Hardware accelerated packet filter for 100Gbps

User documentation

Lukáš Kekely, Martin Žádník

Technical documentation FIT-TR-2014-XXX

Hardware packet filter for 100Gbps

Lukáš Kekely, Martin Žádník

Fakulta informačních technologií
Vysoké učení technické v Brně
Božetěchova 1/2, 612 66 Brno
{xkekel00, izadnik}@fit.vutbr.cz

Abstrakt 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.). A lot of hash or hash-CAM based lookup concepts have been proposed but they usually suffer from the low memory utilization or insufficient lookup performance. Therefore, 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 carefully evaluated and it is shown that the concept is scalable to external memory components.

1 Introduction

The speed and complexity of network is growing rapidly, creating a demand for new approaches to a high-speed packet processing. One major trend is to make hardware as simple as possible offloading the complexity of a control path into the software, e.g. software-defined networking (SDN) [5]. The offload requires the hardware to look up a piece of data (e.g. action, state) associated to a flow key (e.g. IP prefix, IP address or tuple of source and destination IP addresses, ports and protocol) per each arriving packet. Moreover the associated data are created dynamically, usually with every new flow. The fast lookup is essential not only for SDN applications but also in other existing applications (e.g. NATs, firewalls, load-balancers and network probes [1]).

Field Programmable Gate Arrays (FPGA) are popular platforms utilized in networking applications targeting high-speed packet processing (e.g. [4]). Traditionally, the lookup was performed by external Ternary Content-Addressable Memory (TCAM) or internal TCAM implemented in FPGA logic. Various memory-oriented approaches have been proposed and applied, replacing the resource as well as power demanding TCAM. A common drawback of the memory-oriented approach is inefficient memory utilization and slow lookup requiring multiple memory accesses.

We propose a fast lookup concept designed specifically for FPGA-oriented platforms. The concept combines two well-known memory-oriented lookup algorithms – cuckoo hashing [6] 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 [9]. 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 memory and logic utilization.

The rest of the paper is organized as follows. Related work on lookup algorithms is discussed in Section 2. Section 3 provides design objectives and introduces the proposed concept. The FPGA implementation is described in Section 4. Section 5 evaluates the concept as well as provides the synthesis results. The paper is concluded with future work in Section 6.

2 Related Work

A naive approach to the lookup is represented by a linear search.Unfortunately, the linear search requires too many memory accesses and, therefore, is too slow to satisfy the needs for the fast lookup. Various approaches have been proposed to speed up the lookup up to the point of a constant time and low number of memory accesses. These approaches take advantage of the particular specification of the lookup task, key structure, key set characteristics, target platform and others.

The basic lookup tests whether an element (a key) is a member of a set. If the application can tolerate small percentage of false positive results (the key is considered to be part of the set although it is not) then Bloom filter and its improvements [7] have been shown to achieve near optimal memory utilization with low number of memory accesses.

But in many cases the application requires to assign some data per each key. Here, TCAM has traditionally been used, but due to its high cost, high resource as well as power consumption, it is often replaced by memory-oriented approaches. The basic approach is to utilize naive hash table (NHT). The hashed key is used as an index into a memory location with the key (or its fingerprint to verify successful lookup) and the associated data. If a new key is inserted and the location is already occupied, a new memory location is allocated and linked to the occupied location, forming a list of locations. A hardware implementation of the NHT divides the memory into an equally-sized buckets (each holding N possible locations). The issue of such an approach is that it requires multiple (N) accesses to the memory and some of the buckets overflow whereas some may not be utilized at all. An exemplary hardware implementation of NHT approach using hash-CAM is described in [10]. If more than N colliding keys occur, an additional TCAM is used to store the overflowing keys. An improved approach also utilizes another hash function in order to increase hash table utilization and decrease the required size of the additional TCAM (each key can be stored in $2N$ locations). The main drawback of this approach is that this approach can only be effectively implemented using internal FPGA memory since the $2N$ hash tables are implemented using parallel memories. Also the utilization increases with lower value of N but the size of the costly TCAM must then be increased in order to store all the colliding keys. Song et al. [8] extended the NHT approach with counting Bloom filters to optimize utilization of the buckets by counting their occupancy, but the issue of multiple memory accesses remained.

The goal of cuckoo hashing [6] 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 (hash functions). Standard cuckoo hashing with two hash tables T_1, T_2 and two hash functions h_1, h_2 stores a key x either in position $T_1[h_1(x)]$ or $T_2[h_2(x)]$, but never in both. Therefore, the lookup as well as the deletion of a key can be easily performed in constant time. Insertion of a key x starts by checking the positions $T_1[h_1(x)]$ and $T_2[h_2(x)]$. If at least one of them is free, the key is inserted. But if none of them is free, the key x is still inserted to the position $T_1[h_1(x)]$, thus evicting the stored key y. Subsequently, the algorithm inserts the key y to the position $T_2[h_2(y)]$. If $T_2[h_2(y)]$ is not free, the stored key z is evicted. Note that it is sufficient to access only the position $T_2[h_2(y)]$ and not $T_1[h_1(y)]$ since $T_1[h_1(y)]$ was the previous position of y and is now occupied by x. These steps are repeated until the free position is found or a pre-defined number of steps is reached. If the insertion procedure was not successful, new hash functions h_1 and h_2 have to be selected and complete rehashing takes place (unacceptable in online applications). Cuckoo hashing guarantees constant lookup time but the utilization of memory reaches only about 50 % for two hash tables and two hash functions on average.

There has been an implementation of cuckoo hashing in FPGA for the purpose of pattern matching [9]. 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 single-port cuckoo hash tables for storing addresses to the dual-port memory storing the database of patterns. The architecture also contains two multiplexers and a control logic which together allow performing either a pattern matching operation or a pattern database update (insertion or deletion of a pattern). 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 properties of FPGA internal 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.

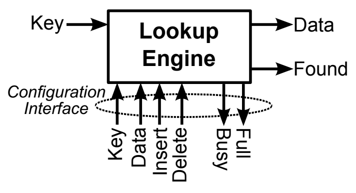
3 Design

The core functionality of our lookup schema is based on cuckoo hashing principle described in the previous section. We selected this principle because it has an advantage of 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). Also, the functionality of update operations (insert/delete key) used in cuckoo hashing is rather simple and can be effectively implemented in hardware enabling on-the-fly updates.

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 [2], 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 key idea behind the adaptation of the binary search for LPM is to recognize each prefix p as a range of keys (values) from k^p_{min} to k^p_{max}, where k^p_{min} is the lowest key with prefix p and k^p_{max} is the highest key with prefix p. Now for each prefix p in the lookup set both k^p_{min} and k^p_{max} values are stored in a sorted array, k^p_{min} is associated with data of prefix p and k^p_{max} is associated with data of the longest prefix from the set which is a sub-prefix of p. Furthermore, if for two prefixes p_1, p_2, where p_1 is shorter than p_2, keys $k^{p_1}_{min} = k^{p_2}_{min}$ resp. $k^{p_1}_{max} = k^{p_2}_{max}$, the relation between keys is considered to be $k^{p_1}_{min} < k^{p_2}_{min}$ resp. $k^{p_1}_{max} > k^{p_2}_{max}$ during sorting. Also, searched key k such as $k = k^p_{min}$ resp. $k = k^p_{max}$ is considered $k > k^p_{min}$ resp. $k < k^p_{max}$. A lookup of k is then implemented as a standard binary search and the data of the last visited smaller key than k are used as a result. From the adaptation description it is clear, that each prefix occupies two records in the memory.

The binary search offers basically the opposite features in comparison with the cuckoo hashing – the key lookup requires relatively large number of sub-



Obrázek 1. General lookup engine interface (some signals are omitted for clarity).

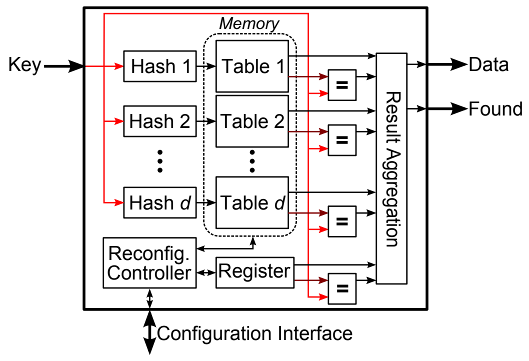
sequent memory accesses, but the achievable memory utilization is always 100 %. Because of the large number of memory accesses, binary search based lookup should be implemented only in the internal FPGA memory. In order to achieve high lookup throughput, the implementation of the binary search must not 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.

4 FPGA architecture

4.1 Lookup engine interface and functionality

We start the description by the general design of an interface and functionality of a lookup engine (either cuckoo or BST). Both engines implement the same interface independently on the details of their lookup procedure. Fig. 1 illustrates particular signals forming the interface. The signals can be divided into 3 basic groups: input (left), output (right) and configuration (bottom). 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 configuration signals can be further divided into two subgroups: update requests and status flags. The update requests are used to manage an active key set and the associated data in the engine. The keys can be inserted or deleted one by one. When insertion of a new key is requested, the value of the key and the associated data must be both specified. If the key is already in the set, the insertion fails and the set remains unchained (same keys would quickly build up a list of chained records). When deletion of a key is requested, only the value of the key itself needs to be specified (the associated data are ignored). Furthermore, when the lookup engine supports LPM, the key value in the update request must be accompanied by the length of a valid prefix. The status flags



Obrázek 2. Conceptual schema of cuckoo hash based lookup engine.

inform about the special states of lookup engine configuration logic. An active busy flag means that an update of a key set is running and new updates cannot be requested until the update finishes. The flag is activated by the engine right after each valid request and remains active for finite time. A full flag is activated when the supported lookup capacity is reached and new key insertions cannot be requested. Delete requests are still possible even when the full flag is active and they can lead to its deactivation. When no deletes are issued, the full flag can remain active for infinite time or is deactivated if the conflicts are resolved eventually.

Finally, 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).

4.2 Cuckoo hash lookup engine

Fig. 2 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.

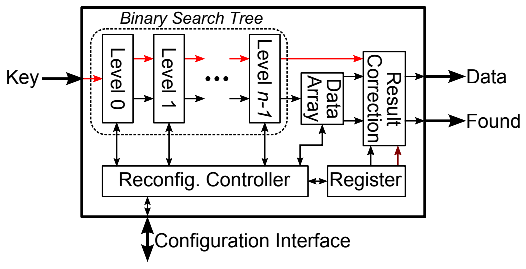
Update of an active key set is entirely managed by the reconfiguration controller based on requests received from the configuration interface. When inserting a new key, the controller can take advantage of the reconfiguration register included in the lookup path. Using this register the controller can 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 ($T_i[h_i(x)]$) 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 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 occur. Until the register is freed the cuckoo hash engine is considered full, but not busy. Therefore, 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 all $T_i[h_i(x)]$ since our implementation supports arbitrary number of hash tables). 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. During the deletion process, the cuckoo hash engine is considered busy.

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.

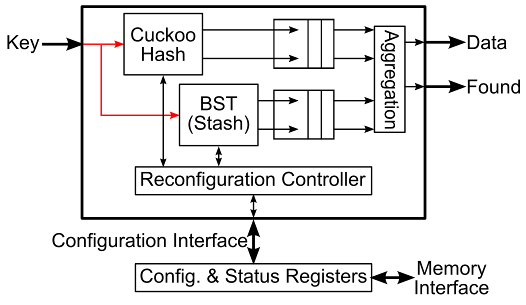
4.3 Binary search tree lookup engine

Fig. 3 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 of x 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. During the update process, the BST engine is considered busy. Therefore, deletion and insertion cannot be active together as



Obrázek 3. Conceptual schema of binary search tree based lookup engine.



Obrázek 4. Conceptual schema of our top-level lookup engine.

in cuckoo hash engine. The full status may be issued after the successful insertion and can only end 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.

4.4 Top-level lookup engine

Fig. 4 depicts a top-level schema of our cuckoo hash with BST stash lookup engine implementation. The lookup of an input key (red arrows) is implemented in both cuckoo hash and BST engines in parallel. 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 reaches full state (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 in the full state when both the cuckoo hash and the BST stash are full. The top engine is in busy state when at least one of them is busy. 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 $C_{total} = d \times t + 1 + s$.

5 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. 5, 6 and 7. We have tested three different values of d parameter (2, 3 and 4), 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. 5 shows that the influence of stash size to 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. 6 and 7 show that the importance of stash in case of more than two cuckoo hash tables is not that high as for two tables. But as you can see, it can help to achieve nearly 100 % mean memory utilization of cuckoo hash tables.

The second series of memory utilization tests is oriented to more precise examination of achievable memory utilizations for a few selected sizes of stash. The results of these evaluations are shown in the graphs in Fig. 8, 9 and 10. Here we have also tested three different values of d parameter (2, 3 and 4), 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.

The dispersion reduction is noticeable especially for the cuckoo hash with two tables (Fig. 8). 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 %.

Finally, we present the FPGA resources requirements of our lookup engines in selected configurations. The requirements in terms of LUTs, registers and BlockRAMs are shown in Tables: 1 for cuckoo hash based lookup engine alone, 2 for binary search tree based stash alone and 3 for the top-level lookup engine architecture. Also the maximal clock frequencies are shown. 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 are shown directly in the tables. Furthermore, Table 3 have columns with mean achievable memory utilization (U_{total} is used) and capacity based on test results presented earlier in this section. Also, achieved frequencies over 200 MHz and the fact that each

Key			FPGA Resources			Frequency
Width	d	t	LUTs	FFs	BRAMs	[MHz]
32	2	1 024	592	186	3	284.715
32	2	8 192	630	195	25	277.313
32	3	1 024	827	223	5	265.600
32	3	8 192	863	235	38	287.115
32	4	1 024	1 053	260	6	277.795
32	4	8 192	1 138	275	50	235.238
128	2	1 024	2 075	322	9	264.886
128	2	8 192	2 190	331	73	265.799
128	3	1 024	2 798	367	14	261.985
128	3	8 192	2 927	379	110	262.953
128	4	1 024	3 500	412	18	263.576
128	4	8 192	3 655	427	146	263.902

Tabulka 1. FPGA resources requirements of our cuckoo hash implementation in selected configurations.

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.

6 Conclusion

The paper proposed a new lookup concept based on the well-known cuckoo hash augmented by the adapted binary search tree for offloading the colliding keys and supporting LPM lookup. The proposed architecture elaborated the combination of the cuckoo hash engine with BST engine with a focus on parallel implementation in FPGA. The concept was evaluated in terms of achievable memory utilization as well as utilization of memory and logic resources of the FPGA. The results show that the concept is feasible allowing not only fast lookups for every arriving packet on the 100+ Gbps links but also effective utilization of FPGA resources.

Our future work will test the concept in use cases of packet filtering in legal interception probe and as a flow cache lookup procedure in a software defined monitoring probe [1]. In both these use cases the LPM support is vital as the prefixes would have to be represented by multiple exact match rules.

Reference

1. Kekely, L.; Pus, V.; Korenek, J.: Software Defined Monitoring of Application Protocols. In *INFOCOM*, To be published, 2014.
2. Kirsch, A.; Mitzenmacher, M.; Wieder, U.: More Robust Hashing: Cuckoo Hashing with a Stash. In *ESA*, LNCS, Springer, 2008, ISBN 978-3-540-87743-1.

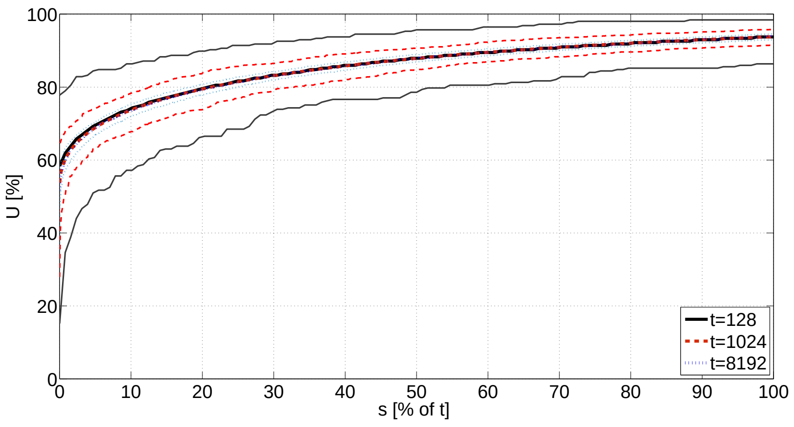
Key		FPGA Resources			Frequency
Width	Capacity	LUTs	FFs	BRAMs	[MHz]
32	255	1 618	1 024	3	323.236
32	511	1 744	1 117	5	319.472
32	1 023	1 833	1 098	7	283.624
32	2 047	1 916	1 151	12	282.016
32	4 095	1 987	1 205	21	280.580
32	8 191	2 115	1 260	41	275.558
128	255	5 173	2 974	6	309.900
128	511	5 593	3 205	10	266.280
128	1 023	5 865	3 144	15	290.370
128	2 047	6 153	3 293	25	278.861
128	4 095	6 318	3 443	46	277.439
128	8 191	6 731	3 594	90	275.461

Tabulka 2. FPGA resources requirements of our binary search tree implementation in selected configurations.

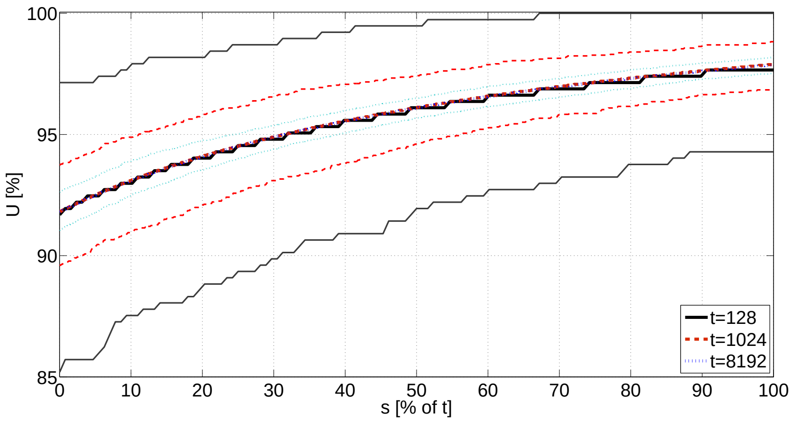
Key				FPGA Resources			Frequency	Mean	Mean
Width	d	t	s	LUTs	FFs	BRAMs	[MHz]	Utilization	Capacity
32	2	8 192	2 047	3 721	2 111	45	264.116	**83.5 %**	**15 388**
32	3	8 192	4 095	4 138	2 221	71	265.437	**96.7 %**	**27 711**
128	2	1 024	255	8 336	4 059	15	257.631	**83.5 %**	**1 923**
128	3	1 024	511	9 564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

Tabulka 3. FPGA resources requirements and memory utilizations of our lookup engine implementation.

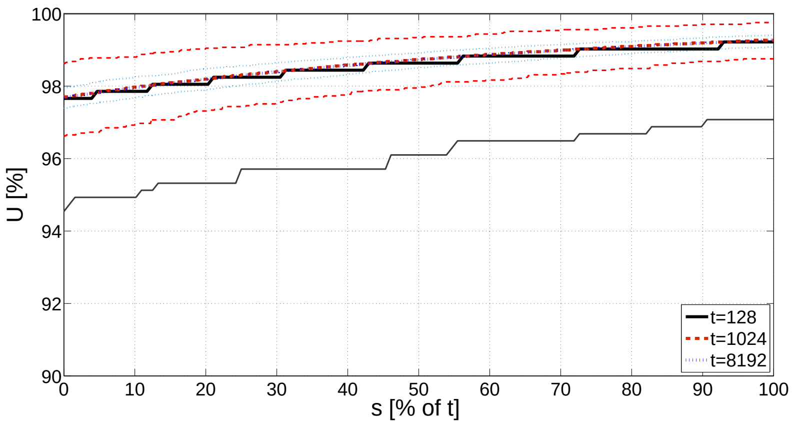
3. Lampson, B.; Srinivasan, V.; Varghese, G.: IP Lookups Using Multiway and Multicolumn Search. In *INFOCOM*, 1998, s. 1248–1256.
4. Naous, J.; et al.: Implementing an OpenFlow Switch on the NetFPGA Platform. In *Proceedings of ANCS*, NY, USA, 2008, ISBN 978-1-60558-346-4, s. 1–9.
5. ONF Market Education Committee: Software-Defined Networking: The New Norm for Networks. Onf white paper, Palo Alto, CA, USA, 2012.
6. Pagh, R.; Rodler, F. F.: Cuckoo Hashing. *J. Algorithms*, ročník 51, č. 2, Květen 2004: s. 122–144, ISSN 0196-6774.
7. Putze, F.; Sanders, P.; Singler, J.: Cache-, Hash-, and Space-efficient Bloom Filters. *J. Exp. Algorithmics*, ročník 14, Leden 2010: s. 4:4.4–4:4.18, ISSN 1084-6654, doi:10.1145/1498698.1594230.
8. Song, H.; et al.: Fast Hash Table Lookup Using Extended Bloom Filter: An Aid to Network Processing. *SIGCOMM Comput. Commun.*, 2005, ISSN 0146-4833, doi:10.1145/1090191.1080114.
9. Tran, T.; Kittitornkun, S.: FPGA-Based Cuckoo Hashing for Pattern Matching in NIDS/NIPS. In *MNGNS*, LNCS, 2007, ISBN 978-3-540-75475-6.
10. Yang, X.; et al.: High-Performance random data lookup for network processing. In *SOC Conference*, 2010, ISSN Pending, s. 272–277.



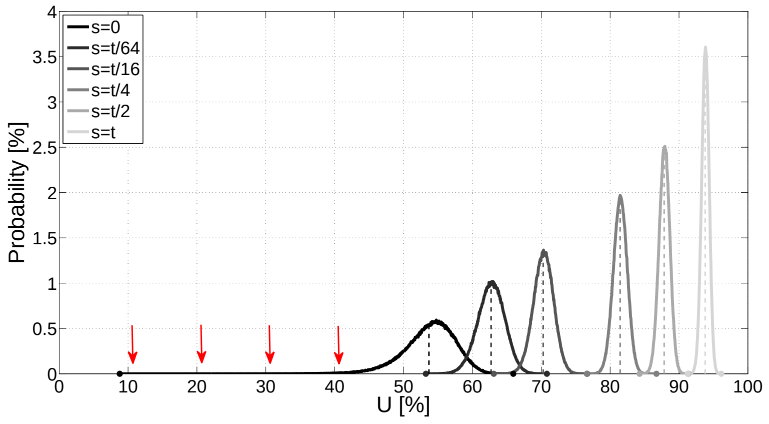
Obrázek 5. Achievable memory utilization for cuckoo hash with two tables ($d = 2$) for different sizes of stash.



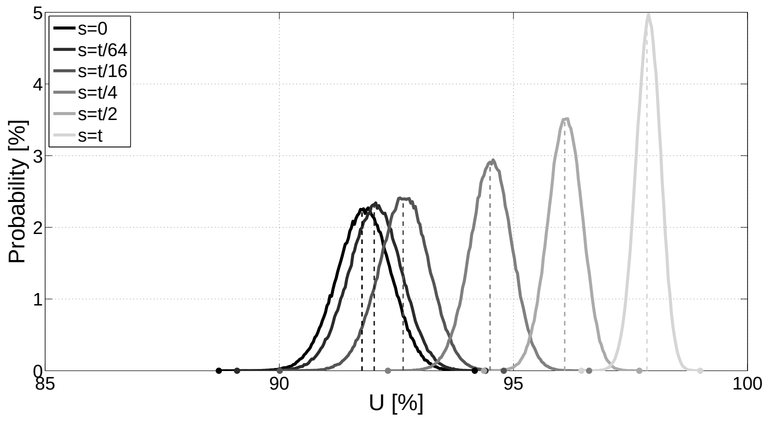
Obrázek 6. Achievable memory utilization for cuckoo hash with three tables ($d = 3$) for different sizes of stash.



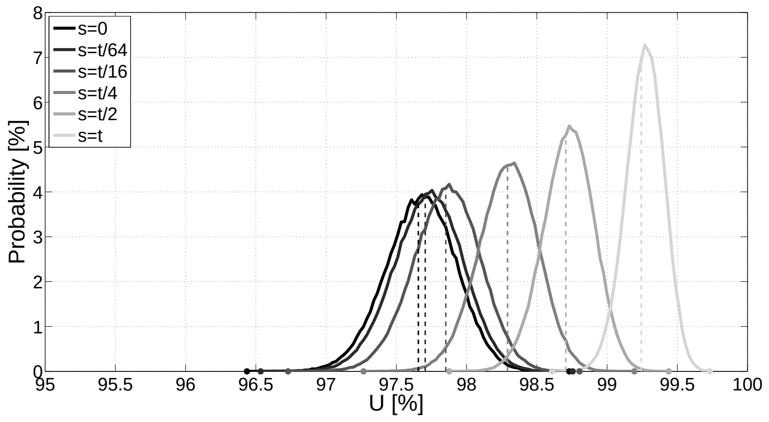
Obrázek 7. Achievable memory utilization for cuckoo hash with four tables ($d = 4$) for different sizes of stash.



Obrázek 8. Probability distribution of achievable memory utilization for cuckoo hash with two tables ($d = 2$, $t = 1\,024$).



Obrázek 9. Probability distribution of achievable memory utilization for cuckoo hash with three tables ($d = 3$, $t = 1\,024$).



Obrázek 10. Probability distribution of achievable memory utilization for cuckoo hash with four tables ($d = 4$, $t = 1\,024$).