Fast Lookup for Dynamic Packet Filtering in FPGA

Lukáš Kekely, Martin Žádník, Jiří Matoušek, Jan Kořenek
IT4Innovations Centre of Excellence
Brno University of Technology, Czech Republic
Email: ikekely,izadnik,imatousek,korenek@fit.vutbr.cz

***Abstract*—Rapidly growing speed and complexity of computer networks impose new requirements on fast lookup structures which are utilized in many networking applications (SDN, firewalls, NATs, etc.). We propose a novel lookup concept based on the well-known cuckoo hashing, which can achieve good memory utilization, supplemented by a binary search tree for offloading the colliding keys and supporting LPM lookup. We also propose a hardware architecture implementing this lookup concept in the FPGA. Our solution is suitable for lookup of the variable-length keys in 100+ Gbps networks. Memory utilization of the proposed concept is thoroughly evaluated and it is shown that the concept is scalable to external memory components.**

Keywords*—*Cuckoo hash; binary search; packet filtering; FPGA

I. Introduction

Field Programmable Gate Arrays (FPGA) are popular platforms utilized in networking applications targeting high-speed packet processing (e.g. [1]). We propose a fast lookup concept designed specifically for FPGA-oriented platforms. The concept combines two well-known memory-oriented lookup algorithms – cuckoo hashing [2] and binary search tree (adapted for best/longest prefix matching [3]). Each algorithm efficiently complements the other in area where the other fails. The concept achieves almost 100% memory utilization with efficient utilization of the memory and logic resources in comparison to the TCAM or Hash-CAM concepts [4]. At the same time, our concept allows fast lookups (200 mil. lookups/s designed for 100+ Gbps solutions). Our contributions also include: (a) the possibility to utilize external memory when the number of rules cannot fit in the internal FPGA memory, (b) increasing the lookup functionality with the longest prefix match, (c) efficient implementation of the whole scheme in FPGA including the update logic enabling on-the-fly updates and (d) evaluation of the concept in terms of achievable resources utilization.

II. Related Work

The goal of cuckoo hashing [2] is to reduce the number of memory accesses during a lookup and thus speeding-up a lookup operation. Standard cuckoo hashing utilizes two hash tables with two different hash functions but it can be generalized for a higher number of hash tables/functions. There has been an implementation of cuckoo hashing in FPGA for the purpose of pattern matching [4]. This architecture contains dedicated matching blocks for all patterns of the same length (up to the length of 16 characters). Each matching block consists of two cuckoo hash tables for storing addresses to the database of patterns. The architecture also contains multiplexers and a control logic which together allow performing either a pattern matching operation or a pattern database update (pattern insertion or deletion). The approach offers only medium memory utilization since it does not utilize any type of overflow memory and also cannot scale well to external memory since it is tailored to the internal FPGA memory.

The advanced lookup procedures also include prefix matching (PM, i.e. there is a single prefix for a given key in the set but it is not known apriori) and longest prefix match (LPM, i.e. selecting the longest matching prefix from the set for a given key). Although the LPM itself is out of the primary scope in this paper, the unique combination of cuckoo hash and binary search tree renders it possible for our implementation to support LPM lookup.

III. Design and Architecture

The core functionality of our lookup schema is based on cuckoo hashing principle due to a very fast lookup with only a few memory accesses needed for each search. This feature favors the usage of cuckoo hashing even on architectures with limited memory interface throughput (e.g. external memory). On the other hand, cuckoo hashing can suffer from a low achievable utilization of the memory caused by hash conflicts. To address this problem, our design augments basic cuckoo hashing principle by the usage of a stash for offloading the conflicting keys. The proposed design of cuckoo hashing with the stash is not entirely new. It has been already described in [5], where the authors proposed and evaluated the usage of only a very small stash (capacity under 5 keys implemented in TCAM) to improve the worst case memory utilization of cuckoo hashing.

In our design we propose and evaluate the usage of a significantly larger stash – a stash with the capacity comparable to the capacity of the used cuckoo hash tables to improve not only the worst case but also to improve average memory utilization. Furthermore, our stash also supports LPM lookups, thus augmenting the lookup functionality of the basic cuckoo hashing. The lookup support of not only the whole keys but also key prefixes can be very useful in many different areas (e.g. packet filtering). Instead of TCAM, we propose an FPGA implementation of a well-known binary search algorithm adapted for the LPM lookup (described in [3]) as an effective approach to implement the larger stash.

The binary search offers basically the opposite features in comparison with the cuckoo hashing – the key lookup requires relatively large number of subsequent memory accesses, but the achievable memory utilization is always 100 %. Because of the large number of memory accesses, binary search based lookup should be implemented only in the internal FPGA memory. In order to achieve high lookup throughput, the implementation of the binary search must not be sequential but rather divided into pipelined stages. This can be achieved by establishing a tree structure in the searched array (binary search tree – BST) and slicing it by the tree levels (each tree level forms a pipeline stage). Finally, the functionality of update operations in the described BST can be easily implemented in the hardware with support of on-the-fly updates.

A. *Lookup engine interface and functionality*

We start the description by the general design of an interface and functionality of a virtual lookup engine (either cuckoo or BST). Both engines implement the same interface independently on the details of their lookup procedure. The signals can be divided into 3 basic groups: input, output and configuration. The only input of a lookup engine is the value of a key to search. The lookup implementation should be able to process new input key every clock cycle. For each input key, the engine produces one result on the output based on performed lookup. The lookup result consists of arbitrary data (e.g. routing decision, matched key identification) associated with the searched key and one bit information about the key lookup success (Found). When the input key is not found, the value of data on the output is unspecified (invalid).

The lookup engine (and its interface) is configurable by these three basic generic parameters: **key width** (maximum width of key representation in bits), **data width** (width of data representation in bits), **maximum capacity** (theoretical size limit for the set of keys, representation may differ).

B. *Cuckoo hash lookup engine*

Fig. 1 depicts a basic schema of cuckoo hash engine implementation. The lookup process starts by parallel computing of key hash values (outputs of hash blocks). As the basis for the hash blocks we utilize CRC implementation generated for commonly used polynomials. The lookup continues with hash values being used as addresses for reading records from hash tables in memory. Each record forms a pair composed of a key and data associated with the key. A record can also be stored in a register outside the tables (the purpose of the register is explained in the next paragraph). Subsequently, the input key is compared with the keys from the memory (and the register) records for equality. At most one comparison may be successful, because each unique key appears only in a single place at a time. Therefore, aggregation of result is very simple – if none of the compared keys is equal to the searched key, the found flag is not set, otherwise it is set and data associated with the matching key are provided.

Update of an active key set is entirely managed by the reconfiguration controller based on requests received from the configuration interface. When inserting a new key, the controller can take advantage of the reconfiguration register included in the lookup path. Using this register the controller can evict records from hash tables on-the-fly preserving the set of active keys. More precisely, the insertion of a new key x starts with storing x in the register. Then all possible locations for x in the hash tables are checked sequentially. If one of them is empty, x is inserted into the table and the reconfiguration ends. Otherwise a victim y is selected and evicted from the table, leaving free space for x. The evicted record y is actually

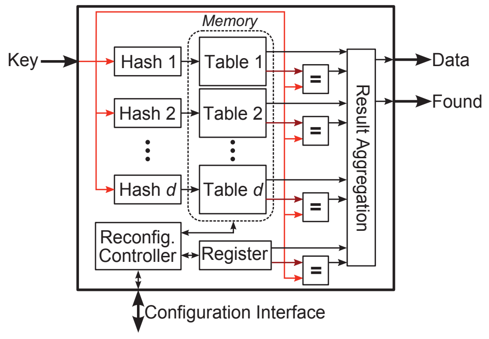

Fig. 1. Conceptual schema of cuckoo hash based lookup engine.

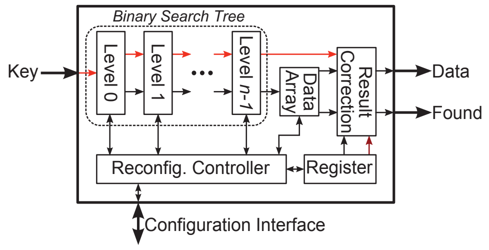

Fig. 2. Conceptual schema of binary search tree based lookup engine.

swapped with x and the insertion continues with y except x cannot be selected as the next victim. The reconfiguration cycle can repeat itself multiple times, until the register is freed or can even repeat itself infinite times when a chain of collisions occurs. Until the register is freed the cuckoo hash engine is considered full. Deletion of a key is possible even during active insertion reconfiguration. Deletion of x starts by pausing the reconfiguration process and continues with sequential checking of all possible locations for x (i.e. the register and a single position in each table). If a key identical to x is found in one of those positions, it is invalidated. After the deletion ends, the reconfiguration process is resumed.

The maximum capacity of the cuckoo hash engine can be configured by two values: d – the number of used hash tables (hash functions) and t – the size of individual table. Theoretical capacity limit is defined by formula $C_{cuckoo} = d \times t + 1$. The plus one accounts for the additional reconfiguration register.

C. *Binary search tree lookup engine*

Fig. 2 depicts a basic schema of our BST lookup engine. The engine starts the lookup by a pipelined and sequential search of an input key (red arrows) through the levels of the tree. Each tree level forms a pipeline stage with its dedicated piece of memory and a key comparator. The output of a stage is an address of a node where to continue binary search in the next tree level and the searched key. The address from the last tree level is used to address the data array containing associated data to the key. The lookup result must be corrected according to a record in the reconfiguration register due to atomicity of operations.

Update of an active key set is entirely managed by the reconfiguration controller based on requests received from the configuration interface. The controller can take advantage of the single reconfiguration register included in the lookup path during the update. More precisely, the update (deletion or insertion of x) starts with storing the record x in the register. Subsequently, the update process consists of three sequential steps. (1) The key x is searched in the tree sequentially. The search must fail when inserting x. The search must succeed when deleting x. (2) The record x is activated in the register to correct the lookup process in the last stage. (3) Sequential reconfiguration is performed to merge x into the nodes and the data array. Finally, the update process ends and the reconfiguration register is freed. Deletion and insertion share resources and cannot be active together as in cuckoo hash engine. The engine can become full only after successful insertion and can become empty again only after successful deletion.

The capacity of the BST based engine can be configured by the number of BST levels l. The capacity is then defined by formula $C_{bst} = 2^l - 1$ when adaptation for LPM is not used or $C_{bst} = 2^{l-1} - 1$ when LPM lookup is supported. Our implementation supports the adaptation for LPM, but if LPM is not needed, it can be easily modified (simplified) to support only precise key lookup gaining two times higher capacity.

D. Top-level lookup engine

Top-level engine instantiates both Cuckoo and BST engine in parallel. The lookup of an input key is also performed in parallel in both engines. The results are then stored in FIFOs, because the two engines do not have same processing delays. Result aggregation then selects data from engine with successful lookup. When both engines successfully find a key, the result from cuckoo hash is preferred, because in that case the result from BST is only for a matching prefix, but the result from cuckoo hash is for the whole matching key.

Reconfiguration of the key sets in both engines is managed by the top level reconfiguration controller. All updates for prefixes are directly forwarded into the BST stash. Deletions of the whole keys are implemented in both engines in parallel. Insertions of the whole keys are forwarded into the cuckoo hash. If cuckoo hash is full (its reconfiguration register is occupied) and new key insertion is requested, then the key that is currently in cuckoo hash reconfiguration register is moved into the stash and the new key is inserted into cuckoo hash. The top-level engine is full when both the cuckoo hash and the BST stash are full. Furthermore, in our implementation the configuration interface of the top-level lookup engine is connected to the block with address decoder and registers for key, data, requests and status flags. This block is then accessible from the software using standard memory interface. This way the management of the active key set can be easily controlled from the software.

The maximum capacity of the cuckoo hash with stash lookup engine can be defined by three parameters: parameters d and t of the cuckoo hash and the stash size s. Theoretical capacity limit is then defined by formula $C_{total} = d \times t + 1 + s$.

IV. Evaluation and results

The proposed architecture was implemented in VHDL and synthesized into FPGA. We conducted experiments to evaluate achievable memory utilization and FPGA resources consumption in different configurations of the architecture. The results of these evaluations are summed up in this section.

We start the evaluation by experiments on achievable memory utilization of our concept. The achieved utilization can be computed in two basic ways: $U_{cuckoo} = (n - m)/C_{cuckoo}$, $U_{total} = n/C_{total}$, where n is the total number of successfully inserted keys before the memory became full and m is the number of keys that resides in the stash. Because, our implementation uses stash which can be always filled up to 100 % of its capacity, we can always put $m = s$. The values of n must be acquired from the test runs.

In the first series of tests we have evaluated the relation between achievable memory utilization of cuckoo hash and the used sizes of stash for different parameters. The results of these evaluations are shown in the graphs in Fig. 3 and 4. We have tested three different values of d parameter (2, 3 and 4 not depicted in the figures), three different values of t parameter (128, 1 024 and 8 192) and multiple values of s (from 0 to t). We have also tested different key sizes (32 b, 64 b and 128 b), but the achieved results have been very similar, therefore we do not show different graphs for each key size. The memory utilization plotted in the graphs is U_{cuckoo} and the size of the stash (s) is plotted as a portion of t. The graphs show mean (thick darker lines) and minimal resp. maximal (thin lighter lines) achieved utilizations from 10 000 tests with random generated keys for each combination of values of d, t and s. From data plotted in the graphs it is clear that the mean achieved memory utilization of cuckoo hash is independent on the values of t. Parameter t only influences the difference between minimal and maximal achieved utilization, when the span is higher for smaller values of t.

Moreover, Fig. 3 shows that the influence of stash size on the achievable memory utilization is significant for two cuckoo hash tables – the mean utilization raises from 50 % in the case without the stash to 75 % with $s = t/10$ or even around 90 % for $s > t/2$. Also the differences between minimal and maximal achieved utilizations are reduced with the raising size of stash. Fig. 4 shows that the importance of stash in case of more than two cuckoo hash tables is not that high as for two tables. But it contributes in achieving nearly 100 % mean memory utilization of cuckoo hash tables.

The second series of memory utilization tests is oriented on a thorough examination of achievable memory utilizations for a few selected sizes of stash. The results of these evaluations are shown in the graphs in Fig. 5 and 6. Here we have also tested three different values of d parameter (2, 3 and 4 not depicted), but only a single value of $t = 1\,024$ and only a few values of s (0, $t/64$, $t/16$, $t/4$, $t/2$ and t). The graphs show histograms of probability (percentage of all conducted tests) that achieved precisely the specified utilization (U_{cuckoo} used) with highlighted mean (dashed line) and minimal resp. maximal utilizations (points). The area under each histogram line is exactly 100 % even though the individual values are rather small. The results are from 1 000 000 tests with random generated keys for each combination of values of d and s. From data plotted in the graphs it is clear that the dispersion of achieved utilizations is lower for the rising stash size. Also the effect of stashes with size $s < t/16$ for the cuckoo hash with $d > 2$ is negligible.

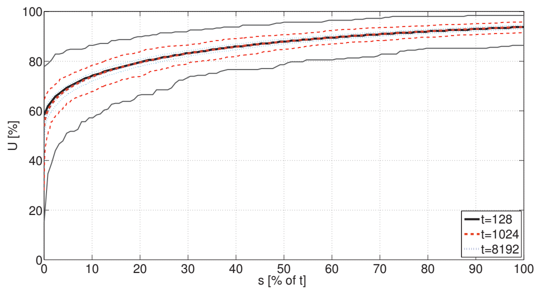

Fig. 3. Achievable memory utilization for cuckoo hash with two tables ($d = 2$) for different sizes of stash.

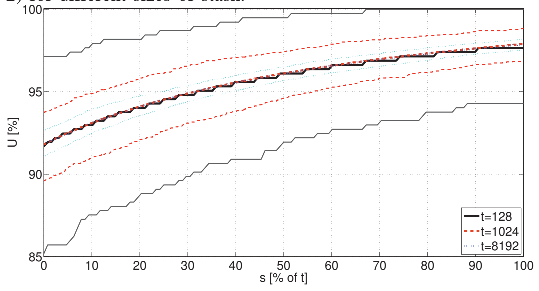

Fig. 4. Achievable memory utilization for cuckoo hash with three tables ($d = 3$) for different sizes of stash.

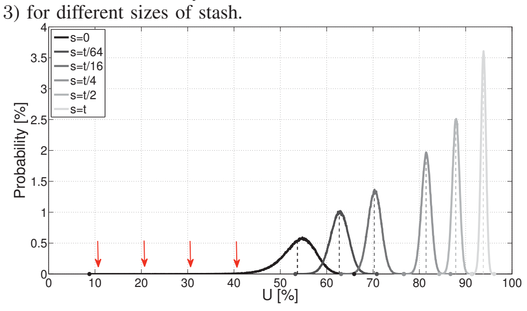

Fig. 5. Probability distribution of achievable memory utilization for cuckoo hash with two tables ($d = 2$, $t = 1\,024$).

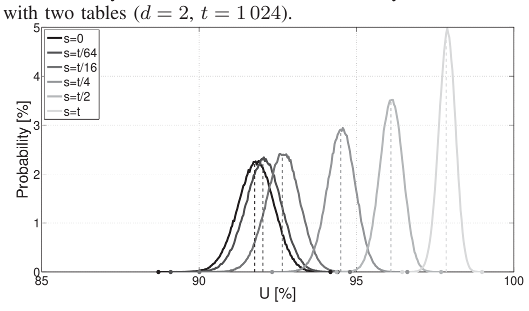

Fig. 6. Probability distribution of achievable memory utilization for cuckoo hash with three tables ($d = 3$, $t = 1\,024$).

The dispersion reduction is noticeable especially for the cuckoo hash with two tables (Fig. 5). For two tables without a stash there is a very real chance of achieving memory utilization that is significantly lower than the mean utilization (marked by red arrows). The solution to this problem is even a relatively small stash ($s = t/64$ or $s = t/16$). This particular situation is very important when cuckoo hash is implemented using large external memory to store cuckoo hash tables. The bottleneck in such an implementation lays in the throughput of external memory interface, which limits the number of usable cuckoo hash tables usually to only 2. These results suggests that stash of size $s = t/64$ or $s = t/16$ can significantly improve the achievable memory utilizations in exactly this case. So for example, the implementation of cuckoo hash with $d = 2$ and $t = 2^{20}$ in external memory require stash with size only $s = 2^{20}/16 = 65\,536$ to achieve mean external memory utilization of 70 % (mean capacity over 1.5 million keys) with very low chance to achieve utilization under 65 %.

TABLE I. FPGA RESOURCES REQUIREMENTS AND MEMORY UTILIZATIONS OF OUR LOOKUP ENGINE IMPLEMENTATION.

Key				FPGA Resources			Frequency	Mean	Mean
Width	d	t	s	LUTs	FFs	BRAMs	[MHz]	Utilization	Capacity
32	2	8 192	2 047	3 721	2 111	45	264.116	**83.5 %**	**15 388**
32	3	8 192	4 095	4 138	2 221	71	265.437	**96.7 %**	**27 711**
128	2	1 024	255	8 336	4 059	15	257.631	**83.5 %**	**1 923**
128	3	1 024	511	9 564	4 304	23	263.704	**96.7 %**	**3 463**

Finally, we present the FPGA resources requirements of our top-level lookup engine in selected configurations. The requirements in terms of LUTs, registers and BlockRAMs are given in Tab. I together with the achievable clock frequencies. Values in tables are acquired from the synthesis by XST tool for the XilinxVirtex-7 870HT FPGA and data width of 32 bits. Variable key widths (32 and 128 bits as lengths of IPv4 and IPv6 addresses were selected) and capacity parameters d, t, s are given in the table. Tab. I also presents mean achievable memory utilization (U_{total}) and capacity based on test results presented earlier in this section. The achieved frequencies over 200 MHz and the fact that each lookup implementation is capable of one lookup on each clock cycle suggest, that our architecture is capable of over 200 million lookups per second, which is sufficient for packet filtering on 100+ Gbps networks.

V. Conclusion

The paper proposed a viable concept for fast packet filtering for FPGA. The proposed architecture leverages the combination of the cuckoo hash engine with BST engine with a focus on parallel implementation in FPGA. The results of evaluation show that the concept allows not only fast lookups for every arriving packet on the 100+ Gbps links but also effective utilization of FPGA resources.

Acknowledgement

This research has been supported by the grant MVCR VG20102015022, BUT project FIT-S-14-2297 and the IT4-Innovations Centre of Excellence CZ.1.05/1.1.00/02.0070.

References

[1] J. Naous and et al., "Implementing an openflow switch on the netfpga platform," in *Proceedings of ANCS*, NY, USA, 2008, pp. 1–9.

[2] R. Pagh and F. F. Rodler, "Cuckoo hashing," *J. Algorithms*, vol. 51, no. 2, pp. 122–144, May 2004.

[3] B. Lampson, V. Srinivasan, and G. Varghese, "Ip lookups using multiway and multicolumn search," in *INFOCOM*, 1998, pp. 1248–1256.

[4] T. Tran and S. Kittitornkun, "Fpga-based cuckoo hashing for pattern matching in nids/nips," in *MNGNS*, ser. LNCS, 2007.

[5] A. Kirsch, M. Mitzenmacher, and U. Wieder, "More robust hashing: Cuckoo hashing with a stash," in *ESA*, ser. LNCS. Springer, 2008.