A Dynamic Load-Balanced Hashing Scheme for Networking Applications

N. Sertac Artan, Haowei Yuan, and H. Jonathan Chao
Department of Electrical and Computer Engineering
Polytechnic Institute of NYU

***Abstract*—Network applications often require large data storage resources, fast queries, and frequent updates. Hash tables support these operations with low costs, yet they cannot provide worst-case guarantees because of hash collisions. Also, the widely used, low-cost Dynamic Random Access Memory (DRAM) cannot suitably accommodate hash tables because DRAMs provide full bandwidth only if accessed in bursts, whereas hash tables require random access. In this paper, we propose a hash co-processor to support hash tables on DRAMs. The co-processor provides a load-balancing method to reduce the impact of hash collisions on the worst-case behavior by moving multiple keys within the hash table in constant time. This leads to a balanced distribution of keys in the hash table despite the collisions. Furthermore, the co-processor guarantees the full DRAM bandwidth is always utilized by defining all fundamental hash table operations, namely insert, query, and delete, in terms of burst accesses. In the worst case, the query, delete, and insert operations take one, two, and three burst accesses, respectively. The proposed architecture reduces hash overflows by 35% compared to a naïve hash table and for each key uses 6.42 bits of on-chip memory.**

I. Introduction

Network applications such as IP traceback [1], route lookup [2], TCP flow state monitoring [3] [4], and malware detection [5] [6] often require large data storage resources, fast queries, and frequent updates. Hash tables are traditional data structures that allow large amounts of data to be stored, queried, and updated in a space- and time-efficient manner on average. However, in the worst case, which is critical for network applications, hash tables perform poorly. This poor performance is due to hash collisions, which require multiple keys to be accessed and examined sequentially. To reduce the impact of hash collisions on worst-case performance, the hash table can be modified to store multiple keys, say up to G_{max} keys in a single hash bucket, where up to G_{max} keys can be accessed together. Hence, as long as the number of collisions in a bucket is less than G_{max}, the worst-case performance is determined. Unfortunately, even when multiple keys are allowed to be stored in one bucket, occasional overflows cannot be prevented.

Additionally, the standard bulk data storage device, the Dynamic Random Access Memory (DRAM), cannot suitably accommodate hash tables. This is because DRAM provides full bandwidth only if it is accessed in a certain sequence of bursts [7]. This burst-access requirement for full bandwidth conflicts with the random-access behavior of hash tables and, thus, it adversely affects hash table performance.

In this paper, we propose an on-chip co-processor to support hash tables on the DRAM for network applications as shown in Figure 1a. This paper proposes two main contributions: (1) A load-balancing (LB) method to reduce the number of overflows in a hash table and (2) a co-processor architecture that uses the full bandwidth of the DRAM.

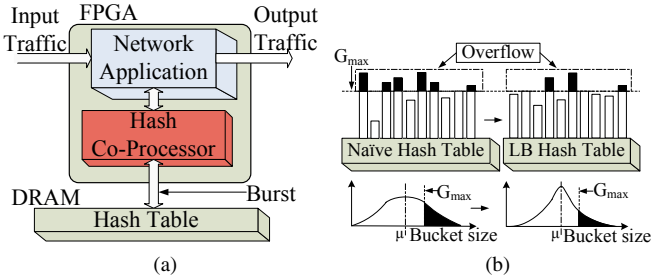

Fig. 1: (a) The proposed co-processor architecture. (b) The LB hash table reduces number of overflows compared to the naïve hash table and provides an even distribution of keys, where the bucket size (*i.e.,* number of keys in a bucket) distribution is denser around the average load, μ.

As shown in Figure 1b, the keys hashed to a bucket in excess of G_{max} cause overflows. These keys need to be stored elsewhere, such as an on-chip Content-Addressable Memory (CAM), which is very limited. If a new key is hashed to a bucket with G_{max} keys, our proposed LB method prevents a significant portion of overflows by moving some of the keys from this bucket to a lightly loaded bucket, opening a vacancy for the new key. In addition, the move makes the distribution of keys in the hash table more balanced, reducing the chance of a future overflow. Using the LB method, the CAM requirement is reduced as a result of the reduction in overflows.

The proposed co-processor architecture uses the full DRAM bandwidth by always obeying the burst-access pattern of the DRAM for all three fundamental hash table operations, namely insert, query, and delete. We show that our architecture reduces the number of overflows by 35% compared to a naïve hash table, while using 6.42 bits of on-chip memory per key including a tiny on-chip CAM to store the 0.2% of the keys. The insert operation takes 1.38 and 3 burst accesses in average and worst cases, respectively. The query and delete operations take one and two burst accesses for both cases, respectively.

The rest of this paper is organized as follows: In Section II, we summarize the related work. The proposed co-processor architecture is presented in Section III and analyzed for various

978-1-4244-2324-8/08/$25.00 © 2008 IEEE.

parameters in Section IV. Section V shows performance of the proposed architecture. Section VI concludes the paper.

II. Related Work

Recently, multiple choice hashing schemes, especially Cuckoo Hashing [8], demonstrated encouraging results for hash tables. However, the Cuckoo Hashing scheme requires possibly moving multiple keys in the hash table, which limits hardware implementation. In [9], it is shown that with rigorous analysis, a hardware implementation of this scheme is possible by limiting the number of moves to a single move and using a CAM to store a small percentage of the keys. However, the authors did not specify a hardware architecture for their proposed scheme: thus, the exact performance of the scheme in practice is not clear. There is also recent work to improve access performance [10] [11] of external hash tables using on-chip structures. In [10], a Counting Bloom Filter (CBF)-based scheme [12] is proposed to improve the performance of naïve hash tables using high-speed, on-chip memory. The on-chip memory required for the scheme is reported as 400 kbits per $10,000$ keys or 40 bits/key, which can be reduced by approximately 50% by optimizing the construction of the CBF [13]. In [11] another BF-based scheme is proposed to improve the distribution of keys in a multiple choice hash table to minimize collisions and, thus, increase throughput using an on-chip memory of 32 bits/key. In both of these schemes, an expected external memory access around one is reported; however, this is only under the assumption that a majority of accesses are queries to the keys not in the hash table.

While these schemes provide a very rich variety of options for optimizing hash tables, the underlying memory structure, DRAM, which with its slowness is in fact the main source of the problem, is often neglected. Due to technological limitations, in between two accesses, DRAMs need to be reconditioned, *i.e., the bit lines need to be precharged*. In this time period, the DRAM cannot be accessed. To hide these idle periods and thus improve throughput, DRAMs are arranged in multiple banks. In this arrangement, each bank is accessed in turn, and each bank is precharged while the other banks are accessed. This access model favors burst accesses, where a series of accesses are made to banks with the condition that no bank is accessed twice in a single *burst period*[1]. This way, the DRAM throughput is maximized. To utilize all the available DRAM bandwidth, the hash table should be organized such that accesses should be done in terms of bursts rather than random individual accesses. In the following section, we propose an architecture that groups the keys in a hash table that will always follow the burst-access pattern while providing load balancing. Note that, although the proposed architecture uses a single hash function, it can be extended to the multiple choice hashing scenario.

[1]In a burst period, a predetermined number of consecutive locations from one bank can be accessed. Here, we assume these consecutive locations belong to a single key and are accessed as a single word stored at a single address.

III. The Hash Co-Processor

Figure 2 shows the proposed hash co-processor architecture and the DRAM storing the external hash table. The hash table has a *capacity* of C keys and it is divided into g blocks (*i.e.,* hash buckets), where each block consists of one word from each bank and can be accessed in one burst. Each block corresponds to a *group* on the chip. We define $loc[G] \rightarrow B$ as a pointer that points to the B^{th} key in group G which is physically stored in bank B at memory address G in the DRAM.

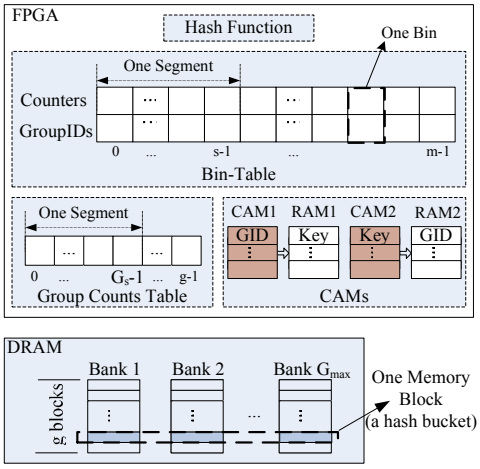

Fig. 2: Hash Co-Processor Architecture

The co-processor consists of four parts (1) Group Counts Table (GCT), (2) Bin-Table (BT), (3) Hash Function, and (4) CAMs. The GCT consists of g entries, where each entry, GCT_i shows the number of keys currently in the group i. Each group can accommodate up to G_{max} keys. BT has m bins addressed by the hash function. Each bin in BT consists of two fields, a *Counter* and a *GroupID*. The counter shows the number of keys currently hashed to this bin. Each bin belongs to a single group shown by the GroupID. The keys are first hashed to these bins and the bins are distributed to groups. All the keys, which are hashed to the bins with the same GroupID i, are stored in the external memory block, i. CAMs are used as temporary storage for keys that cannot be placed into a group immediately. The keys in the CAMs can be deleted or queried as other keys stored in the hash table. For each key that requires temporary storage, CAM1 stores a GroupID assigned to this key, while the key itself is stored in an on-chip auxiliary RAM, RAM1, using the same address. CAM2 stores the key and the GroupID of the key is stored in RAM2 at the same address. CAM1 is indexed with the GroupID, while CAM2 is indexed with the key. To aid movement of bins from one group to another as described below, both GCT and BT are divided into equal-sized segments. Let the number of groups in a single segment of GCT be G_s and the number of bins in a single segment of BT be s. Next, the three operations in the co-processor are described.

A. Insertion

In this section, we first show a naïve insert algorithm we called *Non-Search Algorithm* (NSA), which immediately stores a key that causes an overflow in CAMs, to provide a basis for our discussion. Later, we show two new algorithms that achieve load-balancing during the insert operation.

A new key is inserted into the hash table in two steps. First, the co-processor assigns a group to the key. Then, the key is written to the location in the hash table allocated for this group.

1) Group Assignment: To assign a group to a new key, the new key is first hashed and the corresponding bin, b, is determined. The new key is assigned to the group to which bin b belongs. Note that even if a bin is empty, it belongs to some arbitrary group.

2) Writing the Key: If bin b belongs to a group i with vacancies (*i.e.,* $GCT_i < G_{max}$), bin b's counter and GCT_i are incremented by one and the key is inserted into the hash table at the location pointed to by $loc[i] \rightarrow GCT_i$. However, if this group is already full, the new key is said to cause an *overflow* and it is stored in the CAMs until this group has a vacant space.

3) Insertion with Load-Balancing: In this section, *Single-Search Algorithm* (SSA) and *Double-Search Algorithm* (DSA), which provide insertion with load-balancing, are described. These algorithms move keys from a group that overflows to open a vacancy in the hash table for a new key causing an overflow, rather than immediately storing this key in the CAMs. This way, (1) precious CAM space is saved, and (2) the external hash table becomes more balanced by moving the load from a full group to a lightly loaded group thus avoiding further overflows.

When a new key causes a group to overflow, its prospective bin in BT is recognized as a *troubled bin* (b_t) and the group it belongs to as the *troubled group* (g_t). Note that since each bin belongs to only one group, only one group can be assigned to the keys hashed to a bin. The *Single-Search Algorithm* first finds the group with a minimum number of keys. To avoid a long search, only the local minimum within the segment, which includes b_t is considered. Let us call this group g_{min} and let $|g_{min}|$ show the number of keys in g_{min}. If g_{min} can accommodate the b_t along with the new key, specifically if $|g_{min}| + |b_t| < G_{max}$, then the bin b_t is *moved* from g_t to g_{min} as described below. Otherwise, since g_{min}, the group with the most vacant space cannot accommodate b_t, no other group in this segment can. Therefore, the new key is stored in the CAMs until the hash table can accommodate it or it is deleted.

A bin b is *moved* from one group g_{src} to another group g_{dst} as follows. The GroupID of b is changed from g_{src} to g_{dst}, thus g_{dst} is assigned to all keys in b. To reflect this change in the external memory, first all the keys in g_{src} are read and, by means of hashing, the keys hashed to bin b are found. The remaining keys are written to the hash table starting at location $loc[g_{src}] \rightarrow 1$. Additionally, the keys in bin b (as well as the new key, if it is associated with b) are written to the hash table starting at location $loc[g_{src}] \rightarrow GCT_{g_{dst}} + 1$. $GCT_{g_{src}}$ and $GCT_{g_{dst}}$ are changed to reflect the new key counts in these groups.

Compared to NSA, SSA needs a smaller number of keys to be stored in CAMs. Additionally, SSA balances the load within a segment by promoting the movement of load from an overflowing group to the least occupied group in the segment.

In fact, we can optimize further. Note that the group is overflown by just one key, so all we need to do is make this group size one less. Even if g_{min} cannot accommodate b_t, it may accommodate another bin that belongs to g_t and moving this bin is enough to avoid writing the new key to the CAM. DSA provides a solution for this case. The algorithm searches for a smaller bin b_s in g_t, that g_{min} can accommodate. If there is such a b_s, then the new key is stored in g_t, b_s is moved to g_{min}, and the insertion is completed. Otherwise, the new key is stored in CAMs. Using DSA, the CAM size is smaller as shown in Section V, with the cost of increased insertion time due to the search time for b_s. Both SSA and DSA require additional time to search; however, this search is done on-chip. Thus it is less critical when compared to external memory access. The time complexities of this search and external memory access are given in Section IV.

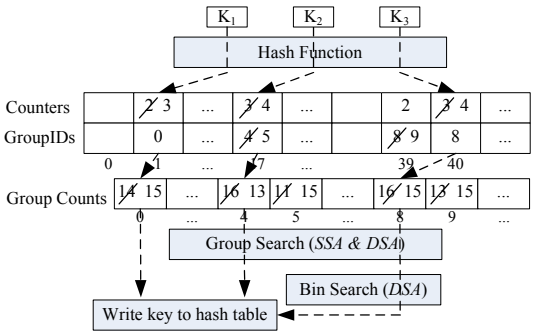

Fig. 3: Examples for insert operation. $G_{max} = 16$.

Figure 3 shows insertion of three keys, K_1, K_2, and K_3. K_1 is hashed to bin b_1, which belongs to group g_0. g_0 has 14 keys and $G_{max} - 14 = 2$ vacancies, so it can accommodate a new key. Hence, b_1's counter and g_0's entry are incremented by one. K_1 is inserted into the hash table at $loc[g_0] \rightarrow 15$.

K_2 is hashed to bin b_{17}, which has three keys and belongs to a full group, g_4. The three algorithms have different procedures to insert K_2. If NSA is applied, K_2 is written to CAMs. For SSA and DSA, first g_{min} is searched in the segment of b_{17}. The search results in $g_{min} = g_5$ with 11 keys. g_5 can accommodate b_{17} along with K_2. b_{17}'s counter is incremented by one and then it is moved from g_4 to g_5 as described above.

K_3 is hashed to bin b_{40}, which belongs to another full group, g_8. The $g_{min} = g_9$ in this segment cannot accommodate b_{40} together with the new key, K_3 $(13 + 3 + 1 > 16)$. SSA stops at this point and writes K_3 to CAMs. In DSA, bin search is applied and it is found that g_9 can accommodate b_{39} from g_8 with two keys. b_{39} is moved from g_8 to g_9. b_{40}'s counter and g_8's entry are incremented by one and K_3 is inserted to $loc[g_8] \rightarrow (16 - 2 + 1) = loc[g_8] \rightarrow 15$. If even bin search

cannot find a suitable bin, the new key is stored in CAMs.

B. Query

Query returns the data associated with a queried key if it is stored in the system. If in the system, the queried key is either in the hash table or in the CAMs. If the key is found in the CAMs, it is returned immediately. Otherwise, the group of the bin where the key is hashed is read as a single burst from the hash table. The keys in this group are compared to the queried key to see whether the key is in the system. The result of this comparison is returned. So, the query operation takes at most one burst access.

C. Deletion

Here, we assume the key to be deleted is already in the system. To delete a key, the key is first queried. If the key is in the CAMs, it is removed from the CAMs and deletion is completed. If this key is not in the CAMs, the group to which this key belongs is read as a single burst from the off-chip structure. The key is deleted and the remaining keys are written back to the off-chip structure in another burst. So, the delete operation takes at most two burst accesses.

Each key deletion can be an opportunity to move a key from the CAMs to the hash table. Each time a key is deleted, CAM1 is queried to see whether there is any key from the same group in the CAMs. If so, this key can be moved to the hash table with an insert operation. This way, the number of keys in the CAMs can be reduced.

IV. Analysis

To evaluate the proposed architecture, three metrics are defined (1) on-chip memory requirement, (2) time complexity of on-chip operations, and (3) time complexity of external hash table operations in terms of burst accesses.

A. On-chip Memory Consumption

The on-chip memory is used for BT, GCT, and the CAMs (including RAM1 and RAM2). Each counter value in BT can be $\{0 \dots G_{max}\}$ to allow full groups with only one bin. Each GroupID value can be $\{0 \dots G_s - 1\}$, where G_s is the number of groups per segment and GroupIDs are local to each segment. Thus, one bin requires $M_b = \lceil \log_2(G_{max} + 1) \rceil + \lceil \log_2(G_s) \rceil$ bits. Recall that there are m bins in BT. Additionally, there are g entries in GCT and each of these entries can be $\{0 \dots G_{max}\}$. Therefore, each entry in GCT requires $M_g = \lceil \log_2(G_{max}+1) \rceil$ bits. CAM1 and RAM2 store identical information (GroupIDs of temporarily stored keys). Likewise, CAM2 and RAM1 also store identical information (temporarily stored keys). Thus, the total CAM storage is equivalent to the total storage of RAM1 and RAM2. This total CAM storage can be expressed as $M_c = n \times [K_s + \lceil log_2(g) \rceil]$, where n is the number of keys stored in CAMs and K_s denotes key size. Unlike GroupIDs in BT, GroupIDs in CAM1 and RAM2 are absolute and not relative to the segments. We define *load factor*, l as the ratio between the keys stored in the system (*i.e.,* in hash table and CAMs) at a given time and hash table capacity. Considering all these factors, the on-chip memory requirement per key can be given as,

$$\Omega = \frac{m \times M_b + g \times M_g + 2 \times M_c}{C \times l} \quad (1)$$

B. Time Complexity for On-chip Search

The main contributor to on-chip time complexity is the search times for the SSA and DSA. The group search used by both algorithms takes time proportional to G_s, since the search is limited to a single segment and $G_s - 1$ groups need to be searched. Note that for a hardware implementation, since G_s is small, the group search can be done in parallel using a simple priority encoder. For DSA, in addition to group search, a bin search is also done, which takes time proportional to s.

C. External Memory Access

The worst-case external memory access for different operations is as follows. As explained in Section III, the query and delete operations take at most one and two burst accesses, respectively, regardless of the algorithm used. Insert takes at most one burst access for NSA and 3 burst accesses for SSA and DSA. For SSA, first the g_t is read, the updated g_t from which the keys in b_t are deleted is written back, and, finally, the keys in b_t are written to the group g_{min}. DSA is similar to SSA, where the only difference is the bin moved is b_s rather than b_t. Note that inserting a key temporarily to the CAMs does not require any external memory access. However, an external memory access may occur later when and if this key is moved from the CAMs to the hash table. In our scheme, the key is moved to the hash table only if it does not cause an overflow. If there is an overflow, the key is kept in the CAMs without any external memory access so moving a key from the CAMs to the hash table always takes one burst access.

The expected number of external memory accesses for different operations is as follows. When a queried key or a key to be deleted is in the CAMs, no external memory access is needed for these keys. However, since the number of keys in the CAMs is negligible compared to the number of keys in the hash table, this does not have a significant impact. Thus, the expected number of external memory accesses for query and delete is similar to the worst-case, *i.e.,* one burst access for query and two burst accesses for delete.

For inserts using NSA, if the new key causes an overflow, no external memory access is needed for this key. Let N show the total number of keys inserted into the system. Then, the number of accesses required per key on average for NSA is $A_{avgA} = (N - n)/N$. This is because only the keys left in the CAMs at the end of a time period do not need external memory accesses in this time period. For keys temporarily stored in the CAMs, one burst access is required as with a normal insertion when they are finally inserted to the hash table.

For SSA and DSA, the number of accesses for insert may vary. Below is the analysis for bounds on the number of external memory accesses for insert with SSA and DSA. Both algorithms require three burst accesses for the same conditions,

thus we analyze them together. Let W show the number of overflows in a give time period. First we look at the problem when only inserts are allowed, *i.e.,* no deletes. The $N - W$ keys that did not cause an overflow require one burst access to be inserted into their groups in the hash table. Out of the W keys causing overflows, the n keys are stored in the CAMs without any external memory access. Since there is no delete and n out of the W overflows is in the CAMs, the rest is inserted into the hash table after a search. Thus, they require three burst accesses each. The total number of burst accesses required for the keys causing overflows is then $(W-n)\cdot 3$. So for the insert only case, for the N keys inserted, $(N-W)\cdot 1 + (W-n)\cdot 3$ burst accesses are required. Then, the number of burst accesses required per key on average for SSA and DSA is $A_{avgB} = 1 + (2W - 3n)/N$. Since, $2W \gg 3n$, $A_{avgB} = 1+2W/N$. When both insert and deletes are allowed, the calculation above is still valid as a loose upper bound.

V. Performance Study

This section presents the performance study of our co-processor architecture based on the three metrics introduced in the previous section. For the simulations, we assume a hash table that can accommodate $C = 65,536$ keys. First, the three performance metrics are compared for different G_s values for $G_{max} = 8$ DRAM banks, a load factor, $l = 0.8$, and memory ratio, $\gamma = 0.5$. Here, *memory ratio* is defined as the ratio of the number of bins in BT to C. For each G_s value, the simulations are repeated 100 times, each time with $1,048,576$ 32-bit keys and a random hash function. The simulations have two phases. In the first phase, we start with an empty system and keep inserting keys until the designated load is reached. After this point, the second phase starts where the load is kept constant by deleting the oldest key in the system every time a new key is inserted until all $1,048,576$ keys are inserted.

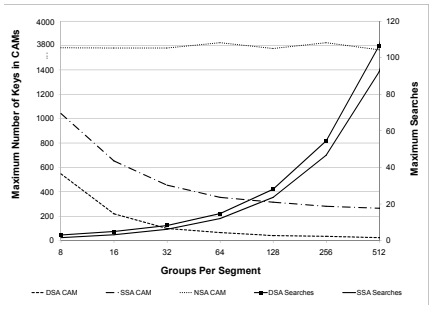

Fig. 4: Maximum number of keys in CAMs and maximum search time per insert operation.

Figure 4 shows the maximum number of keys that need to be stored in the CAMs (left y-axis) and the maximum number of total searches per insert operation (right y-axis) for the three algorithms over the 100 simulations. Since it does not employ any searches, the number of keys in the CAMs for the NSA is the maximum among the three algorithms at $3,765$ keys or 7.2% of the total number of keys in the system. However, the number of keys in the CAMs drops significantly if the SSA or DSA are used. As expected, for both SSA and DSA, the number of keys in the CAMs decreases with increased G_s (more groups per segment thus more chances to avoid writing to the CAMs) and the search time increases with increased G_s. In our simulations, the average external memory access is at most 1.5 bursts for NSA and 1.7 for the DSA per operation.

TABLE I: Performance comparison for NSA, SSA, and DSA, where GSI and BSI stands for group and bin search per insertion, respectively. AI is external memory access for insert.

Algorithm	G_s	n	$\Omega(bits)$	GSI	BSI	AI
NSAS	N/A	$3,765$ (7.2%)	7.09	0	0	0.997
SSA	32	455 (0.87%)	7.03	6.19	0	1.309
DSA	32	100 (0.19%)	6.42	6.03	2.21	1.376

Table I summarizes the performance of the three algorithms for the metrics described above. Since, there is no search in NSA, the BT can be eliminated for NSA, and the GCT can be addressed directly. To be fair to NSA, we do not consider the BT space when calculating Ω for NSA and called this simplified version of NSA as *NSAS*. According to the table, for DSA less than 0.2% of the keys are stored in the CAMs, this is a 97% reduction compared to NSA. Additionally, the total on-chip memory used for DSA, including the CAMs, is 6.42 bits per key, which is an 80% reduction compared to [10] [11] and a slight reduction compared to NSAS. The DSA requires 8.24 search steps on average, which can be easily achieved on-chip. The insert operation for DSA takes 1.376 burst accesses on average. Note that as discussed in Section IV, the average-case for query and delete operations are similar to their worst-case.

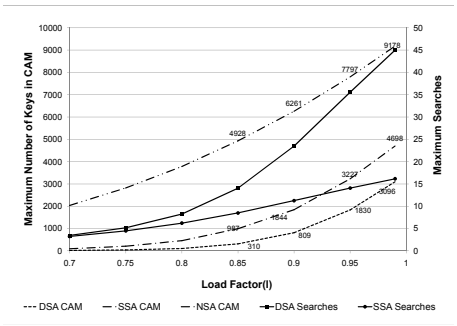

Fig. 5: Maximum number of keys in CAMs and maximum search time for different load factors.

In Figure 5, the CAM and search requirements are shown for different load factors. Even for the very high load factor of 99%, DSA requires a CAM space of 4.8%, total on-chip memory of 9.35 bits, 46 search steps, and 1.88 burst accesses for insert (on-chip storage and external memory accesses are not shown in the figure).

A. Distribution of Keys To Groups

The group/bin overflows are the main factors determining the CAM size, search time, and average external memory

access. Ideally, if there were no overflows, no CAMs or searches would be required and for insert, the number of external accesses would always be 1 burst. This effect of overflows is not unique to our scheme and all dynamic hash schemes need to deal with overflows one way or another. Thus, it is important to show how our proposed scheme impacts the overflows. In this section, we look at group occupancy distribution for different algorithms and discuss the impact of our scheme on overflows and other performance parameters.

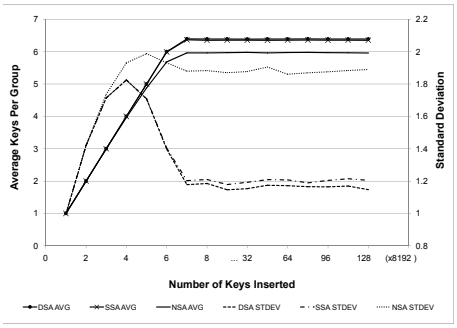

Fig. 6: Average and standard deviation of number of keys per group.

The insertion of a key overflows a group/bin only if that group/bin is already full. In SSA and DSA, we essentially move the load from full groups to lightly loaded groups, reducing the chance of overflow. Since SSA and DSA move the load every time there is an overflow, each overflow potentially avoids some of the future overflows. In Figure 6, average and standard deviation, σ, for the number of keys per group are shown during the simulation. At first, as the load factor increases for all the three algorithms, σ follows the average. However, as the load factor increases, since G_{max} is constant, the rate of increase in σ starts slowing down to around 50% load (at $4 \times 8K = 32K$). However, this reduction is much more significant for SSA and DSA as compared to NSA. As a result, for SSA and DSA, the group sizes are closer to the average and away from the G_{max}, with respect to NSA. Ideally it is preferred that all groups have an equal number of keys and provide a narrower distribution. SSA and DSA provide a better approximation to this ideal condition. Additionally, the difference between averages for NSA and the other two algorithms is because NSA has more keys in the CAMs, causing fewer keys in the hash table thus a lower average compared to SSA and DSA.

TABLE II: Overflows and number of keys moved to the CAMs.

	NSA	SSA	DSA
Overflows	$314,883$	$208,257$	$202,759$
To CAMs	$314,883$ (100%)	$47,089$ (22.6%)	$6,571$ (3.2%)

In Table II, the effect of the reduction in σ is shown, where the total number of overflows significantly decreases for SSA and DSA, with respect to NSA. Additionally, the percentage of overflows resolved by putting the new key in the CAMs is reduced 85% by using DSA, with respect to using SSA.

VI. Conclusion

This paper proposes a hash co-processor architecture to provide load-balanced hash tables to network applications. The proposed architecture allows moving keys from overloaded groups to lightly loaded groups, providing more balanced distribution of the keys in the hash table. Our architecture is designed to allow full bandwidth use of the DRAMs by providing fundamental hash table operations that follow the burst-access requirements of DRAMs. In the proposed scheme, the load balancing takes place only when a group becomes full. In fact, load balancing does not have to wait for a group to be full. It can also be done in an opportunistic manner whenever the link is not too busy. This may result in a further reduction of the average insertion time.

Acknowledgment

We would like to thank Najla Alfaraj for her comments.

References

[1] A. C. Snoeren, C. Partridge, L. A. Sanchez, C. E. Jones, F. Tchakountio, B. Schwartz, S. T. Kent, and W. T. Strayer, "Single-Packet IP Traceback," *IEEE/ACM Transactions on Networking*, vol. 10, no. 6, Dec. 2002.

[2] S. Dharmapurikar, P. Krishnamurthy, and D. E. Taylor, "Longest Prefix Matching Using Bloom Filters," in *SIGCOMM '03: Proceedings of the 2003 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*. New York, NY, USA: ACM, 2003, pp. 201–212.

[3] S. Dharmapurikar and V. Paxson, "Robust TCP stream reassembly in the presence of adversaries," in *SSYM'05: Proceedings of the 14th Conference on USENIX Security Symposium*, 2005, pp. 5–5.

[4] D. Schuehler, J. Moscola, and J. Lockwood, "Architecture for a Hardware Based, TCP/IP Content Scanning System," in *Symposium on High Performance Interconnects (HotI)*, Stanford, CA, Aug. 2003, pp. 89–94.

[5] S. Singh, C. Estan, G. Varghese, and S. Savage, "Automated Worm Fingerprinting," in *Proc. of the ACM/USENIX Symposium on Operating System Design and Implementation*, San Francisco, CA, Dec. 2004, pp. 45–60.

[6] B. Madhusudan and J. Lockwood, "Design of a System for Real-Time Worm Detection," in *Proc. of 12th Annual IEEE Hot Interconnects (HotI-12)*, Stanford, CA, Aug. 2004, pp. 77–83.

[7] B. Agrawal and T. Sherwood, "Virtually Pipelined Network Memory," in *MICRO 39: Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*. Washington, DC, USA: IEEE Computer Society, 2006, pp. 197–207.

[8] R. Pagh and F. F. Rodler, "Cuckoo hashing," in *ESA*, 2001, pp. 121–133.

[9] A. Kirsch and M. Mitzenmacher, "The power of one move: Hashing schemes for hardware," in *27th Annual IEEE Conference on Computer Communications (INFOCOM)*, Phoenix, AZ, USA, Apr. 2008, to appear.

[10] H. Song, S. Dharmapurikar, J. Turner, and J. Lockwood, "Fast Hash Table Lookup Using Extended Bloom Filter: an Aid to Network Processing," in *SIGCOMM '05: Proceedings of the 2005 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, 2005, pp. 181–192.

[11] S. Kumar and P. Crowley, "Segmented hash: an efficient hash table implementation for high performance networking subsystems," in *Proceedings of the 2005 symposium on Architecture for networking and communications systems (ANCS)*, 2005, pp. 91–103.

[12] B. Bloom, "Space/Time Trade-offs in Hash Coding with Allowable Errors," *Communications of the ACM*, vol. 13, no. 7, 1970.

[13] F. Bonomi, M. Mitzenmacher, R. Panigrahy, S. Sing, and G. Varghese, "An Improved Construction for Counting Bloom Filters," in *14th Annual European Symposium on Algorithms, ESA 2006*, ETH Zrich, Zrich, Switzerland, Sep. 2006.