On the Performance of Multiple Choice Hash Tables with Moves on Deletes and Inserts

Adam Kirsch[*] and Michael Mitzenmacher[*]
School of Engineering and Applied Sciences
Harvard University
{kirsch,michaelm}@eecs.harvard.edu

Abstract— **In a multiple choice hash table scheme, each item is stored in one of $d \geq 2$ hash table buckets. The ability to choose from multiple locations when storing an item improves space utilization, while the simplicity of such schemes makes them highly amenable to hardware implementation, as in a router. Some variants, such as cuckoo hashing, allow items to be moved among their d choices in order to improve load balance and avoid hash table overflows. We consider schemes that move items on insertion and deletion operations, as arguably one would be willing to incur more time on such operations as opposed to more frequent lookup operations. To keep the schemes as simple as possible for hardware implementation, we focus on schemes that allow a single move on an insertion or deletion. Our results show significant space savings when moving items is allowed, even under the limitation of one move per insertion and deletion operation.**

I. Introduction

High-performance hashing has become a fundamental subroutine for a wide variety of high performance network processing tasks, including header lookup for routing, measurement, and monitoring. In considering hashing alternatives, many possible considerations arise, perhaps the most important being how much time is spent performing lookup, insert, and delete operations, and how much space the table requires. Here time primarily corresponds to the number of hash tables entries that are read and, when items can be moved within the hash table, the number of items that are moved, as computation is often free compared to the time to perform a memory access or a write. Here space corresponds to the amount of space required to hold n items, with high probability.

For example, cuckoo hashing [10] is a scheme where each item has d possible locations where it can be stored in a hash table, for a small constant d. Lookups are therefore constant time. The space required is linear in n, with small constant factors in practice. Deletions are performed by simply removing the item, and therefore also require only constant time. Insertions, however, while constant time on average, generally take time logarithmic in n with non-negligible probability, which may not be suitable for many applications. In order to avoid this high cost for an insertion, previous work has considered multilevel hash tables (MHTs) [1], [4], which also use d possible locations per item and linear space. To lower the insertion time to constant, additional hardware, namely small content addressable memories (CAMs), were used to handle potential overflow caused by collisions in the hash table. In [3], MHTs allowing only one additional move of an item on an insertion are studied. In [5], an alternative construction where a CAM is used as a queue for move operations is discussed.

In this paper, we consider further variations of multilevel hash tables that also allow moves when items are deleted. As we shall see, handling moves on a deletion is generally harder than handling moves on an insertion. This is because schemes for moving items when a new item is inserted can be based on moving an existing item that collides with the new item out of its way. That is, it is clear what items to try to move. On a deletion, we generally want to try to move an existing item in the table into the now-vacant space, but it is not immediately clear where an appropriate item can be found. We consider two approaches. First, we use additional memory to store small hints of where items to move can be found when a deletion occurs. Second, we consider an idea from [7]: we make the locations of an item dependent in such a way so that when an item is deleted, another item in the table can be easily found to be moved into its location.

While we aim for analyses of our schemes, gener-

[*]This work was supported by NSF grant CNS-0721491 and a research grant from Cisco Systems, Inc.

ally this does not seem to be possible using standard techniques. Analyses of several of our basic schemes remain open, and we rely on simulations to obtain insight into performance. Our experiments are designed to highlight tradeoffs with these schemes and examine the comparative value of schemes that move items on a deletion against those that move items on an insert. Naturally, we consider whether allowing a move on both an insertion and a deletion can yield substantially better results than moving on just an insertion or deletion alone.

II. BACKGROUND: MULTILEVEL HASH TABLES

The basis for our hash table schemes is the multilevel hash table (MHT) of Broder and Karlin [1]. This is a hash table consisting of d sub-tables $T_1, \ldots, T_d$, with each T_i having one hash function h_i. (In this work, we make the heuristic assumption that hash functions are fully random; for more on this, see for example [9].) We view these tables as being laid out from top (T_1) to bottom (T_d). To insert an item x, we find the minimal i such that $T_i[h_i(x)]$ is unoccupied, and place x there. We assume that each bucket can store at most one item, although generalizations to larger bucket sizes are certainly possible. If $T_1[h_1(x)], \ldots, T_d[h_d(x)]$ are all occupied, then we declare a *crisis*. There are multiple things that we can do to handle a crisis. The approach in [1] is to resample the hash functions and rebuild the entire table. That work shows that it is possible to insert n items into a properly designed MHT with $O(n)$ total space and $d = \log\log n + O(1)$ in $O(n)$ expected time, assuming only 4-wise independent hash functions.

Assuming fully random hash functions, Kirsch and Mitzenmacher [4] modify the analysis of [1] to show that, if the sub-tables are sized properly, then no re-hashings are necessary in practice. Essentially, the idea is that if the T_i's are (roughly) geometrically decreasing in size, then the total space of the table is $O(n)$. If the ratio by which the size of T_{i+1} is smaller than T_i is, say, twice as large as the expected fraction of items that are not stored in $T_1, \ldots, T_i$, then the distribution of items over the T_i's decreases doubly exponentially with high probability. This double exponential decay allows the choice of $d = \log\log n + O(1)$. For a more detailed description of this intuition, see [1] or [4].

A very useful property of MHTs is that they naturally support deletions, as one can just perform a lookup on an item to find its location in the table, and then mark the corresponding item as deleted. Also, MHTs appear well-suited to a hardware implementation. In particular, their open-addressed nature seems to make them preferable to approaches that involve chaining, and their use of separate sub-tables allows for the possibility that all of the hash locations for a particular item can be accessed in parallel.

Two important methods for improving the performance of MHTs are proposed by Kirsch and Mitzenmacher in [4] and [3]. The key contribution of [4] is a very compact and simple Bloom filter-based *summary* data structure that, for any item in the MHT, can efficiently answer a query as to what sub-table contains that item. For items not in the MHT, the summary has some false positive probability, like a standard Bloom filter. In practice, the summary data structure is small enough that it can be stored in fast memory when the hash table is so large that it can only be stored in much slower memory. Thus, the summary allows for a hash table lookup to be performed with only one access to slow memory, whereas the naive approach would require d accesses (possibly in parallel). One could also use a Bloomier filter [2], although a summary specifically designed for this setting can perform better.

The paper [3] shows that the space utilization of a MHT can be substantially improved by allowing a single item in the table to be moved during an insertion operation. This observation is a major motivational force behind this work, and so we elaborate in some detail. In particular, [3] proposes the *Second Chance* insertion scheme, described as follows. Essentially, the idea is that as we insert items into a standard MHT with sub-tables $T_1, \ldots, T_d$, the sub-tables fill up from top to bottom, with items cascading from T_i to T_{i+1} with increasing frequency as T_i fills up. Thus, a natural way to increase the space utilization of the table is to slow down this cascade at every step.

This idea is implemented in the Second Chance scheme in the following way. We mimic the insertion of an item x using the standard MHT insertion procedure, except that if we are attempting to insert x into T_i, if the buckets $T_i[h_i(x)]$ and $T_{i+1}[h_{i+1}(x)]$ are occupied, rather than simply moving on to T_{i+2} as in the standard scheme, we check whether the item y in $T_i[h_i(x)]$ can be moved to $T_{i+1}[h_{i+1}(y)]$. If this move is possible (i.e., the bucket $T_{i+1}[h_{i+1}(y)]$ is unoccupied), then we perform the move and place x at $T_i[h_i(x)]$. Thus, we effectively get a *second chance* at preventing a cascade from T_{i+1} to T_{i+2}.

Just as in the standard MHT insertion scheme, there may be items that cannot be placed in the MHT during the insertion procedure. Previously, we considered this to be an extremely bad event and strived to bound its probability. An alternative approach if an item is not successfully placed in the MHT during its insertion is to place it in a *stash*, which, in practice, would be implemented with a CAM. To perform a lookup, we simply check the stash in parallel with the MHT.

It turns out that since the Second Chance scheme only allows moves from top to bottom, it is analyzable by a *fluid limit* or *mean-field* technique, which is essentially a way of approximating stochastic phenomena by a deterministic system of differential equations. The technique also applies to the standard MHT insertion procedure, as well as a wide variety of extensions to the basic Second Chance scheme. This approach makes it possible to perform very accurate numerical analyses of these systems, and in particular it allows for some interesting optimizations. We refer to [3] for details.

The Second Chance scheme is also much more amenable to a hardware implementation than it may at first seem. To insert an item x, we simply read all of the items $y_1 = T_1[h_1(x)], \ldots, y_d = T_d[h_d(x)]$ in parallel. Then we compute the hashes $h_2(y_1), \ldots, h_d(y_{d-1})$ in parallel. (Here, for notational simplicity, we are assuming that all of $T_1[h_1(x)], \ldots, T_d[h_d(x)]$ are occupied, so that the y_i's are well-defined; it should be clear how to handle the general case.) At this point, we now have all of the information needed to execute the insertion procedure without accessing the hash table (assuming that we maintain a bit vector indicating which buckets of the table are occupied).

The Second Chance scheme also supports deletions in the natural way: an item can simply be removed from the table. However, as with many hash table constructions, the intermixing of insertions and deletions fundamentally changes the behavior of the system, making analysis via fluid limits inaccurate (albeit still potentially useful). For details, see [3].

III. USING HINTS

With a standard multilevel hash table, each item obtains an independent hash for each level of the hash table. With no correlation between levels, collisions at one level do not affect another, allowing items to easily find free locations. As a downside, however, it is not entirely clear what to do when item are deleted from the hash table. Potentially when an item is deleted from level $i-1$, some item at level i or some deeper level could be moved back to that spot, and possibly then additional items from further levels could recursively be moved up in the table. Intuitively and in practice, pulling items up to lower numbered levels decreases the subsequent probability of a failure, where a newly inserted item cannot be placed. But there is no immediate method to find an appropriate item to move to the now empty location, and exhaustive search is far too expensive. (This in part explains the previous focus on moving items only on insertion operations of [3].)

One approach to circumvent this problem would be to store *hints* in cells, where the hints would consist of a short pointer encoding where to find an item that has previously collided at that cell. A pointer could be expressed as an ordered pair of a level and a cell in that level, which can be written in a small number of bits. (As will become clear, such hints take roughly $\log_2 n$ bits.) We emphasize that the hint is simply a hint; the item at the given location may no longer be an item that collided at the cell with the hint, because of intervening insertions and deletions, and hence its hash value for the level it could be moved to must be checked before moving the item.

A variety of hints and move strategies are possible. One approach would be to store a hint whenever a collision occurs at a cell, always replacing any existing hint. Another approach would be to store the collision corresponding to the item that has been placed at the deepest level. Yet another alternative would be to only store hints for items at the next level; this slightly shortens the length required for a hint, and would still allow items at deep levels to be moved up recursively. We clarify that our goal is not to provide a complete picture of all the various permutations of hint strategies that can be imagined, but to obtain some insight into the potential of schemes that move items on a deletion as compared to other approaches. As our tests of performance will be based primarily on simulations, we remark that performance of any given scheme may depend on a number of variables, particularly the distribution of the lifetime of an item in the table before deletion.

While multiple moves per deletion are possible, following [3] we focus attention on schemes that are limited to one move per deletion operation.

A. Experimental Results

We provide some basic simulation results. We emphasize that these results are not meant to cover the wide range of possibilities, but to give insight into these processes. When choosing a hash table structure, one must consider the tradeoffs among the number of hash functions, the load factor (ratio of items to cells in the hash table), and the probability of an overflow. There are also potential issues in sizing the subtables, deciding the number of items per bucket, determining the number of moves allowed, and so on. In designing simulations, one must consider the distributions of lifetimes among items, how the load varies over time, and the overall length of the simulation.

Here we begin with the goal of aiming for a load of at least 50%. We test settings where each bucket holds only one item, and the size of each sublevel of the hash table falls by a factor of $1/2$ from the previous level. The load is maintained at n items, where in our tests we use $n = 2^{13}, 2^{14}, 2^{15}$; we initially load the table with n inserts, and then alternate deletion and insertion operations for 2^{18} steps, which appears more than sufficient time for the process to reach steady-state. This alternation of insert and deletes is roughly equivalent to assuming item lifetimes follow the memoryless exponential distribution. We track both the load at each level of the hash table at the end of the process to obtain an approximate steady-state average and the maximum load at each level throughout the process. Finally, we allow a small stash (generally, up to size 10, although slightly larger in some cases) to hold items that cannot be placed in the hash table. Each configuration was run $10,000$ times. We consider the number of hash functions required and the maximum size of resulting stash in order to achieve no crisis for those 10,000 trials.

We first note that with no moves at all, a load factor of $1/2$ is just barely possible with a MHT. As shown in Table I, we required increasing the stash size to 32, and a much larger than desirable number of hash functions. For comparison purposes, we also consider the Second Chance scheme of [3], which uses at most one move on each insert and no moves on a deletion. In our initial experiments we found that the approach of replacing any existing hint whenever a collision occurred performed best of our proposed schemes, so we report the results for this algorithm. We consider two variants of this scheme. In the first, only one move is allowed per deletion, so an item can only be moved to the vacated cell. In the second, when an item is moved because of a deletion, another item can recursively be moved into its empty location, and so on as much as possible. As can be seen from our experiments, the recursive variation adds some benefit in terms of performance, but both schemes perform less well than the Second Chance scheme.

TABLE I
NUMERICAL RESULTS FOR HINT-BASED SCHEMES.

Scheme	Items = Size Level 0	Hashes (Levels)	Max. Stash (items)	Avg. Stash (items)
No Move	8192	11	31	4.225
No Move	16384	12	29	4.375
No Move	32768	13	31	3.896
Second Chance	8192	6	2	0.001
Second Chance	16384	6	2	0.001
Second Chance	32768	6	1	0.003
Hint+1 Move	8192	7	2	0.004
Hint+1 Move	16384	7	2	0.006
Hint+1 Move	32768	7	3	0.013
Hint+Moves	8192	6	5	0.063
Hint+Moves	16384	6	5	0.131
Hint+Moves	32768	6	7	0.246
Hint+1 Move+SC	8192	4	10	1.198
Hint+1 Move+SC	16384	4	15	2.345
Hint+1 Move+SC	32768	4	18	4.678
Hint+Moves+SC	8192	4	6	0.236
Hint+Moves+SC	16384	4	8	0.455
Hint+Moves+SC	32768	4	9	0.911

Naturally, we consider combining the Second Chance scheme with these deletion-based schemes, and find that performance improves substantially. The gap between one move on a deletion and multiple moves nearly disappears. With a small CAM, a load factor of $1/2$ with just four hash functions can be achieved, using at most one move per insert and deletion operation. While this still does not meet the performance of a full cuckoo hashing implementation with four choices, the improvement over moving at most one item only on insertion or deletion is strong.

Previous experience with schemes that move only on insertions has shown that for small stashes, the distribution of the stash size is approximately Poisson. Here we find a similar rough correspondence. The distribution for the stash size at the end of 2^{18} moves over the 10,000 trials is given below in Figure 1.

IV. RESTRICTED HASHING

An alternative approach, suggested in [7], avoids the need for additional storage for hints by making hash locations at levels beyond the first depend only partially on the item, and primarily on the bucket at the previous level. For example, one way this could be done would be to have the location of an item x at the first level be given by the hash value $h_1(x)$, at the second level by $h_2(h_1(x))$, and so on. A simple variation is to have the bucket at the ith level be given by $h(x) \bmod 2^{k-i+1}$. This approach simplifies moves when deletions occur; given an item to be deleted, one can easily find items

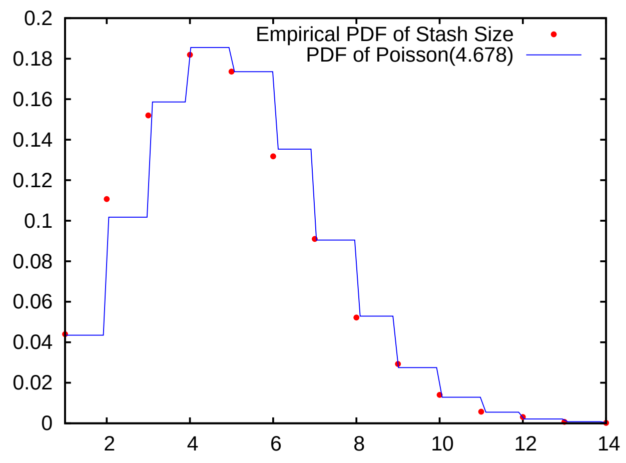

Fig. 1. A comparison of the distribution of the stash size and the Poisson distribution.

that can move to the empty space, based on searching a set of buckets dependent only on the value $h(x)$. A clear problem with this approach is that overloaded buckets simply pass items down level by level; if the load of some bucket at the first level is larger than the number of levels, then there will be a failure. In short, by not randomizing the hash per item at each level, one greatly reduces the spread of items among buckets after the first level.

While it is possible to use more sophisticated schemes, we here present a *negative* result. We consider a very powerful scheme that uses this general approach to avoid the need for hints, and show that it can be numerically analyzed. With this analysis approach, we show that even this scheme has comparatively poor behavior compared to alternative schemes using hints, or even the Second Chance scheme of [3] that only moves items on insertions. We therefore suggest that these schemes will likely prove less effective in almost all contexts.

A. A General Setup

We first discuss how to increase the spread by using a more sophisticated approach, describing the setup in full generality. Our suggested structure depends on a parameter ℓ. (Generally, ℓ will be small; it may help to think of $\ell = 2$ in what follows.) A first hash function H maps items in the universe to the range $[0, 2^k)$, and a second hash function G maps items to the range $[0, \ell^{v-1})$, where v is the number of levels. Thinking of $G(x)$ as an ℓ-ary vector of length $v-1$, let $g_i(x)$ be the ith item of the vector $G(x)$. Also, there are hash functions $h_{i,j}$ mapping items from $[0, 2^{k-i+1})$ to $[0, 2^{k-i})$, for $i \in [1, v-1]$ and $j \in [0, \ell)$.

We describe how these hash functions are used for insertion and deletion. For an input x, its bucket in the first table is given by $H(x)$. If that bucket is free, x is placed there. If there is a collision, however, then x must be placed at a subsequent level. The item will have one bucket at each level, given by

$$h_{1,g_1(x)}(H(x)), h_{2,g_2(x)}(h_{1,g_1(x)}(H(x))), \ldots$$

Alternatively, if $f(x)$ is the possible bucket of x at level i, then $h_{i,g_i(x)}(f(x))$ is its possible bucket at the next level.

More descriptively, the bucket for x at the first level is given by $H(x)$. In subsequent levels, if there are collisions at a bucket, any items that hash to these buckets have ℓ possible buckets at the next level that they can be hashed to, given by the functions $h_{i,j}$, which map buckets at level i to ℓ buckets at level j. Which specific bucket is used for an item x at each level is determined by the hash $G(x)$ (This is referred to as a *decider* function in [7].) Intuitively, the hash functions $h_{i,j}$ are used to spread collisions at one level to multiple buckets at subsequent levels.

This approach increases the spread of items across the table while maintaining the ability to move items up in the table in response to deletions. Specifically, when an item is deleted, there are only ℓ possible buckets at the next level to examine to see if an item can be moved to the open bucket, and at most ℓ^{v-1} possible buckets in total to consider through all levels. (It is possible that an item exists in the table that can be moved to the open bucket, but that it is not at the next level; it might have been placed at a later level because of further collisions.)

In fact, intuitively, we don't necessarily want the $h_{i,j}$ to be "random" hash functions; in such a case, certain buckets at each level could essentially go unused, as no bucket from the previous level would hash to them, and other buckets could receive items from far more than the average number of buckets at the previous level. Instead, we suggest using shifts; although we do not show it here, our analysis shows that shifts perform more effectively. More descriptively, when level sizes are a power of two and each bucket has two choices, we can think of each level as being split into a left half and right half. For each half, each bucket is given two possible buckets on the next level, one on the left and one on the right, on the next level. Specifically, we use $h_{i,j}(y) = y + s_{ij0} \bmod 2^{k-i-1}$ if $y < 2^{k-i}$, and $h_{i,j}(y) = (y + s_{ij1} \bmod 2^{k-i-1}) + 2^{k-i-1}$ if $y \geq 2^{k-i}$, for distinct

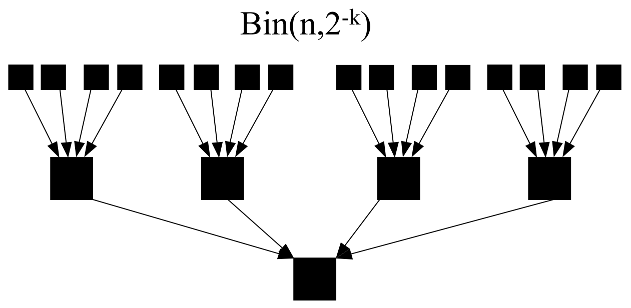

Fig. 2. An example with three levels. At the top level, the number of items in each bucket is distributed according to a binomial random variable, assumed independent. We can then compute the distribution of items in each bucket at the next level, and so on to the last level.

shifts s_{ij0} and s_{ij1}. For appropriately chosen constants s_{ij0} and s_{ij1}, and assuming the table is sufficiently large, we can ensure that the items that hash to a bucket $H(x)$ have ℓ^i distinct possible buckets to hash to in level i. Another way of thinking about it is given in Figure 2; under appropriate conditions regarding the size of the hash table and the choice of shift offsets, branching backwards from a bucket at the last level, the buckets that potentially pass items to this bucket form a tree. This fact avoids dependencies that would otherwise complicate analysis.

B. An Optimistic Analysis

We now suggest how to analyze this approach, utilizing techniques from [3], [4]. For convenience, we first consider the case of insertions only. If we consider the first level, the distribution of items in a bucket is $\text{Bin}(n,2^{-k})$. If buckets can hold c items, then the number of items passed on to the next level is distributed as $(\text{Bin}(n,2^{-k})-c)^+$, where $(x)^+ = \max(x,0)$ is the standard notation. We take $c = 1$ henceforth although this analysis approach is more general. If the remaining elements were split randomly among ℓ buckets at the next level, each would obtain a number of items distributed as $\text{Bin}((\text{Bin}(n,2^{-k})-1)^+,1/\ell)$, and the distribution of the number of items in a bin at the second level would be the sum of 2ℓ random variables with this distribution. From this, we we can calculate the distribution of the number of items in a bucket at the second level, and so on proceeding recursively, until we obtain the probability a bucket at the last level overflows. Using this approach, we can obtain quite accurate predictions for the number of overflowing bins.

We note that there are some simplifications being made in this analysis, and hence it is only approximate. First, while the distribution of items in any single bucket at the first level is indeed given by $\text{Bin}(n,2^{-k})$, the joint distribution among several buckets does not exactly correspond to independent binomial random variables. This difference is negligible asymptotically, and we ignore it henceforth. Similarly, when we reach the last level, we obtain a distribution for the number of items that land in each bucket, but we do not obtain a joint distribution, which would allow a direct calculation of the number of items overflowing into the stash. Again, treating the variables as independent appears to be a rough but suitable approximation. Alternatively, we can derive the expected overflow into the stash, and use the experimental fact that this distribution is approximately the sum of independent Bernoulli trials, and is therefore approximately Poisson or normal in the standard regimes. Finally, as mentioned previously, we are assuming that we have chosen shifts appropriately, so that each bucket receives items passed from 2ℓ buckets in the previous level, and there is no dependence. (See Figure 2.)

Interestingly, the above analysis can be made to hold even in the case of deletions. It is not clear that it would be natural under any deletion scheme for the item kept in a bucket to be chosen randomly from the items hashed to the bucket. However, one could imagine a somewhat impractical algorithm which kept this invariant after any insertion or deletion; as a newly inserted or deleted item only affects the load of a constant number of buckets, such a scheme might not even be completely impractical, as updates would take only constant time and a constant number of move operations (although these constants can be quite high, exponential in the number of levels). We call the scheme where, at each step, each bucket keeps a random item and passed down all others the Random scheme. Our analysis above allows us to compute steady-state quantities for the Random scheme. (The maximum over extended time periods would have to be considered via simulation, as there would be dependence between time steps.)

A more sophisticated scheme, which we call Greedy, is to keep not a random item for each bucket, but instead keep an item that balances as much as possible the items distributed to the next level. That is, if a bucket gets six items, with four mapped to bucket A at the next level and two to bucket B, it makes sense to store one destined to bucket A. (It might *not* be best in any particular instance, but statistically it is the right approach.) We can analyze this scheme numerically as well, using the same approach. Indeed, we can similarly

TABLE II

NUMERICAL RESULTS FOR SCHEMES WITH LIMITED HASH FUNCTIONS.

Scheme	Items (Level 0 Table Size)	Hash Functions	Average Stash (items)
Random	8192	6	0.618
Random	16384	6	1.236
Random	32768	6	2.470
Greedy	8192	6	0.0455
Greedy	16384	6	0.0908
Greedy	32768	6	0.1815

analyze any scheme where the item to be kept at the bucket at each level depends only on the items at the bucket at that level, and not on happenings at future lower levels. Our assumption is that the Greedy scheme, and indeed even the Random scheme, have performance significantly better than one could hope to expect from schemes used in practice. While this assumption is admittedly unproven, we do not expect a better online scheme.

The results from Table II give the expected size of the stash after using six levels. Even the optimistic Greedy version of this approach, which in theory allows multiple moves on any insert or delete operation and potentially requires examining a considerable number of buckets for an item to move, has a higher average stash size than using just the Second Chance scheme of [3] on insertions, as can be seen by comparing with Table I. While the optimistic scheme potentially performs better than schemes that only use moves on deletions, we believe it is clear that the hint-based approach combined with the Second Chance scheme or, when hints might be problematic, just using the Second Chance scheme, provide better performance.

V. CONCLUSIONS

Extending the direction taken in [3], we have considered multilevel hash tables that move items on either an insertion or a deletion. Moving items on a deletion is harder than on an insertion, since one needs a mechanism to find an appropriate item to move into the empty location. Also, such schemes do not appear generally amenable to standard analysis techniques, and for many of them, analysis remains open.

With these caveats, we have found experimentally that using hints to locate possible items to move on a deletion is a reasonable approach that can save space or reduce the number of hash functions used in such schemes. Schemes that allow even just one additional item to move on an insertion and deletion gain substantially over schemes with no moves, or previous schemes that move only on insertion.

An alternative approach based on using highly restricted hash functions, suggested by the work of [7], appears less effective. Here our analysis of a very optimistic scheme shows performance will be worse than even the easily implemented Second Chance scheme that moves items only on insertions.

As an open question, we note that cuckoo hashing schemes generally do not perform moves on deletions, only on insertions [10]. While moves on deletions would not appear capable of changing the asymptotic characteristics of the cost of insertion operations, it is interesting to consider whether moves on deletions could improve practical performance significantly.

REFERENCES

[1] A. Broder and A. Karlin. Multilevel Adaptive Hashing. In *Proceedings of the 1st ACM-SIAM Symposium on Discrete Algorithms* (SODA), pp. 43-53, 1990.

[2] B. Chazelle, J. Kilian, R. Rubinfeld, and A. Tal. The Bloomier Filter: An Efficient Data Structure for Static Support Lookup Tables. In *Proceedings of the Fifteenth Annual ACM-SIAM Symposium on Discrete Algorithms* (SODA), pp. 30-39, 2004.

[3] A. Kirsch and M. Mitzenmacher. The Power of One Move: Hashing Schemes for Hardware. In *Proceedings of the 27th IEEE International Conference on Computer Communications* (INFOCOM), 2008.

[4] A. Kirsch and M. Mitzenmacher. Simple Summaries for Hashing with Choices. *IEEE/ACM Transactions on Networking*, 16(1):218-231, 2008.

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

[6] A. Kirsch, M. Mitzenmacher, and U. Wieder. More Robust Hashing: Cuckoo Hashing with a Stash. To appear in *Proceedings of the 16th Annual European Symposium on Algorithms*, 2008.

[7] S. Kumar, J. Turner, and P. Crowley. Peacock Hash: Fast and Updatable Hashing for High Performance Packet Processing Algorithms. In *Proceedings of the 27th IEEE International Conference on Computer Communications* (INFOCOM), 2008.

[8] M. Mitzenmacher, A. Richa, and R. Sitaraman. The Power of Two Choices: A Survey of Techniques and Results. In *Handbook of Randomized Computing*, edited by P. Pardalos, S. Rajasekaran, J. Reif, and J. Rolim. Kluwer Academic Publishers, Norwell, MA, 2001, pp. 255-312.

[9] M. Mitzenmacher and S. Vadhan. Why Simple Hash Functions Work: Exploiting the Entropy in a Data Stream. In *Proceedings of the Nineteenth Annual ACM-SIAM Symposium on Discrete Algorithms* (SODA), pp. 746-755, 2008.

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

[11] B. Vöcking. How Asymmetry Helps Load Balancing. *Journal of the ACM*, 50(4):568-589, 2003.