

Peacock Hashing: Deterministic and Updatable Hashing for High Performance Networking

Sailesh Kumar, Jonathan Turner, Patrick Crowley

Washington University
Computer Science and Engineering
{sailesh, jst, pcrowley}@arl.wustl.edu

Abstract—Hash tables are extensively used in networking to implement data-structures that associate a set of keys to a set of values, as they provide $O(1)$, query, insert and delete operations. However, at moderate or high loads collisions are quite frequent which not only increases the access time, but also induces non-determinism in the performance. Due to this non-determinism, the performance of these hash tables degrades sharply in the multi-threaded network processor based environments, where a collection of threads perform the hashing operations in a loosely synchronized manner. In such systems, it is critical to keep the hash operations more deterministic.

A recent series of papers have been proposed, which employs a compact on-chip memory to enable deterministic and fast hash queries. While effective, these schemes require substantial on-chip memory, roughly 10-bits for every entry in the hash table. This limits their general usability; specifically in the network processor context, where on-chip resources are scarce. In this paper, we propose a novel hash table construction called *Peacock hash*, which reduces the on-chip memory by more than 10-folds while keeping a high degree of determinism in performance. This significantly reduced on-chip memory not only makes Peacock hashing much more appealing for the general use but also makes it an attractive choice for the implementation of a hash hardware accelerator on a network processor.

Index Terms— System design, Simulations, Network measurements

I. INTRODUCTION

Hash tables are used in a wide variety of applications. In networking systems, they are used for a number of purposes; including load balancing [8, 9, 10, 11], TCP/IP state management [21], and IP address lookups [12, 13]. Hash tables are often attractive implementations since they result in constant-time, $O(1)$, query, insert and delete operations [3, 6]. However, as the table occupancy, or load, increases collisions occur frequently, which in turn reduces the performance by increasing the cost of the primitive operations. While the well known collision resolution policies maintain good average performance despite high loads and increased collisions, the performance nevertheless becomes highly non-deterministic.

With the aid of input queuing, non-deterministic performance can be tolerated in simpler single threaded systems, wherein a single context performs the hashing operations. However, in sophisticated systems, like network processors, which utilize multiple thread contexts to perform the hash operations, a high

degree of non-determinism in performance may not be tolerable. The primary reason is that, in such systems, the slowest thread usually determines the overall system performance. As the number of threads each having a non-deterministic performance increases, the slowest thread tends to become increasingly slower, and the system performance starts to degrade quickly. Thus for such systems, it is critical to keep the hashing performance more deterministic, even though it comes at a cost of the reduced average performance (e.g. d -ary hash reduces the average performance by d -folds, however, they results in very deterministic hashing behavior).

A recent string of papers [22, 23] have been proposed, that enables highly deterministic hash operations; additionally, they also provide better average performance. While effective, these schemes require substantial on-chip memory, roughly 10- bits for every table element in the hash table. This dampens their appeal, especially in the context of network processors, where on-chip resources are expensive and scarce. In this paper, we propose a novel hash table, called a *Peacock hash*, which reduces the on-chip memory by 10-folds as compared to the best known earlier methods, while maintaining the same degree of deterministic performance.

Peacock hash table employs a large main table and a set of relatively small sub-tables of decreasing size, in order to store the elements. These sub-tables whose size follows a decreasing geometric sequence, forms a hierarchy of collision tables; the largest sub-table accommodates the elements that incur substantial collision in the main table; the second largest subtable acts as a collision buffer for the largest sub-table, and so on. It turns out that, if we will appropriately dimension the sub-tables, then we can limit the collision lengths in the main table as well as in each of the sub-tables, or even avoid having collision chains, which will lead to highly deterministic hash operations. Peacock hash enables $O(1)$ hash operations, despite having multiple hash tables, by employing on-chip filters for each sub-table. Since filter is not used for the main table, which is the largest often representing 90% of the total hash table memory, the filter memory is reduced significantly.

While conceptually simple, there are several challenges that arise in the design of Peacock hashing. The most pronounced problem is the imbalance in the sub-table loads, which arise during updates (delete and insert operations). Another problem is associated with the memory utilization: if we maintain a few relatively small sub-tables and enforce stringent collision length restrictions, then the memory utilization can become

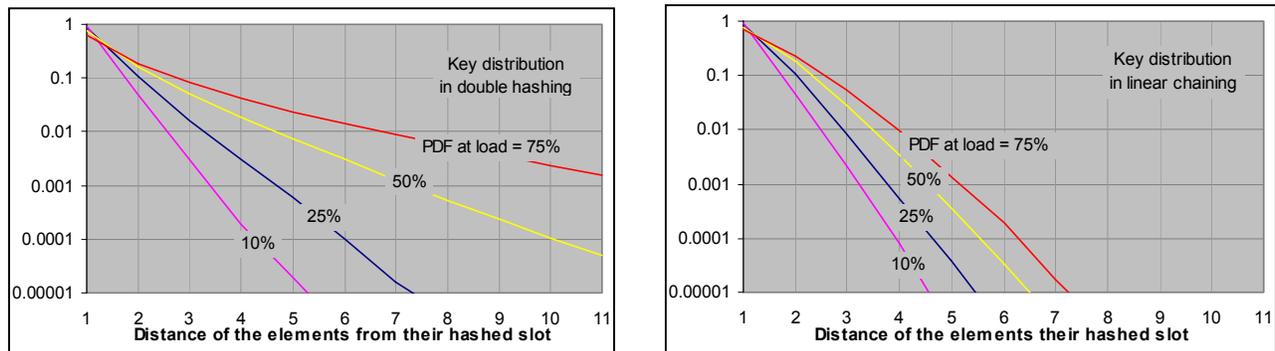


Figure 1. PDF (in base 10 log scale) of distance of elements from their hashed slots in double hashing and linear chaining

pathologically low. We address these problems with ePeacock hash, and show that it enables good memory efficiency and provides fast and stable updates without any load imbalance. The remainder of the paper is organized as follows. Section II discusses why we need a better hashing. Section III describes Peacock hashing in greater detail. Section IV discusses the memory efficiency issues and addresses the concerns about load imbalance. Section V presents the implementation of onchip filters and reports the simulation results. Section VI considers related work. The paper concludes with Section VII.

II. WHY BETTER HASHING?

Hash tables provide excellent average-case performance as a sparse table results in constant-time, $O(1)$, query, insert and delete operations. Performance however degrades due to the collisions; newly inserted elements that collide with the existing elements are inserted far from their hashed slot thereby leading to an increase in the length of the probe sequence which is followed during the query. Long probe sequences not only degrades the average performance, but also makes the performance non-deterministic: elements at the tail of the probe sequence require significantly more time for query than the elements near the head of the probe sequence. Non-deterministic query time is not desirable in real-time networking applications and it makes the system performance vulnerable to adversarial traffic. Additionally, we find that in modern multi-threaded network processor environments, such non-determinism in the hash query can become highly detrimental to the overall system performance. The system throughput drops quickly as the number of threads performing the query operation increases. We elaborate more on these properties in the subsequent sections.

A. More on hash query non-determinism

In this section, we study the characteristics of traditional hash table performance. In Figure 1, we plot the probability density function (PDF) of the distance of elements from their hashed slot (or position along its probe sequence). It is apparent that even at 25% load, approximately 1% and 2% elements remain at a distance of three or higher in linear chaining and double hashing respectively. This implies that even when four times more memory is allocated for the hash table; more than 1% of elements will require three or more memory references, hence three times higher query time. In a hash table with 1 million entries, 1% translates into 10,000 entries, which is clearly

wide enough to make the system performance vulnerable to adversarial traffic, considering that extreme traffic conditions in Internet, which moves several trillion packets across the world, is an everyday event.

A careful examination of the above PDF suggests that, at any given load, a fraction of elements always collides with the existing elements, and therefore are inserted far from their hashed slot. A large hash table does reduce such elements, but doesn't eliminate them. One may argue that, if such elements are somehow removed from the table, then hashing operations can be supported in an $O(1)$ worst-case time. A naïve approach can be to allocate a sufficiently large memory such that load remains low and either remove or not admit such elements that collide with the existing elements. One can argue that it can ensure deterministic performance at the cost of a small **discard or drop rates**. Unfortunately, drop rates can't be reduced to an arbitrarily low constant unless an arbitrarily oversized hash table is allocated. For instance, even at 1% load levels, any newly inserted element is likely to collide with an existing element with 1% probability. Thus, even if we over-provision memory by 100 times, 1% of the total elements will collide and have to be discarded.

Despite the collisions, the average performance of a hash table remains $O(1)$. For example, at 100% load level, linear chaining collision policy results in an average of just 1.5 memory probes per query. Thus, one can over-provision the hash table memory bandwidth by $(50+\epsilon)\%$, and employ a small request queue in front of the hash table to store the query requests to handle the randomness in the hash table query time. Using classical queuing theory results, for any given ϵ , one can appropriately dimension the input queue and also compute the average time for which a query request will wait in the request queue. The issues of vulnerability to adversarial traffic can be addressed by using secret hash functions and for added security, the hash functions may be changed frequently.

While the above approach is appropriate in the context of systems where a single context of operation performs the queries, it does not appeal to more sophisticated systems like a multi-threaded network processor [24], which employs multiple threads to hide the memory access latency. When there are multiple threads, each performing query operations, and there is constraint that the query requests complete in the same order in which they arrived, then the query throughput

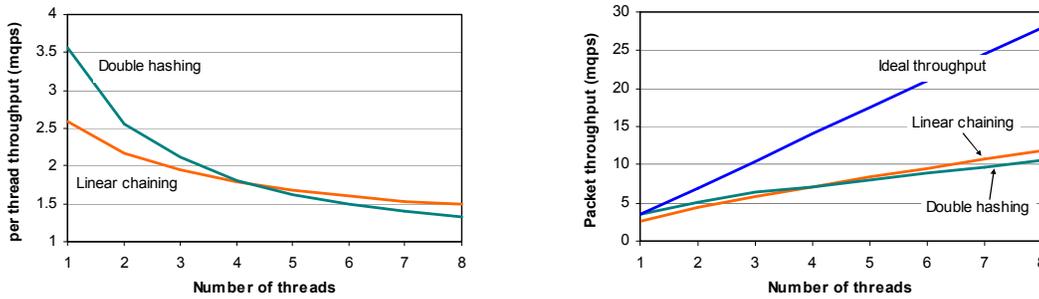


Figure 2. a) per thread hash query throughput in a multi-threaded processor, b) actual throughput in linear and double chaining based conventional hash table and ideal achievable throughput with double hashing

degrades sharply despite the presence of input request queues. We elaborate on this phenomenon in the next section.

B. Catastrophe of multi-threaded systems

Since the access time of external memory remains several processor cycles, modern network processors employ multiple threads to hide the memory latency. The threads collectively perform any given task with each running an identical code, but handling a separate packet. For example, in the case of hash table queries, each thread will query the hash table using the same algorithm; however they will be handling separate query requests. A thread switch will occur as soon as a long latency memory operation is encountered. Such a system can be balanced by adding threads, so long as idle processor cycles and memory bandwidth are available.

An important requirement in network processing systems is that, the processing of packets must finish in the same order in which the packets arrive; in the case of hashing, the queries must complete in the same order in which they arrive. In order to accomplish this, a commonly used technique is to synchronize the threads at the end of their operation. Thus, each thread begins its operation as a new request arrives at the input, and is allowed to proceed in any order relative to each other. However, at the end, as threads finish their operation, they are synchronized, and are allowed to service the next batch of requests only after the synchronization is complete. Thus, such thread which excels and gets out of order by finishing earlier than those threads, which started relatively earlier, has to wait at the end. While such a mechanism ensures that requests finish in the same order in which they arrive, an obvious consequence is that the overall throughput of the system will be limited by the slowest thread.

The next question is how slow is the slowest thread? While the time taken to process different packets remains comparable in a large number of applications, in the traditional hashing, the time taken to query different elements varies considerably. Due to this high variability, as the number of threads, each processing a separate hash query, increases, it becomes highly likely that the slowest thread takes considerable amount of time. This dramatically reduces the overall performance; in our network processor simulation model, we have observed this behavior. In Figure 2, we report the hash query throughput for different number of threads. The model runs at 1.4 GHz clock rate, and the memory access latency is configured at 200 cycles (both figures are comparable to the Intel IXP numbers). The load of hash table is kept at 80% and each thread services a series of random hash query requests. Linear chaining and

double hashing policy are used to handle the collisions; our implementation of linear chaining uses pointers that require an additional memory access compared to double hashing.

The plot on the left hand side reports the per-thread throughput in million queries per second or mqps. Notice that the throughput of each thread will be equal and will be determined by the slowest thread. It is clear that the per-thread throughput degrades quickly as we increase the total number of threads, because the slowest thread becomes much slower. An obvious consequence is that the overall system throughput does not scale linearly with increasing number of threads, which we report in the plot on the right hand side. Here we report the total throughput achieved with linear chaining and double hashing collision policies; we also report the ideal achievable throughput, if we can scale the performance linearly with the increasing number of threads. Clearly, the system becomes inefficient with increasing number of threads, the performance remains highly sub-optimal.

C. How can we do better?

The trouble with the multi-threaded environment arises due to the highly non-deterministic nature of hash query performance and it is important to keep the queries deterministic. A more deterministic performance can be achieved by using a d -ary hashing [1], where d potential table locations are considered before inserting any new element; location which is either empty or has the shortest probe sequence is selected. d -ary hashing quickly reduces the collisions as d increases, however the average query throughput for a given amount of memory bandwidth reduces, because at least d probes are required for any given query. One can reduce the average number of probes by half, by probing the d table entries sequentially until the element is found. However, the average query throughput per memory bandwidth of d -ary hashing remains lower than the traditional hashing. On the other hand, queries in d -ary hashing are much more deterministic due to the reduced collisions, which make it attractive for multi-threaded systems.

A recently proposed *segmented hash* technique [23] uses a variant of d -ary hashing, where it splits the hash table into d equal sized segments and considers one location from each segment for inserting any new element. Subsequently, in order to avoid d probes of the table memory, it maintains an on-chip Bloom filter for every segment, as a compact but approximate representation of its elements. For any query request, each of the on-chip Bloom filter is referred before physically probing any table segments; only those segments are probed whose filter returns a positive response that the element is present.

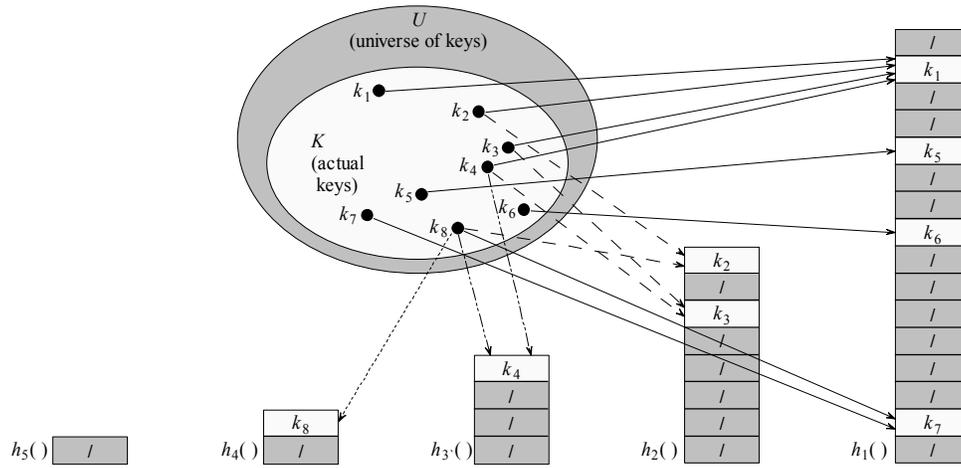


Figure 3. High level schematic and operation of a peacock hash table with 5 segments; scaling factor, r is kept at 2 and size of the largest segment, n_1 is 16. 8 keys are inserted with the collision threshold, c set at 1.

There is an alternative hash table construction introduced in [22], which also employ probabilistic on-chip filters to avoid off-chip memory accesses and enable deterministic hashing performance. Both these approaches require roughly 10-bits of on-chip memory per element to ensure that a query requires one memory probe with more than 99% probability. While such quantities of embedded memory may be acceptable in ASIC/FPGAs, they are not appealing to network processors, where on-chip memories remain scarce and expensive.

In this paper, we propose a novel hash table construction called **Peacock hashing**, which requires an order of magnitude less on-chip memory compared to the previous schemes and yet maintains the same level of determinism in the query performance.

III. PEACOCK HASHING

Peacock hashing utilizes a simple yet effective architecture wherein a hierarchy of hash table segments are used. The size of the tables in the hierarchy follows a decreasing *geometric progression* with the *common ratio* configurable. The first table, which is also the largest, is called the main table, while the remaining tables are called backup tables, since each such table acts as a collision buffer for the tables up in the hierarchy. The objective is to limit the length of the maximum probe sequence in every table to a small constant. Thus an element is inserted in a smaller table only if it collides with an element in all its predecessor tables. If we allow a finite collision length or probe sequence, then an element will be inserted in a backup table only when the probe sequence at the hashed slot of the element exceeds the collision bound in all predecessor tables. We will shortly discuss more about the insertion procedure.

Every backup table (except the main table) has an associated on-chip filter which maintains an approximate summary of the elements present in the table. The filter helps in avoiding off-chip memory accesses for those elements which are not present in the main table. There is a small probability of false positives in the filter decisions, but no false negatives. For any query, filter of each backup table is referred, and only those

backup table are probed whose filters returns a positive response. If the element is not present in any backup table (which will happen in case of false positives), or none of the filters return a positive response, then the main table is probed. Clearly, if we can maintain small false positives in the filters, then the hashing performance can be kept highly deterministic, and we will probe a single table for any given query. At the same time, if we can keep the common ratio very low, so that the backup table will be much smaller than the main table, then the on-chip filters can be kept much smaller (roughly 10% of the total elements need to be summarized in filters for a common ratio of 0.1).

While conceptually simple, there are several complications that arise during the normal insert and delete operations in a Peacock hash table. We will now formalize the Peacock hashing, which will provide us a more clear intuition into these problems, and help us in addressing it.

A. Formalization

Throughout this paper, m denotes the total number of elements stored in all tables, and m_i denotes the number of elements in the i^{th} table, thus $\sum m_i = m$. The main table is denoted by T_1 and the i^{th} backup table by T_{i+1} . The hash functions used in the i^{th} table is referred to as $h_i(\cdot)$; notice that we may use a single hash function for all segments (refer to Section IV.B). The bound on the probe sequence in any table is referred to as collision threshold and is denoted by c (c is 0, if collision is not allowed and the probe sequence is limited to 1). Size of the main table is denoted by n_1 and that of the i^{th} backup table by n_{i+1} , n denotes the total size of all tables. The ratio of size of 2 consecutive table, n_{i-1}/n_i is called **scaling factor**, r ($r < 1$). Clearly there will be $(1 + \log_{1/r} n_1)$ tables and the last table will have a single slot. In practice several small tables may be clubbed together and stored on-chip; depending upon the on-chip bandwidth, the collision threshold can also be relaxed.

B. Insertions and query

The high level schematic of a peacock hash table is shown in Figure 3. In this specific example, the scaling factor, r (ratio of the size of two consecutive hash tables) is kept at 2 and size of the largest segment is kept at 16. Eight keys are inserted into

the table and the collision threshold, c is limited to 0. There are 5 hash tables a separate hash function is used for each table. Clearly, the last hash function is the simplest one, since it always returns one. Grey cells represent empty slots while the light ones are occupied. Since collision threshold is 0, every collision results in an insert into a backup table. In the particular example, keys k_1 through k_4 are hashed to the same slot in the main table. Assuming that k_1 arrived earliest, it is inserted and the remaining are considered for insertion in the first backup table. Since a different hash function is used in this table, the likelihood of these keys to again collide is low. In this case, however, keys k_3 and k_4 collides; k_3 is inserted while k_4 is considered for insertion in the second backup table. In the second backup table, k_4 collides with a new key k_8 . k_4 this time is inserted while k_8 finds its way into the third backup table.

In general peacock hashing, trials are made to insert an arriving key first into larger tables and then into smaller ones. Keys are hashed into tables using their hash functions and then following the probe sequence (or the linked-list in chaining) until an empty slot is found or $c+1$ probes are made. Key is discarded if it doesn't find an empty slot in any table. The pseudo-code for the insertion procedure is shown below assuming an open addressing collision policy. The procedure for chaining will be similar.

```

PEACOCK-INSERT( $T, k$ )
 $i \leftarrow 1$ 
repeat
    if SEGMENT-INSERT( $T_i, k$ ) > -1 then
        return  $i$ 
    else  $i \leftarrow i+1$ 
until  $i = s$ 
error "A discard occurred"

SEGMENT-INSERT( $T_i, k$ )
 $i \leftarrow 0$ 
repeat  $j \leftarrow h(k, i)$ 
    if  $T_i[j] = \text{NIL}$ 
        then  $T_i[j] \leftarrow k$ 
        return  $j$ 
    else  $i \leftarrow i+1$ 
until  $i = c$ 
return -1
    
```

Search requires a similar procedure wherein the largest to the smallest tables are probed. Note that this order of probing makes sense because an element is more likely to be found in a larger table, assuming that both have equal load. Search within every table requires at most $c+1$ probes because of the way inserts are performed. Thus the worst-case search time is $O(s \times c)$, where $s = (1 + \log_{1/r} n)$, the total number of table segments. Also, note that this represents the absolute worst-case search time instead of the expected worst-case. If c were set at a small constant, then the expected worst-case search time will be $O(\lg \lg n)$ for open addressing as well as chaining. This is assuming that every table is equally loaded and the probability that an element is present in any given table follows a geometric distribution. This represents only a modest improvement over a naïve open addressing and almost no improvement over a naïve linear chaining. The average

search time will be $O(\lg \lg n)$, which in fact suggests a performance degrade. In practice, however, things will not be as gloomy as it may seem, because the scaling factor, r , inverse of which is the base of the log scale, will generally be low, like 0.1. This represents the condition, when every backup table is 10% of the size of its predecessor. Our preliminary set of experiments suggests that a small r indeed results in an improvement in the average and worst-case performance.

The real benefit of peacock hashing appears when we employ on-chip filters to ensure constant time query operations. A filter is deployed for each backup table (except the main table), which predicts the membership of an element with a failure probability. Such filters can be implemented using an enhanced Bloom filter as discussed in [23]; at the moment we take the filter as a black box, and represent it by f_i for the i^{th} table. It serves the membership query requests, with a constant **false positive** rate of p_f . We also assume that the filter ensures a zero false negative probability. With such filters, the search procedure looks at the membership query response of the filters and searches the tables whose filters have indicated the presence of the element. If all responses were false positives, or there were no positive response from any filters then the main table is probed. The pseudo-code is shown below:

```

PEACOCK-SEARCH( $T, k$ )
 $i \leftarrow 1$ 
repeat
    if FILTER-MEMBERSHIP-QUERY( $k$ ) > -1 then
        if SEGMENT-SEARCH( $T_i, k$ ) > -1 then
            return  $i$ 
        else  $i \leftarrow i+1$ 
    until  $i = s$ 
 $i \leftarrow 1$ 
repeat
    if FILTER-MEMBERSHIP-QUERY( $k$ ) = -1 then
        if SEGMENT-SEARCH( $T_i, k$ ) > -1 then
            return  $i$ 
        else  $i \leftarrow i+1$ 
    until  $i = s$ 
error "Unsuccessful Search"

SEGMENT-SEARCH( $T_i, k$ )
 $i \leftarrow 0$ 
repeat  $j \leftarrow h(k, i)$ 
    if  $T_i[j] = k$  then
        return  $j$ 
    else  $i \leftarrow i+1$ 
until  $i = c$ 
return -1
    
```

The average as well as the expected worst-case search times can be shown to be $O(1 + f^{\lg n})$. The exact average search time in terms of various peacock parameters can also be computed.

IV. LOADING OF PEACOCK HASH TABLE

In a general hash table, the rate at which collision occurs is independent of the table size, and depends solely upon the table load; in fact, collision probability in open addressed hash tables is precisely equal to the load. Thus, if we will perform a

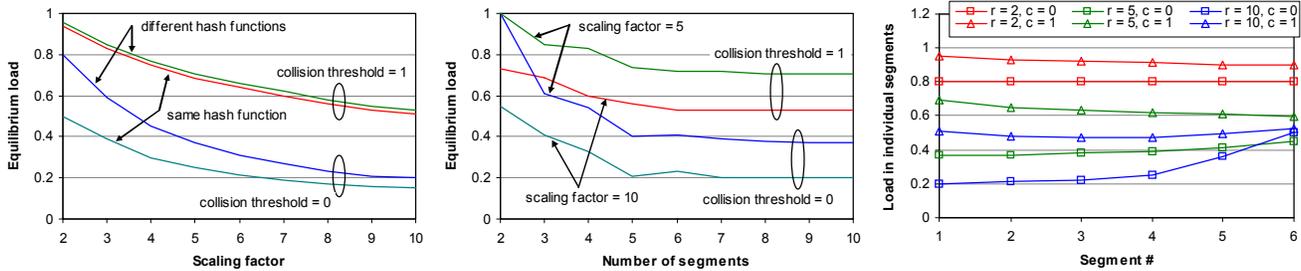


Figure 4. Equilibrium load as we vary a) scaling factor, and b) number of tables; c) load in the individual tables

series of inserts into a Peacock hash table, at the beginning when load is minimal, most of the elements will go into the 1st table segment, and only a small fraction will spill over into the 2nd table. After certain load threshold (which will depend upon the collision policy and bound on the probe sequence) is reached in the 1st table, enough elements will start to spill over to the 2nd table, and soon the 2nd table will reach a load threshold at which enough elements will start to spill to the 3rd table. The process will continue until an element reaches the smallest one slot table. From this point on, any new element may be discarded because it may not find a slot in any of the tables; we call this load of the Peacock hashing **equilibrium load**, b . At loads smaller than b , the discard rates is expected to be very small, while beyond b , there may be significant discards, thus the recommended load for safe Peacock hash table operation is b .

Clearly, it is important to investigate the characteristics of the equilibrium load in order to properly dimension the table segments and determine the memory utilization. Another interesting metric is to determine the load in each individual table at the equilibrium load. In this paper, we only report the results obtained from extensive simulations, and avoid the theoretical analysis due to space constraints.

In Figure 4(a), we report the equilibrium load for different scaling factors and for the two cases of *i*) using different hash function for each table and *ii*) using a single hash function (the implication of using single hash function is covered in more detail in Section IV.B). The collision threshold is set at 0 and 1 respectively. It is apparent that larger scaling factor leads to reduced equilibrium load, thus reduced memory utilization. Notice however that larger scaling factor is desirable, because it will reduce the on-chip memory required to implement the filters.

In Figure 4(b), we report the equilibrium load as we vary the number of table segments. The equilibrium load remains independent of the number of table, and the total size of the Peacock hash table, if we keep the scaling factor and collision threshold fixed. In Figure 4(c), we report the loading in the individual hash tables at equilibrium load. For clarity, we only report the load in the largest six tables. Again, we observe that the load in the individual tables remains roughly comparable, with slight fluctuations only in the smaller tables. This is expected, considering that fact that in Peacock hashing, each backup table provides an equal fraction of buffering for the elements spilled from its predecessor table.

A. Deletes and imbalance

While insert and query appear straightforward in Peacock hash, deletes necessitates rebalancing of the existing elements. This happens because at equilibrium load state, a series of deletes followed by inserts result in relatively high load at the smaller tables which eventually overflows them, resulting in high discard rates. We first provide an intuitive reason for this phenomenon and then report the experimental results. For simplicity, we first consider the case when collision threshold is set at 0. Let us assume that we are in equilibrium load state and each table is roughly equally loaded, *i.e.* $m_{i+1} = r \times m_i$, where m_i is the number of elements in the i^{th} table. Thereafter a stream of “one delete followed by one insert” occurs.

The first observation is that, the rate at which elements arrive at the i^{th} table ($i > 1$) depends upon the load in all of its predecessor tables. Thus, if $m_i(t)$ is the number of elements in the i^{th} table at time t , then during inserts, we will have the following differential equation:

$$m_i(t) = m_i(t-1) + \prod_{j=1}^{i-1} m_j(t-1)/n_j(t-1)$$

During deletes, the rate of deletion from a table i will be $m_i / \sum m_i$, assuming that elements being deleted are picked randomly. Thus we have the following differential equation for deletes:

$$m_i(t) = m_i(t-1) - m_i(t-1) / \sum m_i(t-1)$$

In order to compute the load of each table, we need to solve the above two differential equation. We take $m_{i+1} = r \times m_i$ as the initial values assuming that deletes starts after the equilibrium state. Clearly, a table i will overflow if,

$$\prod_{j=1}^{i-1} m_j/n_j > m_i / \sum m_i$$

At equilibrium loading, when m_i/n_i is equal for all i , this condition will be satisfied easily for smaller tables and they will quickly fill up and overflow. In fact, one can show using these two differential equations that even at loads much lower than the equilibrium load, the smaller tables will overflow. We demonstrate this with the following simple experiment. We consider a Peacock hash table with 6 segments, each using double hashing as the collision policy and with a scaling factor, r of 0.1. The collision threshold is set at 1. In the first phase, 40,000 elements are inserted one after another with no intermediate deletes, notice that this load is much lower than the equilibrium loading (refer Figure 4). In the second phase, an element is deleted and another inserted one after another for an extended period. The resulting load of each of the 6 tables is illustrated in Figure 5(a). The sampling interval for the above plot is once per 1000 events. It is clear that smaller tables fill up quickly during the second phase in absence of

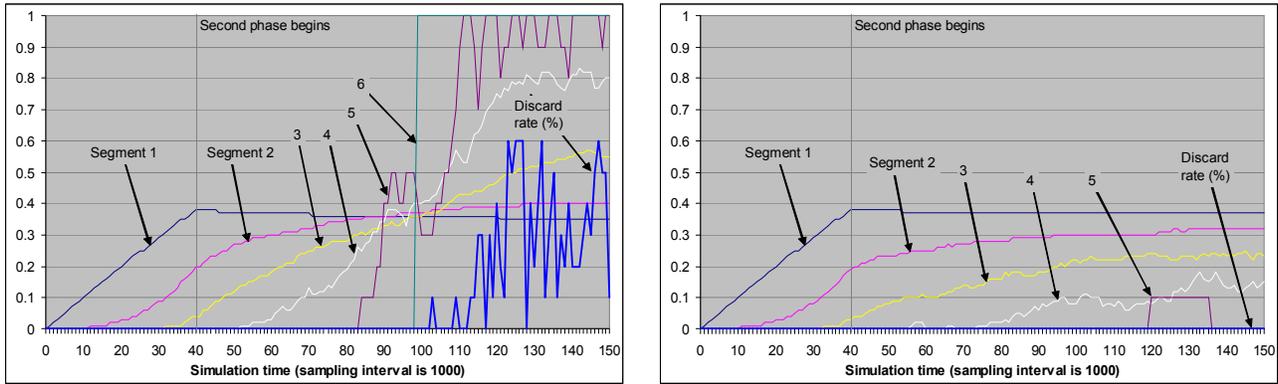


Figure 5. Plotting fill level of various segments and the percent discard rate a) without rebalancing, b) with rebalancing

any re-balancing, even when the total number of elements remains the same, and much below the equilibrium loading.

In the same setup, we now add rebalancing; however since we do not yet have a rebalancing method, we perform rebalancing by exhaustively searching the backup tables and moving the elements upwards (such rebalancing may not be practical for real-time application). We found that, with rebalancing the backup tables neither overflows, nor their load increases. In order to stress the situation further, we start to slowly add new elements in the second phase, in addition to the regular delete and insert cycle. We found that with rebalancing, the load in smaller tables remains low until the total load is smaller than the equilibrium load. The results are illustrated in Figure 5(b). It is obvious that rebalancing is critically important, without which the smaller tables will overflow despite low loads. We now present the challenges associated with the rebalancing in Peacock hashing, and present our solution to the problem.

B. Rebalancing

We define rebalancing as a process wherein we look for such elements which currently resides in a smaller backup table, and can be moved to a larger predecessor table because some elements have been deleted from the predecessor table. Once found, such elements are moved into the largest predecessor table. The general objective is to keep the elements in the larger tables, while keeping the smaller tables as empty as possible. Unfortunately, it is not feasible to do rebalancing in Peacock hash table without doing an exhaustive search, which will take linear time in the number of elements present in the table. The primary cause of this difficulty is the use of separate hash functions in each table. Consider a simple example shown in Figure 6. Here four keys are inserted and their hash slots are shown. At the end, the first key k_1 is removed. Now, k_3 , which had collided with k_1 can be moved back to the first table, however, there is no way to locate the k_3 in the second table without searching the entire table.

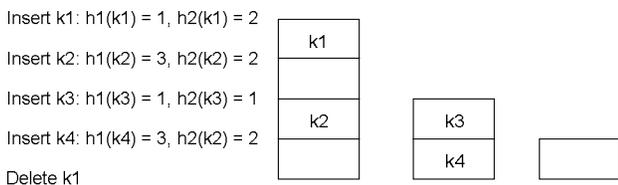


Figure 6. Four inserts and one delete in a Peacock hash table

A simple way to tackle this problem is to employ a composite sequence of hash functions. The first hash function will be applied to the incoming element, to determine the its slot in the first table: $h_1(k)$. The slot in the second table will be computed by applying a second hash function to the outcome of the first hash function: $h_2(h_1(k))$. Slots in the remaining table segments will be computed similarly. This scheme will ensure that all those elements that collide at slot s of table i will be moved to unique slots in the descendent backup tables, defined by $h_{i+1}(s), h_{i+2}(h_{i+1}(s))$, and so on. Thus, for any element deleted from a table i , we can precisely locate those slots in the descendent backup table s from where elements can potentially be moved back to the table i .

A simpler and more practical method to implement the above method is to use a single hash function h with range $[1, n_1]$ which will determine the slots in the first table. The slots in the backup tables will simply be the modulo (table size) of the slot in the first table, i.e. $h(\) \bmod n_i$. This method will also ensure that all elements that collide at a slot of a table will be moved to unique slots in the descendent backup tables, thus they can be easily moved back upon deletes in the larger tables.

This method eases rebalancing as only one probe has to be done in each descendent table upon a delete. However, there may be implications on the memory efficiency and equilibrium load, which we report in Figure 4(a). Clearly, use of a single hash function reduces the equilibrium loading, dramatically if collision threshold is 0, and mildly for higher collision thresholds. Figure 7 explains this reduction. With single hash function method described above, if several elements collide at a slot in a table, then all of them will land up at identical slots in the backup tables and therefore some may not find any empty slot and will be discarded. On the

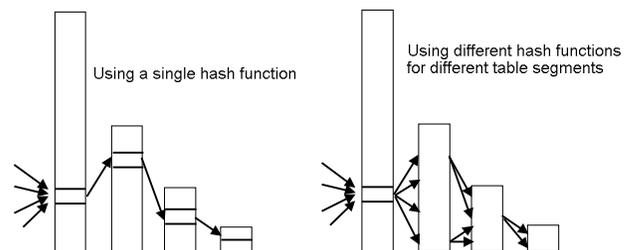


Figure 7. Using a single hash function for all tables versus using separate hash function for each table

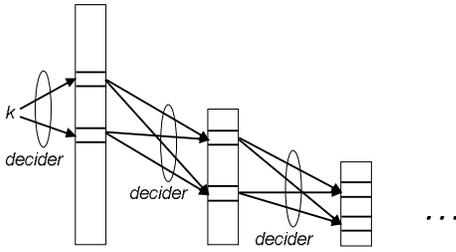


Figure 8. Illustrating hash method used in ePeacock hashing

other hand, using different hash function insulates us from this problem, as illustrated in the diagram on the right hand side. The key observation is that with different hash functions, elements that collide at a single slot in a table get scattered at *multiple*, and in this case random, slots in the backup tables. However the randomness in the slots in backup tables leads to complication with the rebalancing.

C. Enhanced Peacock Hashing

Quite intuitively one may address the above problem of reduced equilibrium loading by devising a method such that elements collided at any single slot will have multiple but predetermined, and not random, slots in the backup tables, so that they can be quickly pulled back in face of deletes. We call this enhanced Peacock hashing or ePeacock hashing.

In ePeacock hashing, we employ l hash functions each with range $[1, n_1]$; the input element is hashed by these l functions to determine l slots in each table, applying modulo (table size) for smaller tables. Afterwards, a separate hash function for each table called its *decider* chooses a slot among the l slots in the table. These chosen slots are then used to insert the newly arrived element. The scheme is illustrated in Figure 8 for $l=2$. In this case, the collided elements at any slot in a table will have 2 slots available in the backup tables, thus the equilibrium load should become better than the case where a single hash function is used. In general, as we increase l , equilibrium loading should improve and approach that of the Peacock hashing, where separate hash functions are used for each table. A tradeoff with increased l is that the rebalancing will require l queries per table. In Figure 9, we report the difference between the equilibrium load of ePeacock hashing and the normal Peacock hashing which uses separate hash function for each table, as we vary l . It is clear that, as we increase l , equilibrium load of ePeacock hash improves rapidly and approaches that of the Peacock hash.

A tradeoff with increased l , however is that the rebalancing will require l -fold higher number of probes per table. Therefore, we would like to keep l small. Fortunately, $l=2$, ensures an equilibrium load which is roughly comparable to the equilibrium load when we use separate hash functions. Thus, ePeacock hash can enable good memory utilization and fast rebalancing, half as fast as the Peacock hash.

V. IMPLEMENTING THE ON-CHIP FILTERS

Recall that the on-chip Bloom filters are used in Peacock hashing to avoid excessive off-chip memory accesses required to service a query. Peacock hashing employs a separate filter for each table segment except for the first table which is the

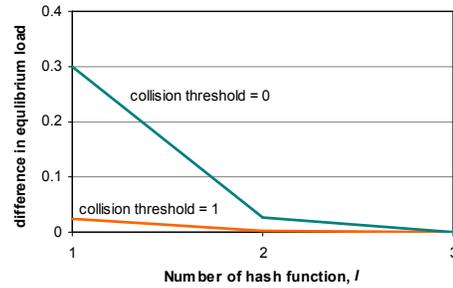


Figure 9. Plotting difference in the equilibrium load of Peacock hash and ePeacock hash table

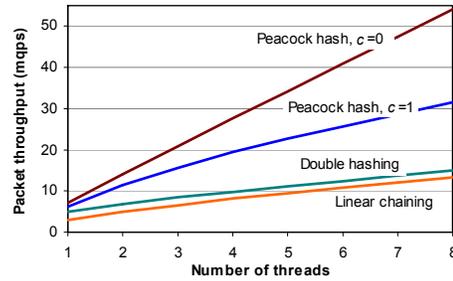


Figure 10. Plotting difference in the equilibrium load of Peacock hash and ePeacock hash table

largest. The filters maintain the summary of the elements that are stored in the respective hash table, and we propose to employ segmented Bloom filter technique [23] to implement these filters. Segmented Bloom filters require roughly 10-bits per entry to enable false positive rates smaller than 0.01. Since deletes have to be supported, counters will be required in addition to the Bloom filter bit-masks; these counters can be however kept in the off-chip memory and only the bit-masks will be required on-chip.

The key benefit of Peacock hashing over recently proposed schemes like fast hash table [22] and segmented hash [23] is that filter is not used for the largest table; thus only a small fraction of the total elements are stored in the filters. For large scaling factors, this will significantly reduce the filter size. High scaling factor however leads to low hash table memory utilization, therefore, striking the right tradeoff will require the information about the availability and cost of the off-chip and on-chip memory and bandwidth. With the current trend of offchip memory becoming significantly cheaper than the cost of the die area available in silicon devices, we recommend using large scaling factors, as high as 10 or even 20. Such scaling factors will lead to a 10-fold reduction in the on-chip memory.

We finally report the performance of Peacock hash table in a multi-threaded system environment and compare it to that of the traditional hashing (double hashing and linear chaining) and the recently proposed fast hash table and segmented hash table. We use a Peacock configuration with scaling factor of 10 and collision threshold of 0 and 1, and employ double hashing as collision policy. The resulting query throughput is reported in Figure 10. It is clear that, Peacock hash results in a much higher performance. A concern however remains that at $c=0$, the memory utilization is as low as 20%, while for $c=1$,

the memory utilization is 50%. Traditional hashing can clearly support higher memory utilization, albeit at the cost of reduced performance. Provided that the off-chip memory bandwidth is much more expensive than the size, such utilization levels may be acceptable. Segmented hash and fast hash table, on the other hand, provides memory utilization comparable to the Peacock hashing, thus Peacock hashing remains significantly superior and efficient by using 10-fold less on-chip resources.

VI. RELATED WORK

A considerable amount of related work has focused on the theoretical aspects of improving hash table performance. The power of multiple choices has been studied by several researchers; [1, 2, 3, 4] are some of the earliest results to suggest that the number of collisions can be reduced significantly with two random choices. These studies argue that using two hash functions to index the hash table, and subsequently choosing the one with fewest keys, reduces collisions and hence probe sequence lengths. Another study [18] examines the use of two independent tables, accessed serially, and shows that proper placement and access strategies can enhance overall performance. Another study [13] describes an approach for obtaining good hash tables for IP lookups based on using multiple hashes of each input key. While most of these results are compelling, they focus on latency while ignoring memory bandwidth. In the packet processing context, where memory bandwidth is at a premium, it is important to minimize the number of memory references needed for each hash operation. Recently proposed architectures [22, 23] attempts to ensure that a single memory reference is used on every query (i.e. one bucket is read) with high probability.

VII. CONCLUDING REMARKS

Recently a number of novel hash tables (fast hash table and segmented hash) have been proposed which are dedicated for the high performance networking applications. These hash tables aim at enabling highly deterministic query, and improving the worst-case, thereby insulating the system from adversarial traffic patterns and potential attacks. While they are effective, they usually require large on-chip memory, which till date remains a major obstacle in their widespread adoption. On-chip memory is expensive and over time, it is expected to become even more expensive compared to the external memory. Therefore it is important to propose methods to reduce the on-chip memory in high performance hash tables. In this paper, we propose Peacock hashing, which improves on the earlier high performance and deterministic hash tables, by reducing the on-chip memory by more than 10-folds. We believe that such reductions in the on-chip memory will not only make Peacock hashing appealing to custom ASIC/FPGAs but will also open up the possibility of getting deployed in network processors. A hardware based high performance hash table can be implemented in a network processor, to be used by all processors/threads. The unit will use a small on-chip memory (roughly 1-bit per element) for the filters, and will also control the off-chip hash table

memory. All primitive hash table functions can then be made available to the processors via specialized APIs, thus the processors can deploy hash tables much more efficiently and with greater confidence, provided by the deterministic behavior of Peacock hashing.

REFERENCES

- [1] Y. Azar, A. Broder, A. Karlin, E. Upfal, Balanced Allocations, Proc. 26th ACM Symposium on Theory of Computing, 1994, pp. 593–602.
- [2] G. H. Gonnet, Expected length of the longest probe sequence in hash code searching, Journal of ACM, 28 (1981), pp. 289-304.
- [3] M. L. Fredman, J. Komlos, E. Szemerédi, Storing a sparse table with $O(1)$ worst case access time, Journal of ACM, 31 (1984), pp. 538-544.
- [4] J. L. Carter, M. N. Wegman, Universal Classes of Hash Functions, JCSS 18, No. 2, 1979, pp. 143-154.
- [5] P. D. Mackenzie, C. G. Plaxton, R. Rajaraman, On contention resolution protocols and associated probabilistic phenomena, Proc. 26th Annual ACM Symposium on Theory of Computing, 1994, pp. 153-162.
- [6] A. Brodnik, I. Munro, Membership in constant time and almost-minimum space, SIAM J. Comput. 28 (1999) 1627–1640.
- [7] M. Dietzfelbinger, A. Karlin, K. Mehlhorn, F. Meyer, H. Rohnert, R. Tarjan, Dynamic Perfect Hashing- Upper and Lower Bounds, Proc. 29th IEEE Symposium on Foundations of Comp. Science, 1988, pp. 524-531.
- [8] B. Vocking, How Asymmetry Helps Load Balancing. Proc. 40th IEEE Symposium on Foundations of Comp. Science, 1999, pp. 131-141.
- [9] M. Mitzenmacher, The Power of Two Choices in Randomized Load Balancing, Ph.D. thesis, University of California, Berkeley, 1996.
- [10] Y. Azar, A. Z. Broder, A. R. Karlin, E. Upfal, Balanced allocations (extended abstract), Proc. ACM symposium on Theory of computing, May 23-25, 1994, pp. 593-602.
- [11] M. Adler, S. Chakrabarti, M. Mitzenmacher, L. Rasmussen, Parallel randomized load balancing, Proc. 27th Annual ACM Symposium on Theory of Computing, 1995, pp. 238-247.
- [12] M. Wadvoel, G. Varghese, J. Turner, B. Plattner, Scalable High Speed IP Routing Lookups, Proc. of SIGCOMM 97, 1997.
- [13] A. Broder, M. Mitzenmacher, "Using Multiple Hash Functions to Improve IP Lookups", IEEE INFOCOM, 2001, pp. 1454-1463.
- [14] W. Cunto, P. V. Poblete: Two Hybrid Methods for Collision Resolution in Open Addressing Hashing, SWAT 1988, pp. 113-119.
- [15] T. H. Cormen, C. E. Leiserson, R. L. Rivest, Introduction to Algorithms, The MIT Press, 1990.
- [16] P. Larson, Dynamic Hash Tables, CACM, 1988, 31 (4).
- [17] M. Naor, V. Teague. Anti-persistence: History Independent Data Structures. Proc. 33rd Symposium on Theory of Computing, May 2001.
- [18] R. Pagh, F. F. Rodler, Cuckoo Hashing, Proc. 9th Annual European Symposium on Algorithms, August 28-31, 2001, pp.121-133.
- [19] D. E. Knuth, The Art of Computer Programming, volume 3, Addison-Wesley Publishing Co, second edition, 1998.
- [20] L. C. K. Hui, C. Martel, On efficient unsuccessful search, Proc. 3rd ACM-SIAM Symposium on Discrete Algorithms, 1992, pp. 217-227.
- [21] G. R. Wright, W.R. Stevens, TCP/IP Illustrated, volume 2, Addison-Wesley Publishing Co., 1995.
- [22] H. Song, S. Dharmapurikar, J. Turner, J. Lockwood, "Fast Hash Table Lookup Using Extended Bloom Filter: An Aid to Network Processing," SIGCOMM, Philadelphia PA, August 20-26, 2005.
- [23] S. Kumar, and P. Crowley, "Segmented Hash: An Efficient Hash Table Implementation for High Performance Networking Subsystems", IEEE/ACM Symposium on Architectures for Networking and Communications Systems (ANCS), Princeton, October, 2005.
- [24] M. Adiletta, et al. "The Next Generation of Intel IXP Network Processors," Intel Technology Journal, vol. 6, no 3, pp. 6-18, Aug 2002.