-existing items that evoke a visit to the stash. We can see from the results that unless we leave a sufficiently large margin in the table spare space to accommodate all the possible surges, a small wave of items inserted to an already crowded hash table

may very likely fail the insertion and land into the stash, unless otherwise we can also accept a rehash. A larger stash is more robust to such a situation, but in the traditional on-chip solutions, there is no space for a large stash. McCuckoo can support a large stash that is put to the comparably abundant off-chip memory. Furthermore, unlike the on-chip stash that will be visited every time an item is not found in the main table, the efficient pre-screening mechanism can avoid most unnecessary queries to the stash in McCuckoo, so the off-chip stash is capable of handling much more severe table overflow at very limited cost.

TABLE II. STASH PERFORMANCE FOR 3-HASH 1-SLOT MCCUCKOO

Load	*max loop*	Stash Statistics		
		number of items	*% in all items*	*% visits in lookups*
88%	200	107.8	0.0020%	0.0000%
	500	0.0	0.0000%	0.0000%
89%	200	788.2	0.0148%	0.0000%
	500	5.2	0.0001%	0.0000%
90%	200	4401.2	0.0815%	0.0000%
	500	201.4	0.0037%	0.0000%
91%	200	15989.2	0.2928%	0.0000%
	500	4441.2	0.0813%	0.0000%
92%	200	38702.0	0.7011%	0.0029%
	500	24648.6	0.4465%	0.0001%
93%	200	70060.4	1.2556%	0.0038%
	500	58443.8	1.0474%	0.0007%

TABLE III. STASH PERFORMANCE FOR 3-HASH 3-SLOT MCCUCKOO

Load	*max loop*	Stash Statistics		
		number of items	*% in all items*	*% visits in lookups*
97.5%	200	0.0	0.0000%	0.0000%
	500	0.0	0.0000%	0.0000%
98%	200	0.0	0.0000%	0.0000%
	500	0.0	0.0000%	0.0000%
98.5%	200	0.0	0.0000%	0.0000%
	500	0.0	0.0000%	0.0000%
99%	200	4.2	0.0001%	0.0000%
	500	0.0	0.0000%	0.0000%
99.5%	200	1117.2	0.0187%	0.0000%
	500	56.4	0.0009%	0.0000%
100%	200	18309.6	0.3052%	0.0000%
	500	16791.0	0.2799%	0.0000%

F. Latency and Throughput

The evaluation on latency and throughput is based on a Altera FPGA development board that can run the McCuckoo logic and access the on-chip SRAM at 333Mhz. Hash calculation and the logic is implemented in hardware that can be performed in 1CLK. The on-chip SRAM can be read in 3CLK and written in 1CLK. For the off-chip DDR3 SDRAM, the controller is clocked at 200Mhz, and read costs about 18CLK on average and write costs 1CLK. Due to the time limit, no parallelism or pipeline is implemented, so the write latency is much lower than the read latency because the logic can return to process the next instruction after writing to the memory controller but has to wait for the data back from the external memory during a read. Furthermore, when the record size is small, skip checking some buckets will not make much of a difference in read latency so the benefit of McCuckoo and B-McCuckoo is less significant, while the time to access the on-chip counters becomes relative large because they need to be checked all the time. Fig. 15 shows the average insertion

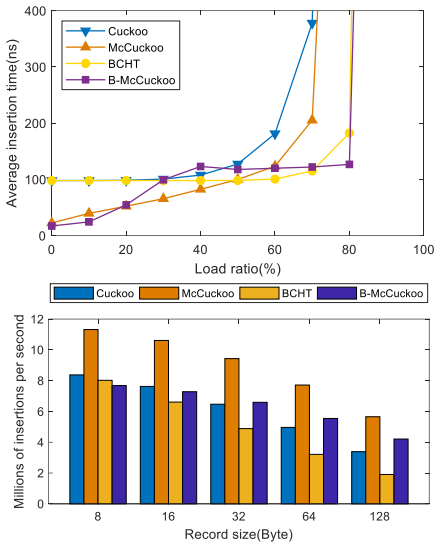

Fig. 15. Latency and throughput for insertion

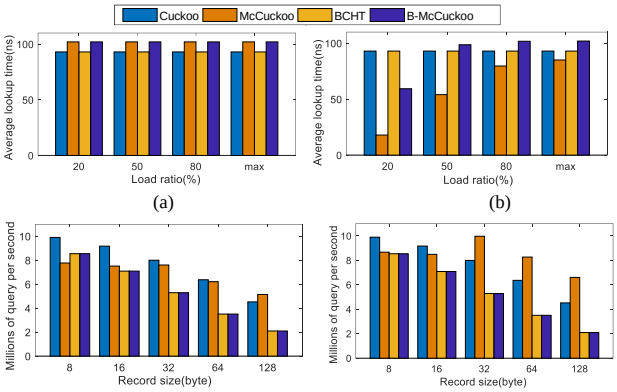

Fig. 16. Latency and throughput for lookup

latency with respect to different load ratios and the throughput at 50% load when the record size changes from 8 Bytes to 128 Bytes. We can see improved throughput from the multi-copy schemes, but the latency of B-McCuckoo is a bit higher at moderate load because kicking-outs happen much less often in the multi-slot Cuckoos due to the added set-associativeness, so the time used for checking counters is not paid back. Fig. 16 show lookup latency in (a)&(b) and throughput in (c)&(d) for existing items to the left and non-existing ones to the right. When the size of the item increases, checking less buckets will benefit the throughput as expected. The added lookup time is due to the checking on the counters, which is significant in this implementation comparing to the cost of reading more data from the external memory. In this case we can actually just skip checking the counters during the lookup to avoid the added latency, which in effect does not affect the correctness of the lookup results. Generally speaking, the end-to-end measurement is very much hardware specific, and McCuckoo demonstrates the expected behavior.

V. CONCLUSIONS

All existing Cuckoo-based hash tables only choose one bucket to store an inserted item even when more candidates are available. With no knowledge on the items that come in

later, such rash decisions are very often sub-optimal and need to be corrected by a series of item relocation. In this paper a multi-copy version of Cuckoo hashing is proposed to keep multiple copies of the items in all the available candidate buckets to circumvent the rash decision on location and improve the availability of the buckets. The number of copies each item has is tracked by counters, with which we can find a resolution faster for a collision, reduce the number of relocations, find an item with less memory accesses and identify and pre-screening queries for non-existing items. Experimental results show that the proposed McCuckoo achieves the expected performance in comparison with the existing single-copy Cuckoo mechanisms.

ACKNOWLEDGMENT

The authors thank all the reviewers for their time and precious comments. We would also like to thank Tao Li and Qiang Zeng for the FPGA implementation based on FAST platform [35]. This work was supported by Shenzhen Peacock Project (app. No. 201803233000214), Shenzhen Key Lab Project (ZDSYS201703031405137), the Shenzhen Municipal Development and Reform Commission (Disciplinary Development Program for Data Science and Intelligent Computing), National Engineering Laboratory for Video Technology - Shenzhen Division, and NSFC (61672061). Dagang Li and Rong Du are co-primary authors. Rong Du finished this work under the guidance of her supervisor Dagang Li. Tong Yang is the corresponding author with email yangtongemail@gmail.com.

REFERENCES

[1] Knuth, Donald (1998). 'The Art of Computer Programming'. 3: Sorting and Searching (2nd ed.). Addison-Wesley. pp. 513–558.

[2] Cormen, Thomas H.; Leiserson, Charles E.; Rivest, Ronald L.; Stein, Clifford (2001). "Chapter 11: Hash Tables". Introduction to Algorithms (2nd ed.). MIT Press and McGraw-Hill. pp. 221–252.

[3] R. Pagh, F. F. Rodler, Cuckoo hashing, Journal of Algorithms 51 (2) (2004) 122–144.

[4] M. Naor, G. Segev, and U. Wieder, “History-Independent Cuckoo Hashing,” in *ICALP 2008*, vol. 5126, pp. 631–642.

[5] B. Debnath, S. Sengupta, and J. Li, “ChunkStash: Speeding Up Inline Storage Deduplication Using Flash Memory,” in USENIX ATC 2010, Berkeley, CA, USA, 2010, pp. 16–16.

[6] H. Lim, B. Fan, D. G. Andersen, and M. Kaminsky, “SILT: a memory-efficient, high-performance key-value store,” in SOSP ’11, Cascais, Portugal, 2011, pp. 1–13.

[7] M. T. Goodrich, M. Mitzenmacher, O. Ohrimenko, and R. Tamassia, “Privacy-preserving Group Data Access via Stateless Oblivious RAM Simulation,” in *SODA ’12*, Philadelphia, PA, USA, 2012, pp. 157–167.

[8] P. Bosshart *et al.*, “Forwarding metamorphosis: fast programmable match-action processing in hardware for SDN,” in *ACM SIGCOMM ’13*, Hong Kong, China, 2013, p. 99.

[9] B. Fan, D. G. Andersen, and M. Kaminsky, “MemC3: Compact and Concurrent MemCache with Dumber Caching and Smarter Hashing,” in *USENIX NSDI 2013*, Lombard, IL, 2013, pp. 371–384.

[10] D. Zhou, B. Fan, H. Lim, M. Kaminsky, and D. G. Andersen, “Scalable, high performance ethernet forwarding with CuckooSwitch,” in *CoNEXT ’13*, Santa Barbara, CA, 2013, pp. 97–108.

[11] X. Li, D. G. Andersen, M. Kaminsky, and M. J. Freedman, “Algorithmic improvements for fast concurrent Cuckoo hashing,” in *EuroSys ’14*, Amsterdam, The Netherlands, 2014, pp. 1–14.

[12] B. Fan, D. G. Andersen, M. Kaminsky, and M. D. Mitzenmacher, “Cuckoo Filter: Practically Better Than Bloom,” in *CoNEXT ’14*, 2014, pp. 75–88.

[13] B. Pinkas, T. Schneider, G. Segev, and M. Zohner, “Phasing: Private Set Intersection Using Permutation-based Hashing,” in *USENIX SEC’15*, Berkeley, CA, USA, 2015, pp. 515–530.

[14] D. Cash, A. Küpçü, and D. Wichs, “Dynamic Proofs of Retrievability Via Oblivious RAM,” *Journal of Cryptology*, vol. 30, no. 1, pp. 22–57, Jan. 2017.

[15] Y. Sun, Y. Hua, S. Jiang, Q. Li, S. Cao, and P. Zuo, “SmartCuckoo: A Fast and Cost-Efficient Hashing Index Scheme for Cloud Storage Systems,” in USENIX ATC 17, Santa Clara, CA, 2017, pp. 553–565.

[16] Q. Li, Y. Hua, W. He, D. Feng, Z. Nie, and Y. Sun, “Necklace: An efficient cuckoo hashing scheme for cloud storage services,” in IEEE IWQoS 2014, Hong Kong, 2014, pp. 153–158.

[17] Y. Sun, Y. Hua, D. Feng, L. Yang, P. Zuo, and S. Cao, “MinCounter: An efficient cuckoo hashing scheme for cloud storage systems,” in MSST 2015, Santa Clara, CA, USA, 2015, pp. 1–7.

[18] U. Erlingsson, M. Manasse, and F. Mcsherry, “A Cool and Practical Alternative to Traditional Hash Tables,” in *WDAS 2006*, 2006.

[19] K. A. Ross, “[19]s on Modern Processors,” in *IEEE ICDE 2007*, Istanbul, Turkey, 2007, pp. 1297–1301.

[20] M. Dietzfelbinger and C. Weidling, “Balanced allocation and dictionaries with tightly packed constant size bins,” *Theoretical Computer Science*, vol. 380, no. 1–2, pp. 47–68, Jun. 2007.

[21] M. Mitzenmacher, K. Panagiotou, and S. Walzer, “Load Thresholds for Cuckoo Hashing with Double Hashing,” in *SWAT 2018*, 2018, pp. 29:1-29:9.

[22] A. Kirsch, M. Mitzenmacher, and U. Wieder, “More Robust Hashing: Cuckoo Hashing with a Stash,” *SIAM Journal on Computing*, vol. 39, no. 4, pp. 1543–1561, Jan. 2010.

[23] M. Aumüller, M. Dietzfelbinger, and P. Woelfel, “Explicit and Efficient Hash Families Suffice for Cuckoo Hashing with a Stash,” *Algorithmica*, vol. 70, no. 3, pp. 428–456, Nov. 2014.

[24] S. Pontarelli, P. Reviriego, and M. Mitzenmacher, “EMOMA: Exact Match in One Memory Access,” *IEEE TKDE*, pp. 1–1, 2018.

[25] D. Li, J. Li, and Z. Du, “Deterministic and Efficient Hash Table Lookup Using Discriminated Vectors,” in *Globecom 2016*, Washington D.C., 2016, pp. 1–6.

[26] K. Huang, G. Xie, R. Li, and S. Xiong, “Fast and deterministic hash table lookup using discriminative bloom filters,” *Journal of Network and Computer Applications*, vol. 36, no. 2, pp. 657–666, Mar. 2013.

[27] D. Fotakis, R. Pagh, P. Sanders, and P. Spirakis, “Space Efficient Hash Tables with Worst Case Constant Access Time,” *Theory of Computing Systems*, vol. 38, no. 2, pp. 229–248, Feb. 2005.

[28] A. Frieze, P. Melsted, and M. Mitzenmacher, “An Analysis of Random-Walk Cuckoo Hashing,” *SIAM Journal on Computing*, vol. 40, no. 2, pp. 291–308, Jan. 2011.

[29] N. Fountoulakis, K. Panagiotou, and A. Steger, “On the Insertion Time of Cuckoo Hashing,” *SIAM Journal on Computing*, vol. 42, no. 6, pp. 2156–2181, Jan. 2013.

[30] A. Frieze and T. Johansson, “On the insertion time of random walk cuckoo hashing,” in *SODA 2017*, 2017, pp. 1497–1502.

[31] D. A. Alcantara et al., “Real-time parallel hashing on the GPU”, ACM Transactions on Graphics, vol. 28, no. 5, p. 1, Dec. 2009.

[32] Bloom, H. Burton, "Space/Time Trade-offs in Hash Coding with Allowable Errors", Communications of the ACM, vol. 13, no. 7, pp. 422–426, 1970.

[33] P. Zou, Y. Hua, J. Wu, “Write-Optimized and High-Performance Hashing Index Scheme for Persistent Memory”, in OSDI 2018.

[34] O. Kocberber, B. Falsafi, and B. Grot, “Asynchronous memory access chaining,” *Proceedings of the VLDB Endowment*, vol. 9, no. 4, pp. 252–263, Dec. 2015.

[35] FAST platform website. http://www.fastswitch.org.