

# Using a Queue to De-amortize Cuckoo Hashing in Hardware

Adam Kirsch and Michael Mitzenmacher  
Harvard School of Engineering and Applied Sciences  
Cambridge, MA 02138  
Email: {kirsch,michaelm}@eecs.harvard.edu

**Abstract**—Cuckoo hashing combines multiple-choice hashing with the power to move elements, providing hash tables with very high space utilization and low probability of overflow. However, inserting a new object into such a hash table can take substantial time, requiring many elements to be moved. While these events are rare and the amortized performance of these data structures is excellent, this shortcoming is unacceptable in many applications, particularly those involving hardware router implementations. We address this difficulty, focusing on the potential of content-addressable memories and queueing techniques to provide a de-amortization of cuckoo hashing suitable for hardware, and in particular for high-performance routers.

## I. INTRODUCTION

Cuckoo hashing is a novel hashing approach that achieves very high space utilization [2]–[4], [8]. It extends previous work on multiple-choice hashing [1], [7], which allows elements to be stored in one of a small number  $d$  of locations, by also allowing elements to move between their hash locations to prevent bucket overflow. The appropriate choice for  $d$  is determined by the application, although typically  $d = 4$  is a suitable choice. In most constructions these  $d$  hash locations can be easily accessed in parallel, so that lookups require only one parallel memory access.

Standard cuckoo hashing variants require only a constant number of move operations per element insertion on average, where the constant depends on the load of the hash table. However, there is also a non-negligible probability that, during a sequence of  $n$  insertion operations into an appropriately sized table, some insertion requires moving at least  $\Omega(\log n)$  elements. This amortized performance is generally suitable for software applications. Here, however, we consider the context of router hardware, where hash tables are used for a variety of operations, including IP lookups and network measurement and monitoring tasks. In this setting, the router must keep up with line speeds and memory accesses are at a premium. Amortized performance guarantees alone are generally not sufficient. The time per operation must have a fixed maximum, or the probability that a particular operation requires a large number of memory accesses must be negligible. Indeed, in our initial consideration of using moves to improve the space utilization of hash tables in hardware implementations [6], we explored the gains obtainable by allowing *at most one move* per insertion.

While our previous work shows that the value of even a single move can be substantial, it also shows that when ele-

ments can be inserted and deleted over time in the hash table, more moves are needed to maintain high space utilization. This leads us to ask whether cuckoo hashing can be implemented effectively in hardware. Our exploration leads to the following contributions:

- We describe how to significantly *de-amortize* cuckoo hashing naturally in hardware, limiting the number of moves per insertion by using a small content-addressable memory (CAM) as a queue for elements being moved.
- We consider the impact of different queueing policies on performance.
- We explore the tradeoff between the utilization of space in the hash table and the number of moves allowed per insertion.

Our work suggests that cuckoo hashing may prove implementable in router hardware in the near future, with the potential to significantly enhance hash table performance.

## II. CUCKOO HASHING: BACKGROUND AND OUR APPROACH

In the original description of cuckoo hashing by Pagh and Rodler [8], there are two sub-tables,  $T_1$  and  $T_2$ , each of size  $r$ , and two hash functions  $h_1$  and  $h_2$  mapping the universe of elements to  $[0, r - 1]$ . Every element  $x$  stored in the hash table is in location  $T_1[h_1(x)]$  or  $T_2[h_2(x)]$  (but not both). Lookups are therefore trivial; we just check the two possible locations. To insert an element  $x$ , we check if  $T_1[h_1(x)]$  is empty. If it is, we are done. If not, we replace the element  $y$  in  $T_1[h_1(x)]$  with  $x$ . We then check if  $T_2[h_2(y)]$  is empty. If it is, we are done. If not, we replace the element  $z$  in  $T_2[h_2(y)]$  with  $y$ . We then try to place  $z$  in  $T_1[h_1(z)]$ , and so on, until we find an empty location. (If an empty location is not found within a certain number of tries, the suggested solution is to rehash all of the elements in the table.) For any constant  $\epsilon > 0$ , when  $(1 - \epsilon)r$  elements are inserted (corresponding to any load factor less than  $1/2$ , since there are  $2r$  buckets in the two hash tables), the total number of required moves is  $O(r)$  with high probability, giving  $O(1)$  amortized time per insertion with high probability. However, there is a significant probability that at least one of these insertion operations requires  $\Omega(\log r)$  moves. Thus, in practice, the insertions require constant time on average, but the worst-case insertion time can be  $\Omega(\log r)$ . This worst-case insertion time is problematic for some applications, but particularly for high speed routers, where insertion operations

come in quickly and cannot be easily buffered. In these cases, standard cuckoo hashing is likely unsuitable.

Before proceeding with our modification, we note that the ideas behind cuckoo hashing have been successfully applied to settings where there are more than two sub-tables and more than one element can be stored in a hash bucket [3], [4]. For these settings, there are still many open questions in the analysis. In practice it appears that one can achieve very high memory utilizations — well over 90% when using multiple choices and/or multiple elements per location — while still using only a constant number of moves per insert on average. However, the number of moves can still be large with significant probability.

In this paper, we always assume that each hash bucket can store at most one element. However, we consider a natural and practical generalization of cuckoo hashing to  $d > 2$  sub-tables, proposed by Fotakis et. al [4] and analyzed through simulations. We consider  $d = 4$  later in the paper, as this choice appears practical and gives a substantial space savings over  $d = 2$ . In this scheme, when attempting to insert an element  $x$ , we check if any of its hash locations  $T_1[h_1(x)], \dots, T_d[h_d(x)]$  are unoccupied, and place it in the leftmost unoccupied bucket if that is the case. Otherwise, we choose a random  $I \in \{1, \dots, d\}$  and evict the element  $y$  in  $T_I[h_I(x)]$ , replacing  $y$  with  $x$ . We then check if any of  $y$ 's hash locations are unoccupied, placing it in the leftmost unoccupied bucket if this is the case. Otherwise, we choose a random  $J \in \{1, \dots, d\} - \{I\}$  and evict the element  $z$  in  $T_J[h_J(y)]$ , replacing it with  $y$ . We repeat this procedure until an eviction is no longer necessary. As we shall soon see, simulations suggest that, like standard cuckoo hashing, this scheme is not well-suited to applications where low amortized insertion times do not compensate for large worst-case insertion times.

To mitigate this problem, we think of each cuckoo hashing insertion operation as consisting of a (random) number of *sub-operations*. The first sub-operation corresponds to the *initial attempt* to place an element in one of its hash locations, evicting the element in a randomly chosen hash location if all of those positions are occupied. Each subsequent sub-operation corresponds to an attempt to place an element previously evicted from some sub-table  $i$  in any of its other hash locations, evicting an element from a randomly chosen hash location other than the one in the sub-table  $i$  if all of those positions are occupied.

We now introduce an *insertion queue*, implemented by a CAM, which stores sub-operations to be processed. A sub-operation can be represented simply as the element  $x$  that the sub-operation is attempting to insert, a number  $i \in \{0, \dots, d\}$  representing, if  $i > 0$ , that the element  $x$  was previously evicted from sub-table  $i$ , or if  $i = 0$ , that the sub-operation is an initial attempt to insert  $x$ . To process a sub-operation, we simply remove it from the queue and execute it. If the sub-operation gives rise to a new sub-operation, we insert that into the queue. The queue is equipped with some policy for determining the order in which sub-operations should be processed.

We emphasize that regardless of this reordering of the sub-operations, lookups, insertions, and deletions can still be efficiently performed. Indeed, a lookup for an element can be performed by checking its  $d$  hash locations and the CAM in parallel. An insertion can be performed by inserting its first sub-operation into the queue. Finally, a deletion can be performed by first finding whether the element to be deleted is in one of its hash locations or the queue. If the element is in one its hash locations, the bucket can simply be marked as unoccupied, to be overwritten by a future insertion sub-operation. If the element is in the queue, then the corresponding sub-operation is removed from the queue. (It is easy to see that an element can appear in at most one sub-operation in the queue at any point in time.) This ability to reorder sub-operations without compromising the integrity of the hash table is the key feature of cuckoo hashing that we exploit in improving its performance.

Of course, for our queueing approach to be useful, the queue must actually fit into a CAM (at least under normal operating conditions). Because large CAMs are expensive, the size of the queue appears to be the primary issue in choosing the queueing policy to use. With this in mind, we now begin to explore the relationship between the choice of queueing policy, the pattern of insertions and deletions, and the resulting size of the queue over time. We emphasize that all of our results are preliminary, but they suggest that our technique of using an insertion queue can be a practical and very effective way to de-amortize the performance of cuckoo hashing.

### III. THE QUEUEING POLICIES

In the standard cuckoo hashing schemes discussed in Section II, we insert elements one-by-one, processing every one of an insertion operation's sub-operations before proceeding to the next insertion operation. Thus, this corresponds to the queueing discipline where, when inserting a new element, we place the corresponding initial sub-operation onto the back of the queue, and when a processed sub-operation yields a new sub-operation, we place the new sub-operation on the front of the queue. Since this policy does not exploit our ability to reorder sub-operations in the queue, we call it the *naive policy*.

The naive policy can give rise to very large queue sizes if it becomes "stuck" attempting to process an unusually large number of sub-operations resulting from a troublesome insertion operation, allowing newly inserted elements to queue up in the meantime. To avoid this, we consider more complex policies. Let us define the *age* of a sub-operation  $o$  in the following way. If  $o$  is an initial attempt to place an element, it has age 0. Otherwise, the sub-operation arose from another sub-operation with some age  $m$ , and so we define the age of  $o$  to be  $m + 1$ . A first obvious point is that elements of age 0 should have priority in the queue, as they have  $d$  choices for placement, while older elements have essentially only  $d - 1$  choices for placement, assuming the element that evicted it has not been deleted. Hence, insertions of new elements are

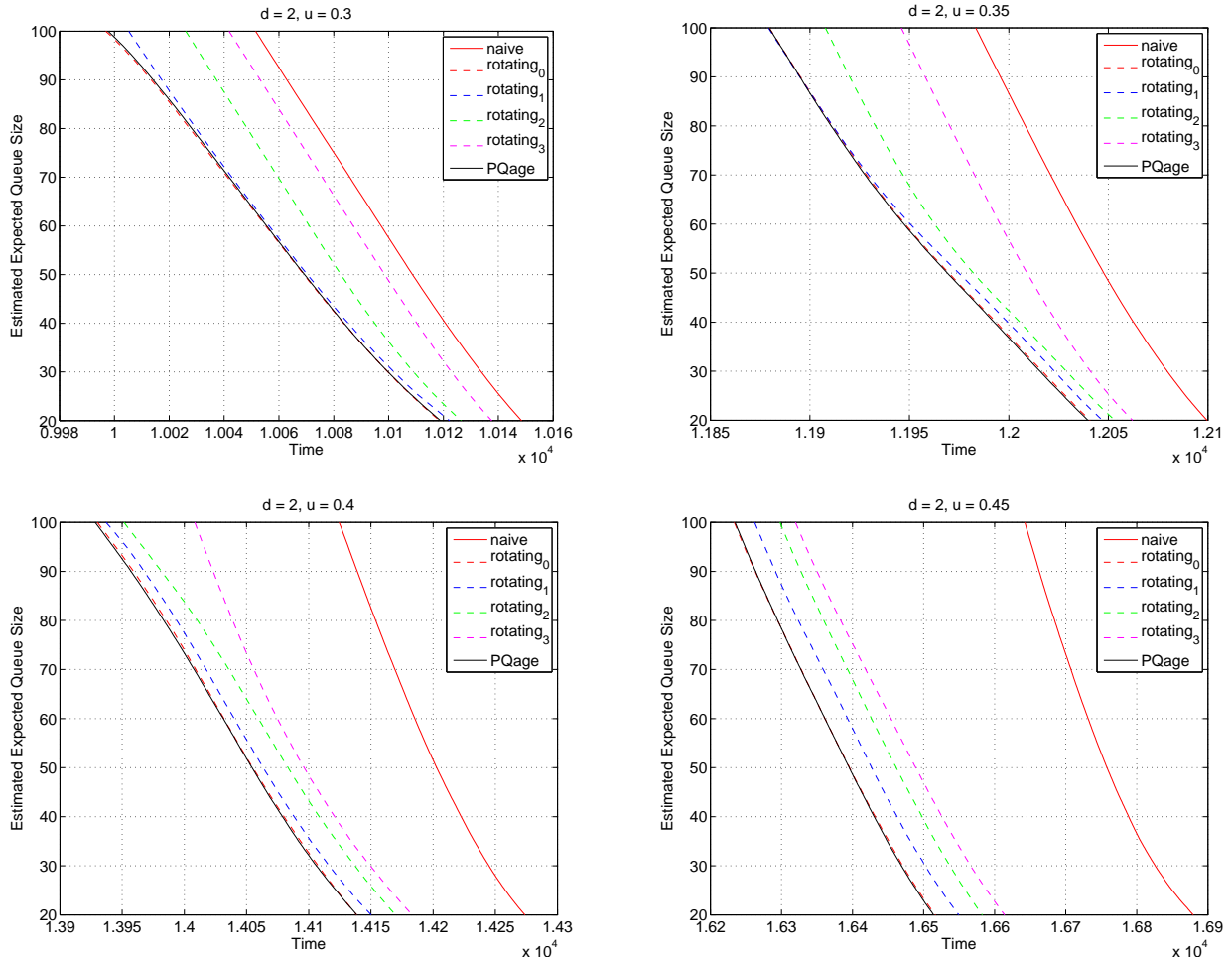


Fig. 1. Results of the simple experiment for  $d = 2$  and various values of  $u$ .

much more likely to succeed than attempts to place an evicted element.

It is also intuitively reasonable to believe that, more generally, the probability that processing  $o$  will not result in the generation of a new insertion operation is roughly decreasing in the age of  $o$ . That is, the longer an element has been in the queue, the less likely it is to succeed in placement. This is certainly true in the rare case where the inability to place an element arises because there is a cycle, causing the same sequence of elements to be evicted repeatedly (until a deletion occurs). With this intuition in mind, we define the policy PQage, which treats the queue as a priority queue where the priority of a sub-operation is its age, and lower ages correspond to higher priorities.

Of course, while the PQage policy is nice in theory, it is likely to be impractical to implement; creating a priority queue within the CAM may be too expensive in terms of operation time or hardware resources. The most natural way to deal with this is to attempt to mimic the PQage policy for queues where some sub-operation has a small age. Since it is intuitively very unusual for *all* of the sub-operations in the queue to have large age, we expect such a policy to perform well. With this in

mind, we define the policy  $\text{rotating}_i$  for a fixed  $i \in \mathbb{Z}_{\geq 0}$  as follows. During an insertion operation, the corresponding initial insertion sub-operation is placed on the front of the queue. When a new sub-operation  $o$  is generated during the processing of some other sub-operation, we place  $o$  on the front of the queue if its age is at most  $i$ , and otherwise we place  $o$  on the back of the queue. Thus, we effectively give precedence to sub-operations with age at most  $i$ , while rotating through the other sub-operations to avoid getting stuck processing the occasional particularly troublesome insertion operation. Furthermore, unlike the PQage policy, the  $\text{rotating}_i$  policy is easy to implement. If for a sub-operation  $o$  with age  $a$ , we store the value  $\max(a, i + 1)$  along with  $o$  in the CAM, then the  $\text{rotating}_i$  requires very little computational and storage overhead, particularly for small  $i$ .

Finally, we consider an even simpler policy. Recall that, according to our intuition, the main shortcoming of the naive policy is that newly inserted elements, which are very likely to be placed successfully in one sub-operation, can get stuck behind much older sub-operations. A very simple way to deal with this problem is to simply modify the naive policy so that newly inserted elements always go on the front of the queue,

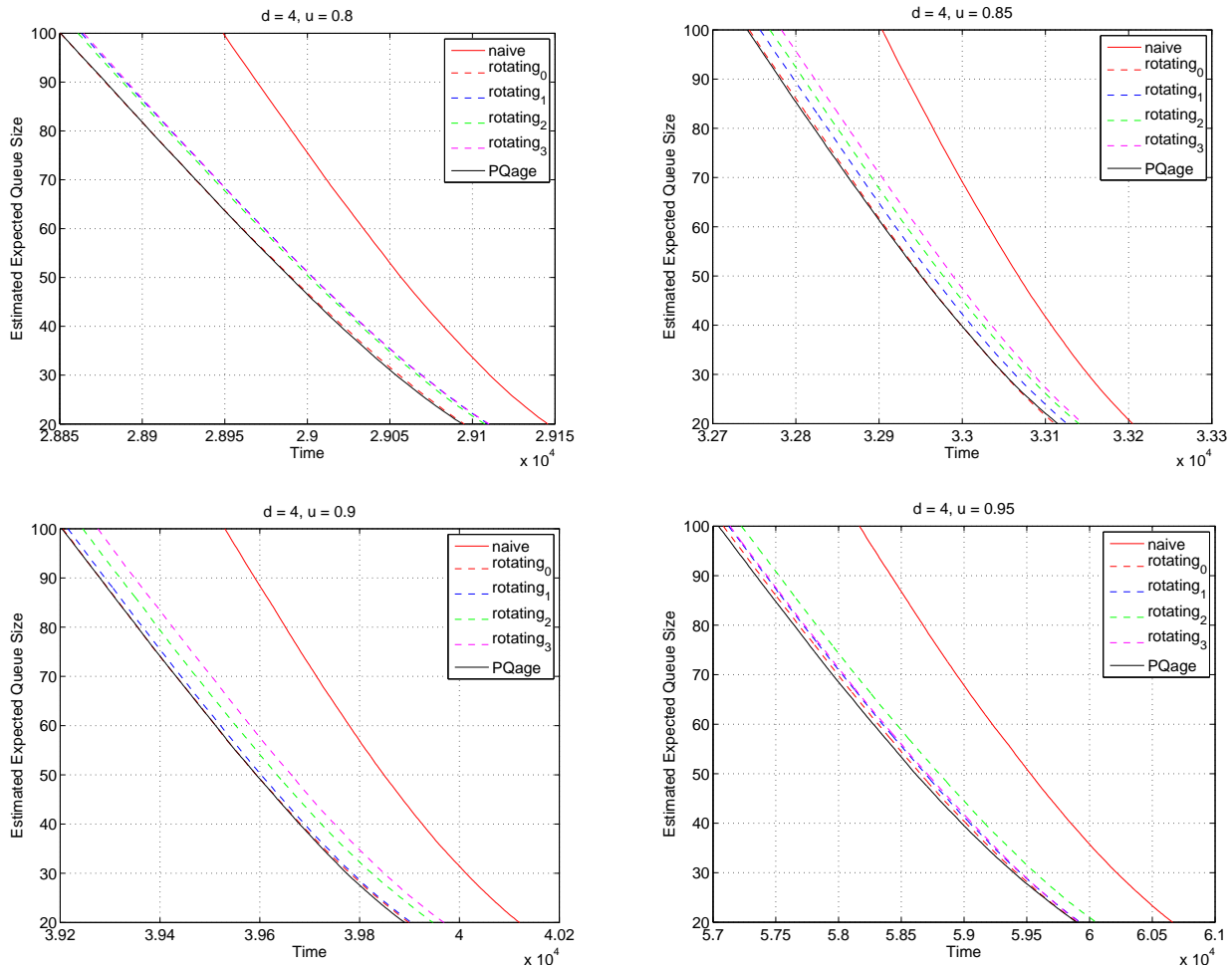


Fig. 2. Results of the simple experiment for  $d = 4$  and various values of  $u$ .

instead of the back. We name this policy *naive\**. Of course, if we are in a situation where the queue is highly backlogged and there are no new insertion operations, then the *naive* and *naive\** policies are exactly the same, and we expect the other policies to perform better.

#### IV. A SIMPLE EXPERIMENT

We now discuss a simple experiment with the queueing policies introduced in Section III. We emphasize that this experiment is not designed to explicitly model any particular phenomenon that is likely to occur in practice. Rather, the point of the experiment is to help verify and clarify the intuition from Section III in what seems to be the simplest possible way.

With this in mind, we recall that the key intuition used in Section III is that (heuristically) the probability that executing a particular sub-operation  $o$  in the queue does not result in a new sub-operation is roughly decreasing in the age of  $o$ . Thus, we expect that queueing policies that favor sub-operations with lower ages tend to reduce the size of the queue faster than the standard *naive* policy. Furthermore, the standard theoretical results behind cuckoo hashing suggest that most

insertion operations require few sub-operations, so it should be sufficient to simply ensure that sub-operations with very small ages take precedence over those with larger ages, and therefore we expect that, for some small  $i$ , the policy *rotating<sub>i</sub>* should perform about as well as the much more complicated *PQage* policy.

In this section, we test these two claims with the following very simple experiment. We consider a table with 32768 buckets, divided into  $d$  equal sub-tables, for  $d = 2$  and  $d = 4$ . We fix a *utilization* value  $u \in (0, 1)$  and do the following for each of the queueing policies discussed in Section III. First, we load the queue with  $\lfloor 32768u \rfloor$  insertion operations and then let the system run, processing one sub-operation per time step, for 65000 time steps, recording the queue size after every step. We average the results over 1000 trials to get a curve corresponding to the queueing policy. We then draw the curves corresponding to all of the different queueing policies (for a fixed  $u$ ) on the same plot, and then zoom in to the portion where all of the curves lie between 20 and 100 on the  $y$ -axis.

The results for  $d = 2$  are given in Figure 1, and the results for  $d = 4$  are given in Figure 2. (Note that since there are no

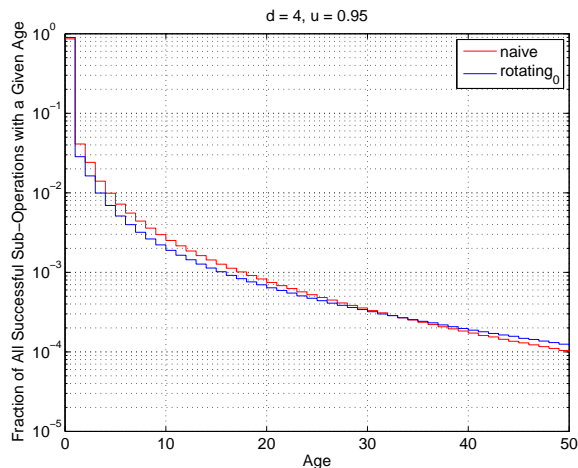
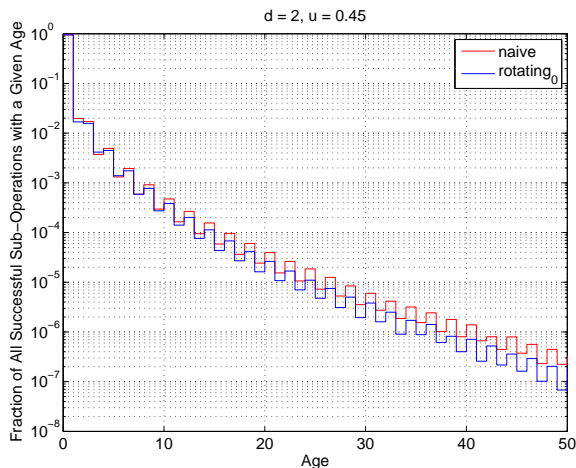


Fig. 3. The age distribution, or the number of moves required to place an element, for the naive and `rotating0` queuing policies.

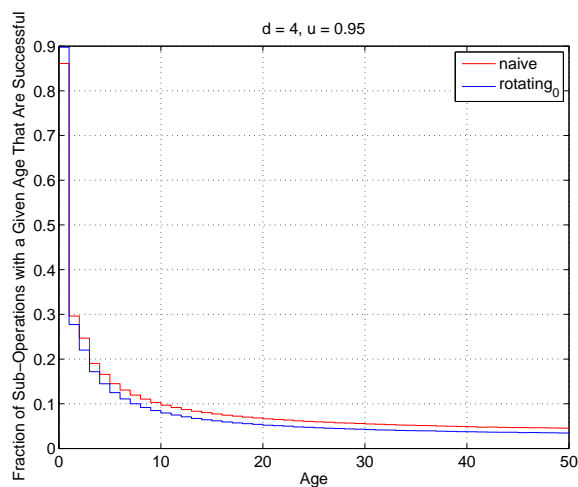
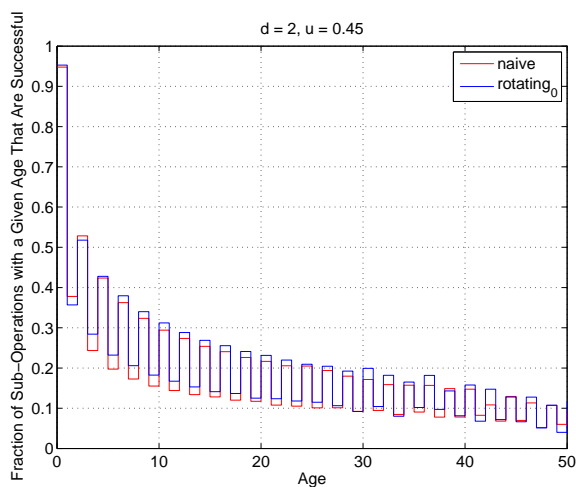


Fig. 4. The fraction of successful placements for each age. Overall, the larger the age of a sub-operation, the less likely it is to succeed.

new insertion operations during the course of this experiment, the naive and naive\* policies are the same, and so we only display results for the naive policy.) The specific values for  $u$  are chosen because prior work [4] observes that cuckoo hashing breaks down at  $u = 0.5$  for  $d = 2$ , and  $u = 0.97$  for  $d = 4$ . The pattern of the curves indicates that for certain large queue sizes, the PQage policy gives the most effective placement of elements in the table, the naive policy gives the least effective placement, and the difference is significant for all utilizations. Also, the performance of the `rotating0` policy is essentially the same as for the PQage policy.

These results match very nicely with our intuition from Section III, as they suggest that the naive policy can be significantly improved by prioritizing sub-operations according to their ages. Furthermore, it appears that the maximum benefit of such a scheme can be effectively realized by simply giving priority to the initial attempts to insert elements, while rotating through the other sub-operations to lessen the impact of the

occasional particularly troublesome insertion operation on the average queue size.

Other results from these experiments are also enlightening. Here we focus on the naive and `rotating0` schemes near the threshold utilization values where cuckoo hashing breaks down, and increase the number of trials from 1000 to 10000 to decrease sampling error. Figure 3 shows the distribution of the age when a successful placement occurs. As we discussed in the introduction, some elements require a significant number of move operations before placement occurs. In fact, most elements are placed during the initial attempt to insert them, backing our intuition that it is important to place new insertions at the front of the queue. Note that there is some skew when  $d = 2$  because of our initial preference to place elements in the first hash table; more elements are placed in the first sub-table, causing bounces between odd and even ages in the age distribution. We have not attempted to exploit this behavior in our queuing policies, although one could conceivably do

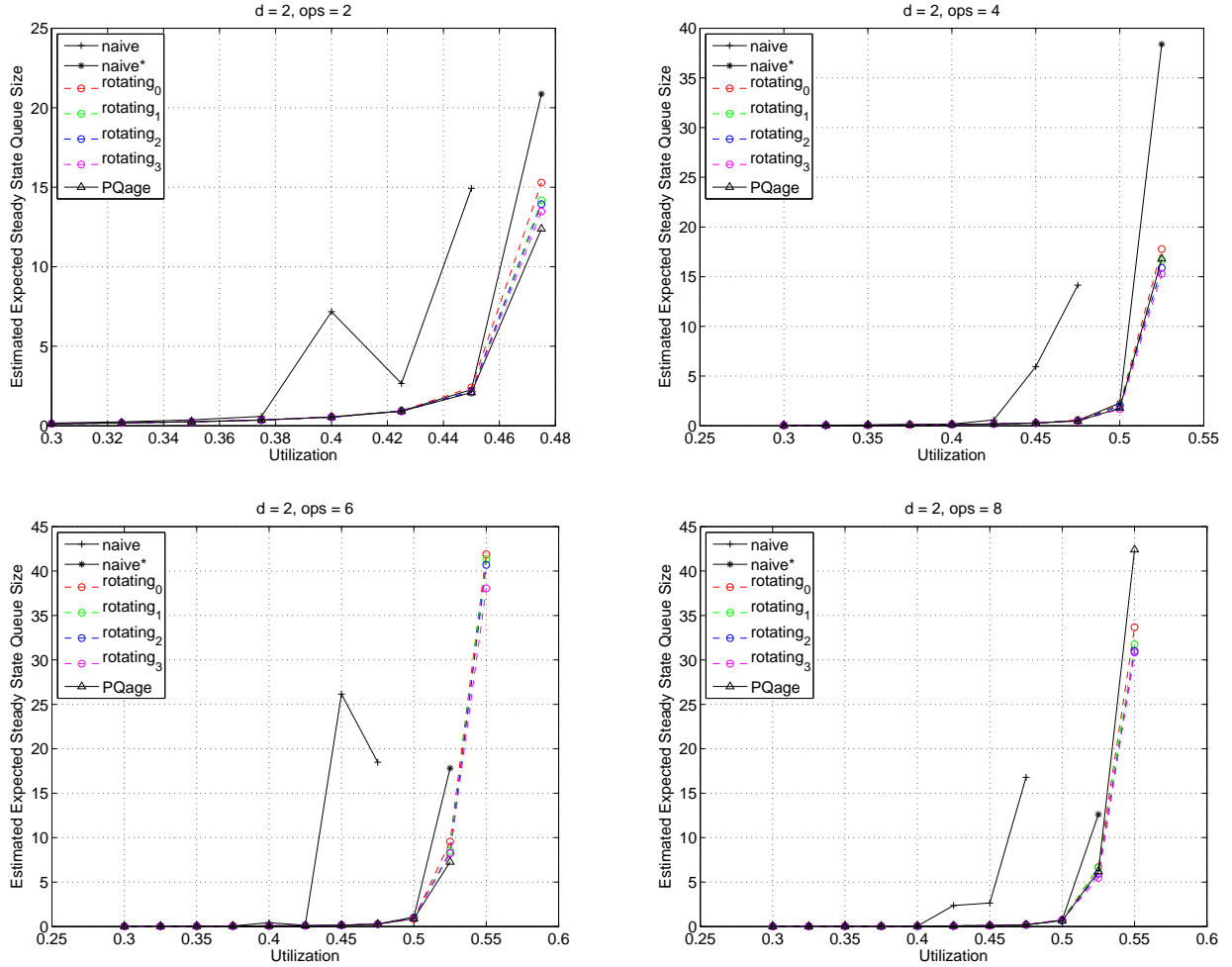


Fig. 5. Results from the more realistic experiment with  $d = 2$ .

so.

Figure 4 shows the distribution of the fraction of successful placements for each age. As we suggested, the main feature is that as the age of sub-operation increases, its success probability decreases. (Again, there are odd-even artifacts when  $d = 2$ , but still the trend is clear.) In particular, it makes sense that a good choice for  $i$  in the `rotatingi` policy is  $i = 0$ , leading to a simple but effective queuing policy.

## V. A MORE REALISTIC EXPERIMENT

While the results in Section IV are certainly significant, it is clearly necessary to perform a more realistic set of experiments. In particular, it is important to consider scenarios that intermix insertion and deletion operations, so that we can begin to see the relationship between the pattern of hash table operations and the choice of queuing policy. As we shall see, the core of our intuition from Sections III and IV continues to apply in these settings.

In this section, we consider the following experiment. We fix a value for  $d \in \{2, 4\}$  and consider a table of size 32768 divided into  $d$  equal-size sub-tables. (We also experimented with several non-uniform allocations of space between the

sub-tables [5], [6], but the results in this case were always significantly worse.) We also fix a value  $ops \in \{2, 4, 6, 8\}$ , a queuing policy from Section III, and a utilization parameter  $u$ . For  $d = 2$ , we take  $u \in \{0.3, 0.325, 0.35, \dots, 0.55\}$ , and for  $d = 4$ , we take  $u \in \{0.7, 0.725, 0.75, \dots, 0.95\}$ . Once again, these values of  $u$  are chosen because prior work [4] observes that standard cuckoo hashing breaks down at  $u = 0.5$  for  $d = 2$  and  $u = 0.97$  for  $d = 4$ .

We then repeat the following procedure 100 times. We insert  $\lfloor 32768u \rfloor$  elements into the data structure, processing  $ops$  sub-operations between insertions. Then we perform 30000 alternations of deleting a random element in the data structure, then inserting a new element, then processing  $ops$  sub-operations. We record the queue size after every operation, and we treat the last 10000 alternations of deletions, insertions, and queuing operations as reflecting the steady state of the queue size. Here, the time scale is queuing operations; each queuing operation is presumed to take one unit of time, whereas deletions and insertions are presumed to be instantaneous. We average the results over all trials to obtain an estimate of the expected value of the steady state size of

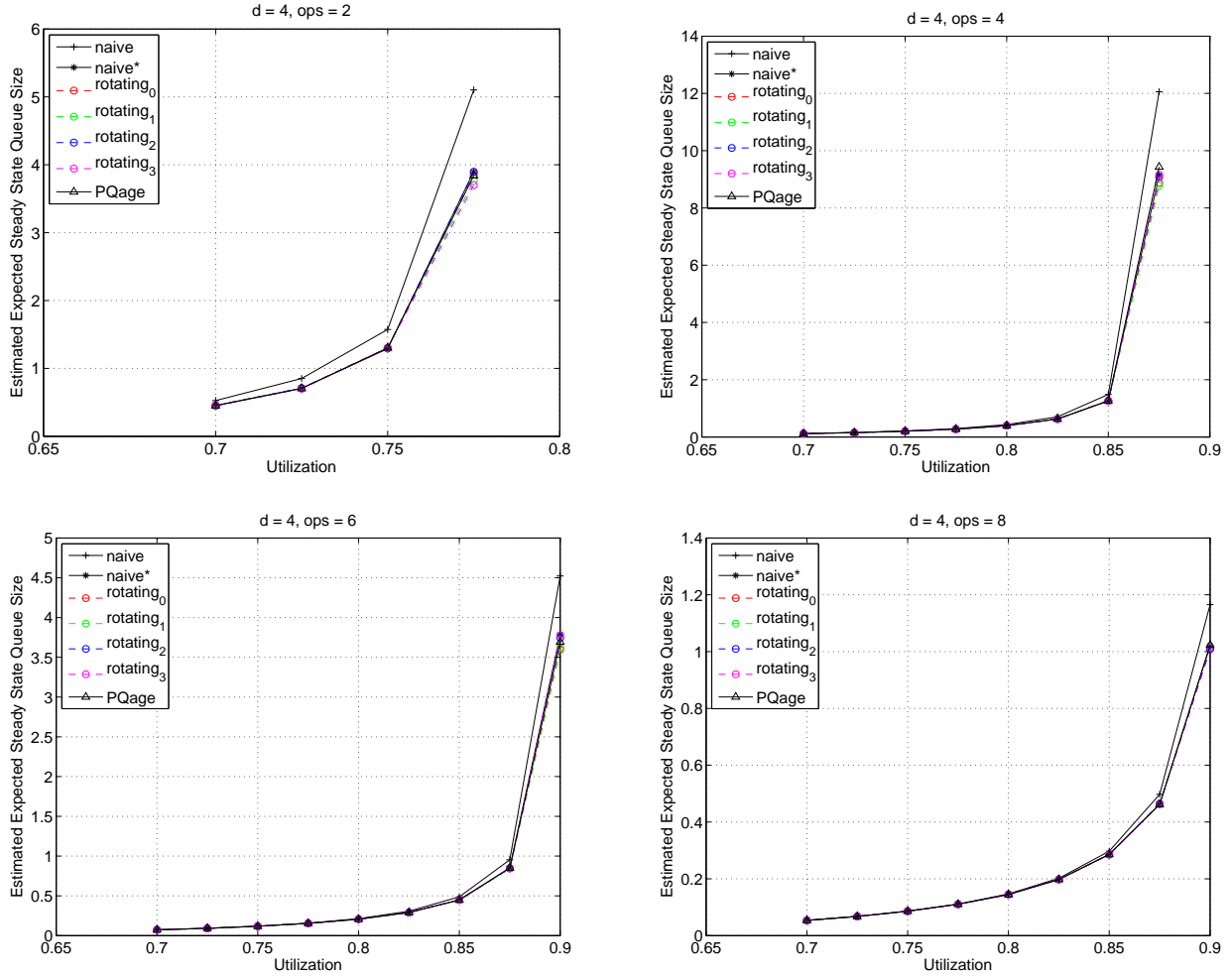


Fig. 6. Results from the more realistic experiment with  $d = 4$ .

the queue.

As an aside, we note that in practice it may be difficult to delete an element from the queue instantaneously. In this case it might be preferable to keep a list of elements in the queue marked as deleted, and not actually remove a deleted element until its sub-operation is served by the queue. This would certainly enlarge the steady state queue sizes, but would not qualitatively affect our results.

This experiment focuses on the average size of the queue when the data structure is designed for a specific utilization of the hash table and average relative frequency of insertion/deletion and queueing operations, and then run at its capacity for a long period of time. In a more heuristic sense, this experiment corresponds to the steady state of an asymptotic regime where the size of the hash table tends towards infinity, the utilization value  $u$  is a fixed constant, insertions occur according to a Poisson process with rate  $1/ops$ , and the lifetime of an element in the table has an exponential distribution with mean  $ops/u$ . The latter setting is significant in that it seems to be the beginning of a realistic model for how the system might behave in practice. Indeed,

the assumption that arrivals occur according to a Poisson process is fairly standard. However, the restriction that the lifetime of an element in the table is exponential is likely to be extremely unrealistic. Nevertheless, the experiments in this section certainly seem like a good start towards understanding how the systems we consider might behave in the real world.

We plot our results for  $d = 2$  and  $d = 4$  in Figures 5 and 6, respectively. To make the plots readable, we omit all data points where the estimated expected steady state size of the queue exceeds 50, which we consider to be a practical value for the applications of interest in this paper. For  $d = 2$ , the results are striking. The naive\* policy performs significantly better than the naive policy, the rotating policies give an additional improvement, and are comparable with PQage. For  $d = 4$ , the differences between the queueing policies are less significant, but there is still a gain in choosing even the naive\* policy over naive. Again, there is an advantage to giving priority to the initial sub-operation of an insertion, but otherwise differences in the steady state are minor.

Of course, for implementation one is interested not only in the steady state queue size, but in the maximum queue size

seen. In particular, a good design must ensure the probability of queue overflow is extremely low. Because we have no analytic bounds on the queue behavior, it is difficult to bound this type of event. Even under the idealized assumptions of our experiments, large-scale simulations would be needed to accurately determine the rarity of overflow events. In practice, the probability would depend on many factors, including the amount of time the process runs, the distribution of how long elements live in the hash table, and the burstiness of insertion and deletion operations. In our experiments, we have found that when the average steady state queue size is in the range  $[1, 5]$ , the largest gap between the average steady state queue size and the maximum queue size seen over all simulations (for each setting of  $d$ , ops, queueing policy, and utilization) was a factor between 10 and 30. The one exception was the `naive` policy, which naturally is significantly worse; the `rotating0` policy performed very well, with a maximum gap factor between 15 and 16. While this issue remains a subject for future analysis and/or experimentation, we maintain that these initial results strongly suggest that cuckoo hashing should be suitable for practical implementations using a reasonably sized CAM.

## VI. CONCLUSION AND FURTHER WORK

We believe our results demonstrate that cuckoo hashing for router hardware, and our CAM-based queueing approach in particular, have merit and warrant further investigation. However, we leave many open questions. On the theoretical side, there remain several open questions regarding the analysis of cuckoo hashing performance, particularly in the setting with  $d > 2$  choices. A more complete understanding could certainly impact practical designs.

On the experimental side, while we have only looked at expected steady state queue sizes in this paper, it is certainly necessary to understand the frequency of very large queue sizes, as these rare events correspond to overflow in the queue. In this same vein, we should learn more about how different queueing policies behave when the occasional disturbance, such as a large influx of insertions, occurs. It seems likely that policies with similar steady state queue size may perform quite differently under these circumstances, as suggested by our simple experiment in Section IV. Another natural possible approach if bursty insertion behavior is expected would allow a varying number of queue operations per insertion depending on the length of the queue, in order to prevent queue overflow.

Additionally, it is very important to understand how the behavior of the system is affected by the distribution of the lifetime that an element spends in the data structure. In our experiments, we always assume that when a deletion occurs, all elements in the structure are deleted with equal probability. Of course, this is unlikely to be the case in practice, and it is important to understand the effect of this assumption on our results. Indeed, if the amount of time that an element spends in the table is not exponential, then it may be worthwhile to take the age of an element (as opposed to just the age of a sub-operation) into account when designing the queueing policy.

The full ramifications of these and related issues would have to be considered more carefully in a complete hardware design. Overall, however, our exploration has shown that cuckoo hashing has great potential for hardware implementations. In particular, we have demonstrated a simple and practical queueing technique that can allow de-amortizing standard cuckoo hashing. Our technique appears to maintain high utilizations, while keeping the cost per inserted element low.

## ACKNOWLEDGMENTS

Both authors were supported by NSF grants CCF-0634923 and CNS-0721491 and research grants from Cisco Systems, Inc. and Yahoo! Research. Adam Kirsch was also partially supported by an NSF Graduate Research Fellowship. The authors also thank George Varghese for several helpful discussions in the course of this research.

## REFERENCES

- [1] Y. Azar, A. Broder, A. Karlin, and E. Upfal. Balanced Allocations. *SIAM Journal on Computing*, 29(1):180-200, 1999.
- [2] L. Devroye and P. Morin. Cuckoo Hashing: Further Analysis. *Information Processing Letters*, 86(4):215–219, 2003.
- [3] M. Dietzfelbinger and C. Weidling. Balanced Allocation and Dictionaries with Tightly Packed Constant Size Bins. *Theoretical Computer Science*, 380:(1-2):47-68, 2007.
- [4] D. Fotakis, R. Pagh, P. Sanders, and P. Spirakis. Space Efficient Hash Tables With Worst Case Constant Access Time. *Theory of Computing Systems*, 38(2): 229-248, 2005.
- [5] A. Kirsch and M. Mitzenmacher. Simple Summaries for Hashing with Choices. *IEEE/ACM Transactions on Networking*, to appear. Temporary version available at: <http://www.eecs.harvard.edu/~kirsch/pubs/sshmc/ton-cr.pdf>.
- [6] A. Kirsch and M. Mitzenmacher. The Power of One Move: Hashing Schemes for Hardware. Submitted. Temporary version available at: [http://www.eecs.harvard.edu/~kirsch/pubs/ hashing \\_ hardware / hashing \\_ hardware \\_ submit . pdf](http://www.eecs.harvard.edu/~kirsch/pubs/ hashing _ hardware / hashing _ hardware _ submit . pdf).
- [7] M. Mitzenmacher. The Power of Two Choices in Randomized Load Balancing. Ph.D. thesis, University of California, Berkeley, 1996.
- [8] R. Pagh and F. Rodler. Cuckoo Hashing. *Journal of Algorithms*, 51(2):122-144, 2004.