TECHNOLOGY GUIDE

Intel Corporation

Intel® Ethernet Controller - Predictable Load Distribution Using Partial Toeplitz Hash Collections

Authors

Vladimir Medvedkin

Andrey Chilikin

Konstantin Ananyev

1 Introduction

Most network interface cards (NICs) need load distribution to better manage the flow of data in the network and to ensure that all cores are evenly distributed to queues. Receive side scaling (RSS) is a network driver technology that enables the efficient distribution of network receive processing across multiple CPUs in multiprocessor systems.

In most cases, modern NICs use Toeplitz hash function to distribute packets across the queues with receive side scaling. Due to the nature of that distribution it is not possible for the user to select desired queue. Such shortcoming becomes especially noticeable for the systems with dynamic assignment of flow attributes that is, parts of hashing tuple such as IP addresses, TCP/UDP ports, ESP SPI, and MPLS labels.

In this technology guide, we present a mechanism for predictable binding of flows to arbitrary queues on the remote receiver or on the local NIC for response packets. The proposed mechanism for predictable load distribution can be applied to multiple use cases, especially at the edge of the network.

Table of Contents

Figures

Tables

Document Revision History

REVISION	DATE	DESCRIPTION
001	May 2022	Initial release.

1.1 Terminology

Table 1. Terminology

ABBREVIATION	DESCRIPTION
BW	Bandwidth
CPU	Central Processing Unit
DPDK	Data Plane Development Kit
DST	Destination
ESP	Encapsulating Security Payloads
GF(2)	Galois Field of two elements
GFNI	Galois Fields New Instructions
IDX	Index
IP	Internet Protocol
LFSR	Linear Feedback Shift Register
LSB	Least Significant Bit
MPLS	Multiprotocol Label Switching
MPLS LSP	MPLS Label Switch Path
MPLS LSR	MPLS Label Switching Router
NAT	Network Address Translation
NIC	Network Interface Cards
RSS	Receive Side Scaling
RSS RETA	Receive Side Scaling Redirection Table
SPI	Security Parameters Index
SRC	Source
TCP	Transmission Control Protocol
UDP	User Datagram Protocol

1.2 Reference Documentation

Table 2. Reference Documents

REFERENCE	SOURCE
Intel® Intrinsic Guide	https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html#text=vgf2p8affineqb
Data Plane Development Kit	https://www.dpdk.org/
Intel® Ethernet Controller 800 Series – Dynamic Device Personalization (DDP) for Telecommunications Workloads Technology Guide	https://networkbuilders.intel.com/solutionslibrary/intel-ethernet-controller-800-series-device-personalization-ddp-for-telecommunications-workloads-technology-guide
Intel® Ethernet Controller E810 - Feature Support Matrix	https://cdrdv2.intel.com/v1/dl/getContent/630155?explicitVersion=true&wapkw=receivesidescaling
Galois Fields New Instructions - Method for Calculating Toeplitz Hash Using GFNI Technology Guide	https://cdrdv2.intel.com/v1/dl/getContent/730527

2 Overview of Hardware Implementation of RSS

Modern NICs support multiple queues to provide network processing scalability for multicore CPUs. Incoming network packets can be distributed across these queues by RSS technology with Toeplitz algorithm as the default hashing function. It works as follows: NIC parses ingress packets and generates an n-tuple, which in general is made of specific packet fields. Then, NIC computes Toeplitz hash signature using the above mentioned n-tuple and predefined Toeplitz hashing key (RSS key). To select a destination queue, NIC uses RSS Redirection Table (RSS RETA), which most commonly consists of 2^N entries. Each entry contains index of the queue where packet should be directed to. Depending on the RSS RETA size, NIC masks N least significant bits (LSB) of the RSS hash signature and uses result as an index in the RSS RETA to get the destination queue for the packet.

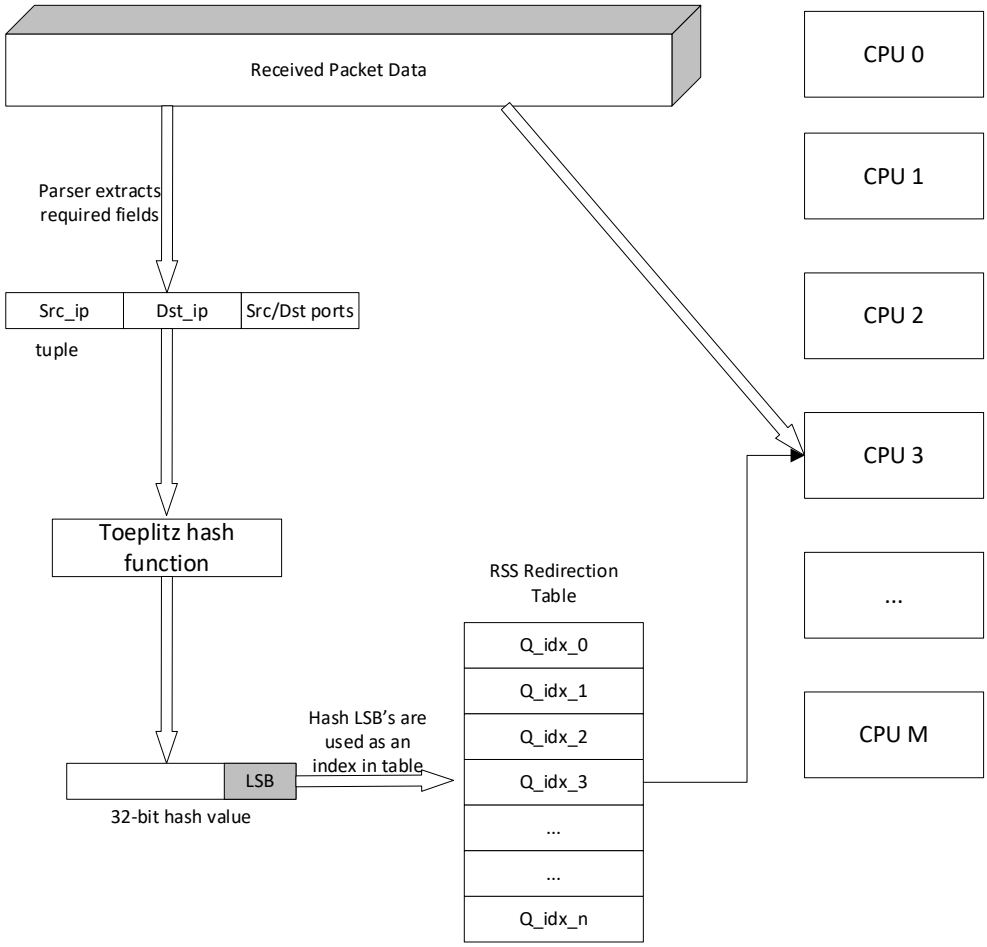


Figure 1. Overview of RSS

Figure 1 shows the RSS use case with one-to-one mapping of queue and CPU.

3 Predictable RSS Algorithm

Toeplitz hash function key, when computed in a specific way can be used to generate RSS hash signatures that meet the distribution requirements of the network packets. Usually, only particular value in LSBs of the hash value is needed because it is used as an index for selecting the queue number from RSS RETA. Thus, by controlling the input value for RSS hashing (by manipulating sub-tuple of the input n-tuple), we can control the queue assignment and enable the RSS distribution to work in a predictable manner.

To compute a desired Toeplitz key, the following two steps are needed:

1. Calculate the key with the given parameters.
2. Calculate the complementary table of bits to be adjusted with the sub-tuple to produce the collision.

The Method for Calculating Toeplitz Hash Using Galois Fields New Instructions Technology Guide describes the Toeplitz hash function as a matrix multiplication with elements over Galois Field (2) (GF(2)):

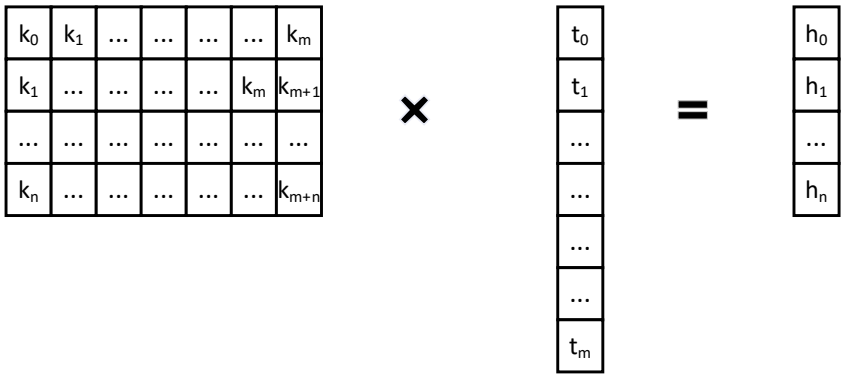


Figure 2. Toeplitz hash representation using matrix multiplication

Matrix K with elements {k0,..., km+n} over GF(2) represents hash key, vector T with elements {t0, ..., tm} over GF(2) represents a tuple, and vector H with elements {h0, ..., hn} over GF(2) represents a hash value. Here, the set of all tuples could be considered as an m-dimension vector space over GF(2), and the set of all hash values H as an n-dimension vector space over GF(2). In practice, resulting hash is a 32-bit value, so the vector space H has 32 dimensions (that is, n = 32).

Thus, T and H form a group with respect to addition (in GF(2), that is modulo2 addition or just XOR) operation - <T, ^> and <H, ^>. Multiplication with matrix K can be treated as a linear map, that is a vector space homomorphism.

So,

$$\mathrm{K} * (t_1 \oplus t_2) = \mathrm{K} * t_1 \oplus K * t_2 = h_1 \oplus h_2 \quad (1)$$

To produce the Toeplitz hash collision, we need to find a desired tuple $t_{desired}$ producing a desired hash value $h_{desired}$. We can express:

$$h_{desired} = \mathrm{K} * t_{desired} \quad (2)$$

So, given original tuple t_{orig} and the corresponding hash value $h_{orig} = K * t_{orig}$ we can express adjustment hash bits h_{adj} like:

$$h_{adj} = h_{desired} \oplus h_{\mathrm{orig}} \quad (3)$$

and from (1) we can express the same for tuples using homomorphism:

$$t_{adj} = t_{desired} \oplus t_{orig} \quad (4)$$

so, we need to adjust t_{orig} with t_{adj} in order to produce hash required collision:

$$t_{desired} = t_{adj} \oplus t_{orig} \quad (5)$$

From (1), (3), and (4):

$$\mathrm{K} * t_{adj} = h_{orig} \oplus \mathrm{h_{desired}} \quad (6)$$

In fact, matrix K usually is not a square matrix. It is not possible to find a K-1 in order to find t_{adj}, so we cannot revert hash function. In other words, the hash function is a one-way function.

We can find associations t_{adj} <-> h_{adj} for each possible value h_{adj}. In general, this is unsolvable task because of the size of t_{adj}, which is very big (for example, 96 bit for IPv4/TCP).

But it becomes possible if the key was built in a specific way.

4 Calculation of the Toeplitz Hash Key

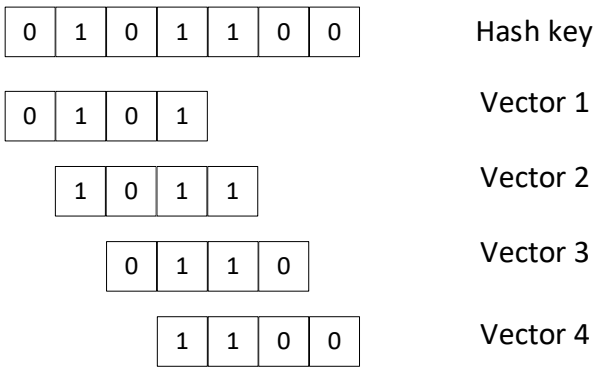
From predictable RSS algorithm we need to calculate hash function for all input t_{adj} that usually is impossible due to size of the tuple that is generally bigger than the size of the hash.

There are several requirements for the key to calculate full association table of t_{adj} <-> h_{adj}.

1. It must be possible to calculate all n-bit values to get proper n-bit h_{adj} to find a required n-bit collision of the hash. That is, there should not be any non-computable h_{adj} values.
2. All variable bits of the t_{adj} must be grouped together, that is, there must be single non splitted (continuous) n-bit sub-tuple to calculate the required h_{adj}.
3. All n-bit h_{adj} values must be calculated from minimal set of t_{adj} – in other words from the bits belonging to the minimal sub-tuple.

To satisfy this requirement, let us use the following approach:

1. As shown in the Method for Calculating Toeplitz Hash Using Galois Fields New Instructions Technology Guide, every n-bit substring of the hash key can be expressed as a vector in n-dimensional vector space as also is shown in the predictable RSS algorithm. Toeplitz hash function represents itself as a linear combination of the key's n-bit substrings where every substring is multiplied by the corresponding bit of the tuple. Every input bit of the tuple has a corresponding n-bit substring of the hash key.
2. To generate any arbitrary n-bit value n of n-bit substrings (that is, vectors) of the hash key must be linear independent from each other that is, be a basis of the vector space. This means that it requires exactly n variable bit of the tuple to generate all possible n-bit h_{adj}.
3. The grouping requirement needs that two nested basis vectors must share (n-1) bit and all n n-bit basis vectors must span in the hash key a bit sequence with length equal to 2*n – 1 bit. For example, for n = 4:



All vectors are linearly independent from each other, and the two nested vectors share n – 1 bit. That is, vectors can be calculated recursively:

$$V_{n+1} = f(V_n) \quad (7)$$

4. There exists a cyclic vector v in V=GF(2^n) for matrix A, meaning that {A0 v, Av, A2v, ..., An−1v} is a basis of V, where A is the Frobenius companion matrix of some polynomial[1]. This means we can express our f(x) as a multiplication of the companion matrix with some initial vector v to generate a basis of a vector space. Unfortunately, not every initial vector is applicable for an arbitrary companion matrix.
5. If Frobenius companion matrix A is companion of the monic polynomial irreducible over GF(2), then every initial non-zero vector v can be multiplied by A recursively spanning a basis of V. If this polynomial is also prime over GF(2), then the bit sequence in this case is called an m-sequence. Refer to Appendix A for more information.

5 Complementary Table Calculation

Complementary table will be composed of 2^N entries, where N is a number of the resulting hash value's least significant bits to calculate collision on. Each entry maps an adjustment hash bit, which are used as a key with a corresponding adjustment bits of the

[1] Horn, Roger A.; Charles R. Johnson (1985). Matrix Analysis

tuple, which are used as a value. For every non-zero n-bit value we calculate n-bit Toeplitz hash signature using a corresponding part of the hash key containing pre-generated m-sequence (with degree n polynomial) and insert in the complementary table a pair <hash_signature -> n-bit_value> as shown on Figure 3.

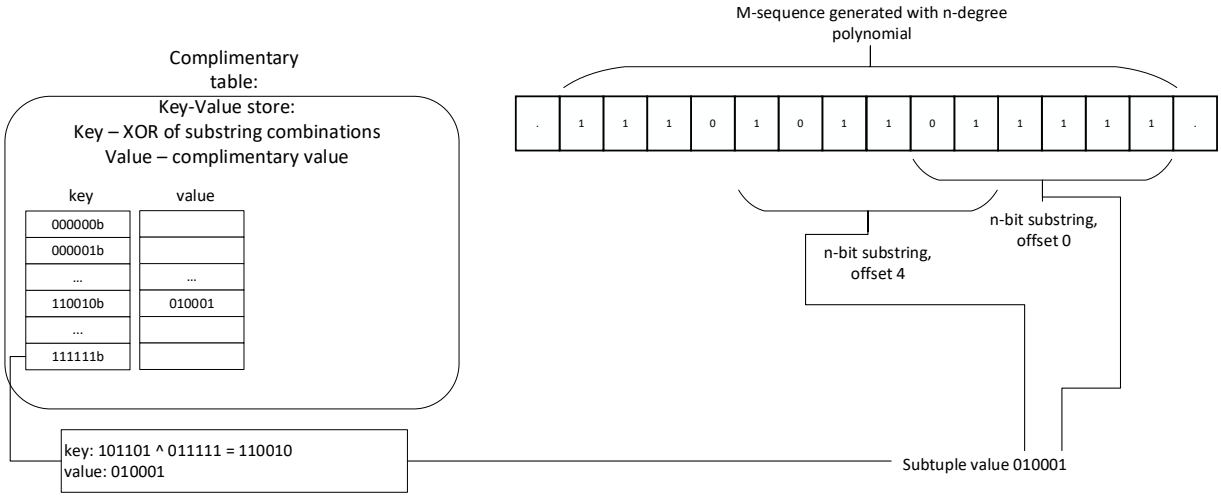


Figure 3 Complete complementary table.

Complementary table now can be used to find sub-tuples (variable part of the full tuple) that will lead to hash signature calculation in a way that LSB's of calculated signature will have required value. Complementary table has 2^lsb key-val entries.

6 Sub-tuple Value Selection Algorithm for Queue Assignment

As shown in Figure 4, to control queue assignment thereby making RSS distribution to work in a predictable manner, we followed these steps:

1. Generate a tuple with a random sub-tuple.
2. Select a desired LSB value for the hash signature.
3. Calculate hash value for the given tuple.
4. Perform XOR with the desired LSB value and use the result to lookup in the complementary table.
5. Find a set of bits using LSBs of XOR of two hashes as a key.
6. XOR previously found bits with the sub-tuple bits to get the new value of the sub-tuple in a way that the hash signature of the full tuple will have the required least significant bits.
7. If the new value of the sub-tuple is already in use, repeat this procedure from step 1 until unused value of the sub-tuple is found.

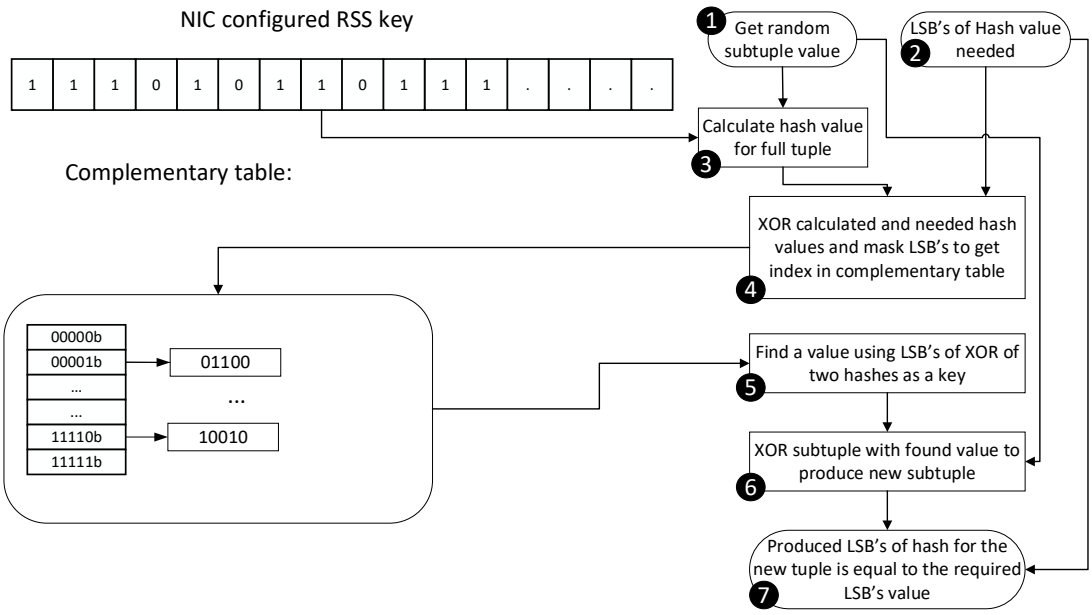


Figure 4 Sub-tuple value selection algorithm

7 Data Plane Development Kit (DPDK) API

To prove the concept, the following DPDK APIs were used:

Control plane functions:

```
/**
Create a new thash context.
**/
struct rte_thash_ctx *
rte_thash_init_ctx(const char *name, uint32_t key_len, uint32_t
reta_sz, uint8_t *key, uint32_t flags);

/**
Add a special property to the Toeplitz hash key inside a thash context.
Creates an internal helper struct which has a complementary table to calculate
Toeplitz hash collisions.
**/
int
rte_thash_add_helper(struct rte_thash_ctx *ctx, const char
*name, uint32_t len, uint32_t offset);

/**
Get a pointer to the Toeplitz hash contained in the context. It changes after
each addition of a helper. It should be installed to the NIC.
**/
const uint8_t *
rte_thash_get_key(struct rte_thash_ctx *ctx);
```

Data plane functions:

```
/**
```

```
Get a complementary value for the subtuple to produce a partial Toeplitz hash
collision. It must be XOR'ed with the subtuple to produce the hash value with
the desired hash LSB's.

**/

uint32_t
rte_thash_get_complement(struct rte_thash_subtuple_helper *h, uint32_t hash,
uint32_t desired_hash);


/**

Function prototype for the rte_thash_adjust_tuple to check if adjusted tuple
could be used. Generally it is some kind of lookup function to check if
adjusted tuple is already in use

**/

typedef int (*rte_thash_check_tuple_t)(void *userdata, uint8_t *tuple);


/**

Adjusts tuple in the way to make Toeplitz hash has desired least significant
bits

**/

int
rte_thash_adjust_tuple(struct rte_thash_ctx *ctx,
      struct rte_thash_subtuple_helper *h,
      uint8_t *tuple, unsigned int tuple_len,
      uint32_t desired_value, unsigned int attempts,
      rte_thash_check_tuple_t fn, void *userdata);
```

- **rte_thash_init_ctx()** – Creates the context associated with a target NIC/set of NICs, which could share the same hash key and other RSS related configuration. Internally, the context has the set of all helpers associated with the context and the RSS hash key. To initialize the context, user must specify size of the hash key as well as logarithm of the size of RSS Redirection Table (RETA) that is, the number of least significant bits from hash value used as an index in RSS RETA. Key and flags are optional.
- **rte_thash_add_helper()** – Creates the helper associated with the given context. User must specify the variable sub-tuple part passing length and offset of this sub-tuple. This function generates the m-sequence inside the hash key for a corresponding length and offset and creates complementary table with adjustment bits for the corresponding sub-tuple. This function changes the hash key, which is kept inside the context so it must be called before NIC initialization.
- **rte_thash_get_key()** – Returns the hash key associated with the context, which must be installed into the NIC on init.
- **rte_thash_get_complement()** – Finds the complementary bits for the sub-tuple defined for the corresponding helper. User must specify helper, the hash value of the tuple he wants to change, and the desired hash value. This is shown as step 5 in Figure 4.
- **rte_thash_adjust_tuple()** – This function changes the tuple in a way to produce partial hash collision, i.e., LSBs of the Toeplitz hash vale for the new tuple will be equal to LSB's of the desired value. User must provide existing tuple to be changed in the subtuple part and the desired hash value. Optionally user can specify callback function and the user data for it to make some additional checks over altered tuple. In essence, this is a user-friendly implementation of an algorithm described in Figure 4.

8 Use Cases

As mentioned previously in the document, this proposed mechanism for predictable load distribution can be applied to multiple use cases, especially at the edge of the network where, for example, network address translation (NAT) is must for many applications. In NAT, the port number is treated as sub-tuple.

It is also applicable for tunnel environment where tunnel distinguisher needs to be mapped to particular queues to separate different tenants or sub-tunnels (Ipsec SPI, TEID in GTP-u, MPLS tag, and so on). In this case tunnel distinguisher would be treated as sub-tuple.

8.1 Multiprotocol Label Switching (MPLS) Label Allocation

This technique can be used in MPLS traffic engineering for resource management to select a worker core with BW availability.

In most cases it is not possible for Label Switch Router (LSR) to know what protocol is transmitted inside the MPLS Label Switch Path (LSP) without using different heuristic methods. So, MPLS packets could be distributed with RSS amongst the queues only by 20-bit MPLS tag. If LSR assigns a MPLS label randomly without considering destination queue for packets belonging to the corresponding allocating label, then it can lead to uneven traffic distribution causing overload of some cores and underload for the others.

With predictable RSS technique LSR can allocate an MPLS label in a controlled way that packets of a corresponding LSP will be handled by desired queue and thus by the desired core.

8.2 IPSec Security Parameter Index (SPI) Allocation

For IPSec, in case of presence of fat pipe tunnels receiving side can split the tunnel into multiple sub tunnels and assign SPI in a way that guarantees even distribution of a received IPSec packet among the queues.

8.3 TCP Stack

In the TCP (Transmission Control Protocol) client connection establishment process, user can select source port in a way that guarantee input packets for this connection will arrive on a particular queue. For example, this might be helpful feature for fast proxy implementations. It allows to avoid additional synchronizations and allows to maintain cache locality.

8.4 Network Address Translation (NAT)

In general, on network address translation (NAT), packets belonging to original and reverse directions of a single bidirectional connection will be distributed by RSS to different queues as shown in Figure 5.

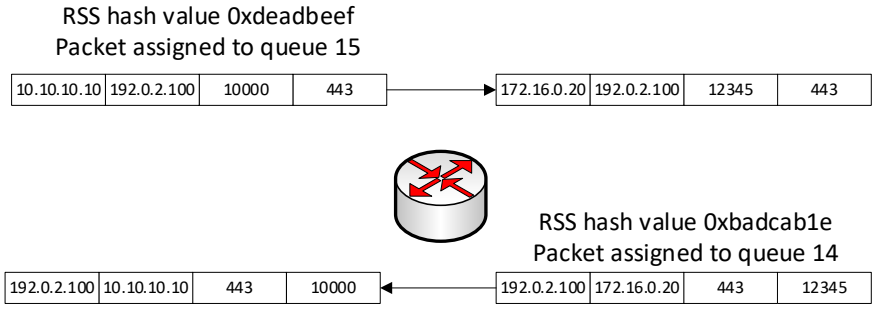


Figure 5. Packet processing in NAT with random port selection

NAT usually requires maintaining a connection table. In this case it requires some sort of synchronization between cores and causes additional overhead in packet processing. On a translation it is possible to choose a source port in a way that guarantees that reverse packets will arrive to desired queue. It is shown in Figure 6.

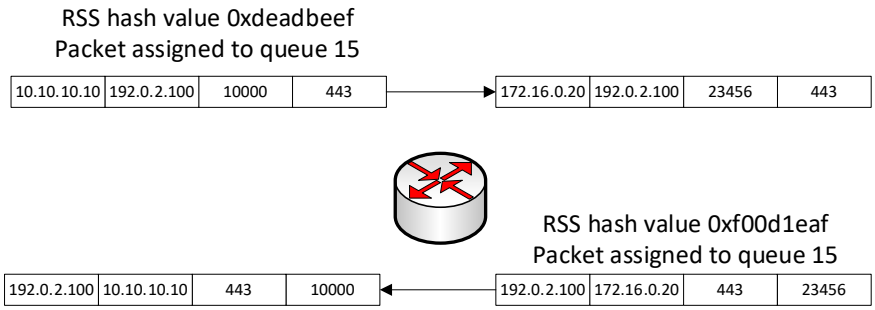


Figure 6. Packet processing in NAT with port selection using predictable RSS

That allows to split connection table among threads and eliminate synchronization overhead. This can significantly simplify NAT implementation and improve the overall performance.

9 Summary

The described mechanism for predictable load distribution using Toeplitz hash collision can be implemented with any modern NIC or other network equipment, which supports Toeplitz hash function for RSS and does not require any specific offloads on the NIC. This can be useful in wide range of popular network processing functions.

Further improvements can be achieved with GFNI implementation of the Toeplitz hash function described in the Method for Calculating Toeplitz Hash Using Galois Fields New Instructions Technology Guide.

Appendix A Calculation of the Toeplitz Hash Key

Theorem: In the sequence, generated by Linear Feedback Shift Register (LFSR) with n-degree polynomial, irreducible over GF(2), linear combination of n 1-bit shifted from each other n-bit subsequences produces all possible n-bit values. In other words, any n n-bit subsequences, 1-bit shifted from each other, inside the sequence forms a basis on an n-dimension vector space over GF(2).

Proof: The sequence could be represented as a recurrent sequence generated by Frobenius companion matrix multiplied with a current value.

$$v_{m+1} = F * v_m \quad (8)$$

Where F is a companion matrix of some polynomial defined for LFSR and vm is a m-th value of the sequence.

So, we need to proove that set of n vectors v: {v0, v1, ..., vn-1} forms a base of n-dimension vector space V, that is, linear combinations of {v0, v1, ..., vn-1} spans a vector space V.

Consider matrix F of the monic polynomial p irreducible over GF2.

(8) could be represented as

$$v_{m+1} = v_m * x \bmod \mathrm{p} \quad (9)$$

from (9) we have:

$$v_{m+n} = v_m * x^n \ mod\ p \quad (10)$$

we can express vm mod p as a constant “c” and different combinations on n consequent v will be expressed as:

$$\mathrm{c} * \sum_n x^{n-1} \ mod\ p$$

so, since $\sum_n \mathrm{x}^{n-1}$ – is a polynomial of degree (n - 1) and p – is an irreducible polynomial of degree n, then every $c * \sum_n x^{n-1} \ mod\ \mathrm{p}$ is unique and all different linear combinations spans a vector space, i.e. {v0, v1, ..., vn-1} are basis of V.

0522/DN/WIPRO/PDF 723538-001US