# Efficient GF Arithmetic for Linear Network Coding using Hardware SIMD Extensions

Stephan M. Günther, Maximilian Riemensberger, Wolfgang Utschick
Associate Institute for Signal Processing, Department of Electrical Engineering
Technische Universität München
Email: {guenther,riemensberger,utschick}@tum.de

***Abstract*—A limiting factor for the performance of coded packet networks is the inherent arithmetic complexity of network coding. This is in particular true for high-throughput networks such as IEEE 802.11n/ac, but also for lower throughput embedded and mobile systems as well as for alternate applications of network coding such as replication of data in distributed systems. While arithmetic complexity is normally not an issue when operating in GF(2), any higher order fields suffer a severe performance degradation. This paper presents hardware-efficient implementations for GF(2), GF($2^2$), GF($2^4$), and GF($2^8$) using different levels of SIMD extensions offered by the x86 and ARM processor architectures. The results are compared to scalar implementations without SIMD, showing an increase of up to factor 15 (x86) and 5 (ARM), respectively. The implementation of the finite field arithmetic called libmoepgf is published under GPLv2 at [1].**

## I. Introduction and related work

A limiting factor for practical deployment of network coding [2] is the inherent computational complexity of coding operations. In practice, generation sizes are most often limited to $N \leq 128$ for reasons such as coding delay and available backlog. Although the complexity to solve the resulting linear system of equations is $\mathcal{O}\left(N^3\right)$, for such generation sizes the constant induced by coding operations rather than solving the linear system of equations is decisive [3–5]. In order to produce a single coded frame, up to $N$ frames have to be multiplied by random coefficients and summed over some finite field. The transmit rate is therefore severely affected by the generation size. This limitation is one major reason why GF(2) is often preferred over larger fields. In case of GF(2), addition as well as multiplication reduce to simple binary operations, which are fast and straight-forward to vectorize. Furthermore, in case of random linear network coding, half of the operations is a multiplication by zero, i.e., no operation at all during encoding. For these reasons, GF(2) outperforms any other finite field in terms of throughput. In addition, coding overhead due to transmission of coefficient vectors is negligible. However, this comes at cost of random linear dependencies, which cause an average overhead of two additional coded packets to allow for decoding [6]. This overhead is mostly independent of the generation size and thus decreases with larger generations.

Coding over larger fields such as GF($2^4$) and GF($2^8$) is slower by an order of magnitude and tends to become less efficient as the field size is further increased. An interesting exception is GF($2^{32}-5$) [3]. Thus computational complexity is of particular importance for mobile applications with demands on high throughput and low energy consumption. Although the performance of mobile devices has significantly increased over the past few years, coding over GF(2) is still favorable in terms of throughput and energy efficiency [5, 7].

The computational complexity of coding operations on binary extension fields led to intensive research to utilize hardware more efficiently. Algorithms to perform coding operations on GPUs are investigated in [8–12]. However, the frameworks needed to access the GPU and the necessary copy operations between GPU and main memory involve a significant overhead that amortizes only for large packet and generation sizes. In addition, powerful GPUs are required that may be available on streaming servers but not on routers, mesh nodes, mobile, and embedded devices. Efficient algorithms for finite field arithmetic on various field sizes using vectorization are described in [13]. Parallelization on Intel's MIC[1] architecture is proposed in [14]. Dedicated hardware designed to accelerate random linear network coding is presented in [15]. A software implementation is available as part of the Kodo network coding library [16], which aims at a compromise between flexibility and performance.

In this paper, we discuss and compare algorithms for efficient finite field arithmetic over GF(2), GF($2^2$), GF($2^4$), and GF($2^8$), which are suitable for network coding and exploit vectorization instructions offered by modern processor architectures, commonly known as *single instruction multiple data (SIMD)* extensions. We show that performance can be increased by more than an order of magnitude with less than 20 lines of hardware-specific code using C instrinsics, enabling single-threaded encoding speeds of 8 Gbit/s on ultra low voltage x86 mobile processors[2] over GF($2^8$) for a generation of $N = 16$ packets. In addition, we present an algorithm that performs multiplications over finite fields by means of ordinary integer multiplications. It can be implemented in a vectorized manner even without SIMD extensions, but naturally scales with wider registers. It is therefore well suited for microarchitectures targeted at the low-end, embedded, and mobile markets that have no or limited support for SIMD extensions.

The remainder of this paper is organized as follows: Section II introduces binary extension fields and random linear network coding. Algorithms for efficient coding operations are presented in Section III, which are evaluated in Section IV on both Intel x86-64 and ARMv7 architectures for various levels of SIMD extensions and compared to lookup-based scalar implementations. Section V concludes the paper.

[1]many integrated cores
[2]Intel Core i3-4010U, 1.7 GHz, 15 W TDP, Haswell microarchitecture

## II. Random linear coding on binary fields

We consider finite fields $F_q$ of order $q = 2^n$, i.e., binary extension fields of degree $n$. Elements of these finite fields are expressed by polynomials over $\mathbb{F}_2$ of degree $n-1$, i.e.,

$$F_q[x] = \left\{ \sum_{i=0}^{n-1} a_i x^i \,\middle|\, a_i \in \mathbb{F}_2 \right\}. \quad (1)$$

These allow for efficient processing on today's processor architectures as the coefficients $a_i \in \mathbb{F}_2$ are represented by individual bits. For the scope of this paper, we focus on the finite fields with $n \in \{1, 2, 4, 8\}$, namely GF(2), GF($2^2$), GF($2^4$), and GF($2^8$). Elements of those fields naturally fit into processor registers. At the same time, these are the most important finite fields for intra-session network coding as the overhead due to coding coefficients is kept small compared to larger fields such as GF($2^{16}$) and GF($2^{32}$).

In order to construct an extension field of order $q = 2^n$, we need a polynomial $g$ of degree $n$ that is irreducible over $F_q$, i.e., it cannot be represented by the product of two other polynomials in $F_q[x]$ of degree strictly less than $n$. Such a polynomial is guaranteed to exist [17], and in general not unique. Depending on the chosen polynomial, one gets a finite field isomorphic to $\mathbb{F}_q[x]$. Addition of $a, b \in F_q[x]$ is defined as

$$a(x) + b(x) = \sum_{i=0}^{n-1} a_i x^i + \sum_{i=0}^{n-1} b_i x^i = \sum_{i=0}^{n-1} (a_i + b_i)\, x^i. \quad (2)$$

Note that coefficients are added according to the rules of the additive group associated with $\mathbb{F}_2$, that is, addition is done modulo 2. Therefore, addition reduces to simple XOR operations. The product of $a, c \in F_q[x]$ is the unique remainder

$$b(x) = (a(x) \cdot c(x)) \bmod g(x). \quad (3)$$

Since $g$ constrains the maximum degree of the result such that $b \in F_q[x]$, it is sometimes called *reduction polynomial*. Irreducibility of $g$ ensures that the remainder is unique for any two factors (except for commutativity).

Polynomials $a \in F_q[x]$ are considered *data words* of $n$ bit length. A data packet of length $kn$ bit is thus represented by a vector $\boldsymbol{a} \in F_q^k[x]$. A *generation* of $N$ source packets can then be written as matrix $\boldsymbol{A} = [\boldsymbol{a}_1, \ldots, \boldsymbol{a}_N]$. A coded packet is obtained by

$$\boldsymbol{b} = \boldsymbol{A}\boldsymbol{c} = \sum_{i=1}^{N} c_i \boldsymbol{a}_i, \quad (4)$$

where $\boldsymbol{c} = [c_1, \ldots, c_N] \in F_q^N[x]$ denotes a vector of random coefficients that are drawn uniformly and independently distributed from $F_q[x]$. Further, we define the *encoding throughput* as the total size of encoded packets over a time interval measured in Gbit/s.

## III. Algorithms

The vectorized algorithms perform $M = B/n$ MUL (multiplication) or MADD (multiply and add) operations in parallel, where $B$ denotes the register width and $n$ the degree of the finite field, i.e., the length of data words. Depending on the hardware capabilities, two different algorithms are used. The first one implements a branch-free, vectorized polynomial multiplication using register shifts, integer multiplications, and XOR operations only. Since it does not rely on any specialized vector instructions, it even allows for an implementation on general purpose registers (GPRs) or graphics processing units (GPUs). It is therefore suitable for microarchitectures without SIMD extensions, but does benefit from larger vector registers if available. However, the number of shifts depends on the degree of the finite field, which makes the approach inefficient for large fields. We refer to this algorithm as *imul (integer multiplication)* algorithm.

The second algorithm follows an elegant approach first described by Anvin [18] and later by Plank et al. [13]. The basic idea is to store results that are calculated in advance in an array that fits the size of SIMD vector registers. For instance, in case of GF($2^4$), all 16 possible results for the multiplication with some constant value $c \in F_{16}[x]$ are stored in such an array. The desired values are indexed by an advanced vector instruction called *shuffle* or *table load*. Larger fields can be reduced to GF($2^4$) making the algorithm also suitable for GF($2^8$). Since it requires specialized vector instructions, it is only suitable for newer hardware (Intel SSSE3[3] and later, ARM NEON with minor restrictions). Furthermore, the algorithm turns out to be inefficient for fields smaller than $2^4$ and for any fields whose degree is not a power of two. We refer to this approach as *shuffle* algorithm. Section III-A discusses the imul and Section III-B the shuffle algorithm.

### A. imul algorithm

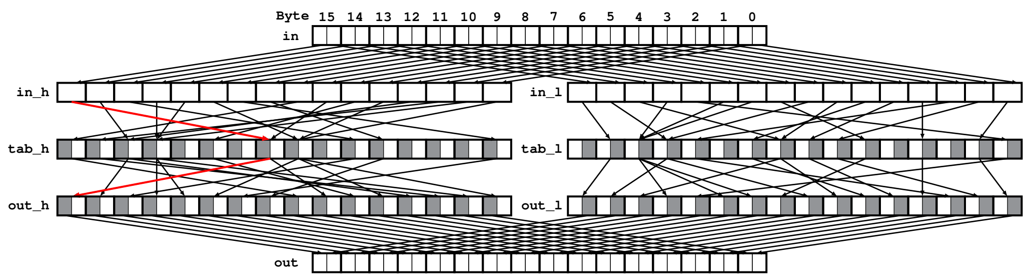
The most significant performance impact of the naive polynomial multiplication described in Section II is the bitwise testing of operands: if a bit is set to 1, the corresponding remainder is XORed into an accumulator register, whereas it is ignored if it is 0. This causes a sequence of branches that are impossible to predict. However, the following reformulation of (3) gives rise to a more efficient implementation:

$$\begin{aligned} b(x) &= (c(x) \cdot a(x)) \bmod g(x) \\ &= \left( c(x) \cdot \sum_{i=0}^{n-1} a_i x^i \right) \bmod g(x) \\ &= \sum_{i=0}^{n-1} a_i \left( c(x) x^i \bmod g(x) \right). \end{aligned} \quad (5)$$

Equation (5) holds since the individual summands are reduced according to $g$ and addition is performed modulo 2 and thus without carry. The improved multiplication uses a $2^n \times n$ table containing the powers $c(x)x^i \bmod g$ for $i \in \{0, \ldots, n-1\}$ for all $c \in F_q[x]$. Given this table, multiplication is performed by a single lookup using $c$ as index followed by a series of conditional XOR operations (additions). The conditions can be replaced by integer multiplications with $a_i \in \{0, 1\}$, which yields a branch-free polynomial multiplication over $F_q[x]$.

Vectorization is now straight-forward (see Listing 1, which shows the implementation for GF($2^2$)): Assume that a region

[3]Supplemental SSE3 (streaming SIMD extensions), first introduced with the Intel Core microarchitecture in 2006.



```
in = _mm_load_si128(region)

in_l = _mm_and_si128(in,0x0f)
in_h = _mm_and_si128(in,0xf0)
in_h = _mm_srli_epi64(in_h,4)

out_l = _mm_shuffle_epi8(in_l,tab_l)
out_h = _mm_shuffle_epi8(in_h,tab_h)

out = _mm_xor_si128(out_h,out_l)
_mm_store_si128(region,out)
```

Figure 1. Schematic of multiplying `region` by some constant over GF($2^4$) and corresponding implementation of the shuffle algorithm using C-intrinsics on 128 bit SIMD registers. Mask registers are defined as `mask_h` = `0xf0` and `mask_l` = `0x0f`, respectively, repeated 16 times to fill a 128 bit register.

of bytes containing data words should be multiplied by a constant $c$. First, registers are loaded with bitmasks used to isolate the coefficients $a_i$ for $i \in \{0, \ldots, n-1\}$ of each word in the region. The bitmasks are repeated $M$ times in each of those registers. Second, the $n$ powers of the constant factor are loaded from the lookup table. In the main loop, $M$ words of the region to be multiplied by $c$ are loaded into a temporary register. This register is ANDed with the bitmasks and shifted such that the respective bit of interest is guaranteed to be the lowest order bit of each word. The resulting $n$ registers are multiplied by the respective powers of the constant factor $c$. XORing the results yields $M$ products at once, which are written back to memory.

Algorithm 1. imul for GF($2^2$) using 64 bit registers.

```
void mulrc4_gpr64(uint8_t *region, uint8_t c, int length)
{
  uint64_t r64[2];
  uint64_t mask1 = 0x5555555555555555;
  uint64_t mask2 = 0xaaaaaaaaaaaaaaaa;
  uint64_t *reg = (uint64_t *)region;
  uint8_t  *pow = powers[c];

  for (; length > 0; reg++, length-=8) {
    r64[0] = ((*reg & mask1) >> 0) * pow[0];
    r64[1] = ((*reg & mask2) >> 1) * pow[1];
    *reg = r64[0] ^ r64[1];
  }
}
```

The algorithm performs $M = B/n$ multiplications per loop without any branches and naturally scales with the register width. Note that it does not need any vector instructions, i.e., the multiplication can be safely performed on GPRs as well as vector registers if available. The code in Listing 1 can easily be rewritten to make use of wider SIMD registers and to support regions whose lengths are not multiples of the register width. It is suitable for any field size as long as the word length $n$ does not exceed the register width $B$. However, for $M$ polynomial multiplications the algorithm requires $n-1$ shifts, $2n$ logic instructions (AND/XOR), and $n$ integer multiplications. Consequently, it is best suited for small field sizes.

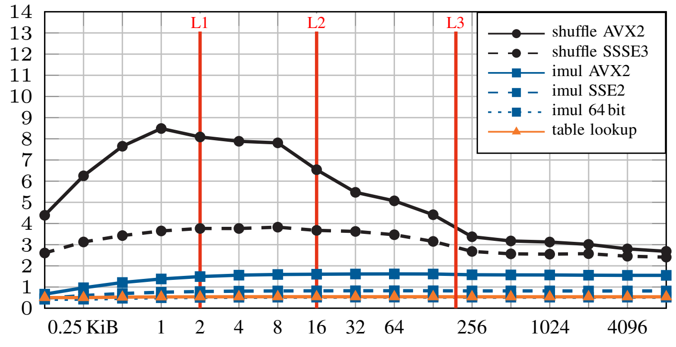
### B. *shuffle algorithm*

The basic idea of the shuffle algorithm originally proposed in [13] is to store all possible multiplication results in a table such that the constant $c$ can be used as row index to obtain all possible results when multiplying a field element by $c$. The value of the respective field element is then used as column index to select the multiplication result. For an efficient implementation, it is necessary to perform this lookup operation for multiple operands in parallel. The suitable *shuffle* operation has been introduced by Intel with SSSE3. It allows to lookup 16 values of 1 B each given a vector of 16 index values in the set {0,1,...,15}. Larger indices are considered out of range and return zero. The difficult part consists in mapping the multiplication to lookups in tables of $2^n \times 16$ elements, where $n$ denotes the degree of the finite field.

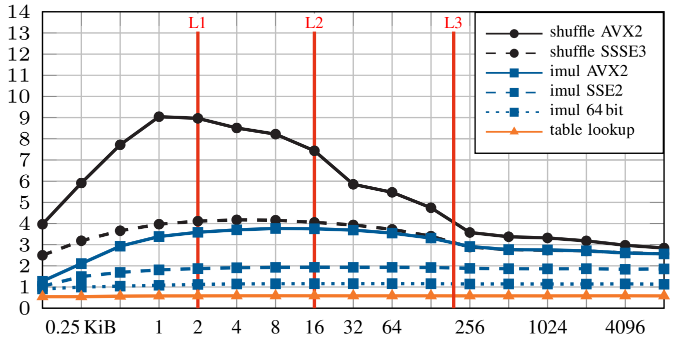
*1) Multiplication over GF($2^4$):* Consider Figure 1, which sketches the contents of 128 bit registers during multiplication over GF($2^4$). First, 16 B corresponding to 32 words are loaded from memory into register `in`. Next, each byte of that register is split into its high (low) nibble. The high nibble is shifted 4 bit to the right such that the upper 4 bit of the registers `in_h` and `in_l` are set to zero. These registers act as indices for the *shuffle* operation in the next step. The registers `tab_h` and `tab_l` are loaded with all possible results when multiplying by a constant $c$. For instance, assume that the leftmost byte in `in_h` contains the value `0x07`. Thus, it selects the byte at position 7 from the vector `tab_h` yielding the corresponding multiplication result in `out_h`. The shuffling is performed by a single hardware instruction for the high and low tables, respectively. The result vectors `out_h` and `out_l` are finally XORed yielding the result packed in a single register.

This procedure fits perfectly for Intel's SSSE3 extensions as it provides the necessary shuffle operation on a byte-by-byte basis for exactly 16 indices. AVX2 extends the register width to 256 bit, i.e., 64 data words. The corresponding shuffle operation still accepts indices ranging from 0 to 15. The only required modification is to double the high and low tables, i.e., repeat the values at byte positions 0 to 15 at positions 16 to 31. Implementation using the NEON instruction set extension of ARM platforms is also possible. However, the shuffle operation, which is there called *table lookup*, returns only a 64 bit vector, i.e., only 16 multiplications are performed simultaneously. Furthermore, the elements of the lookup table have to be interleaved as the ARM processors perform an automatic interleaving on table loads ([19, p. 1095]).

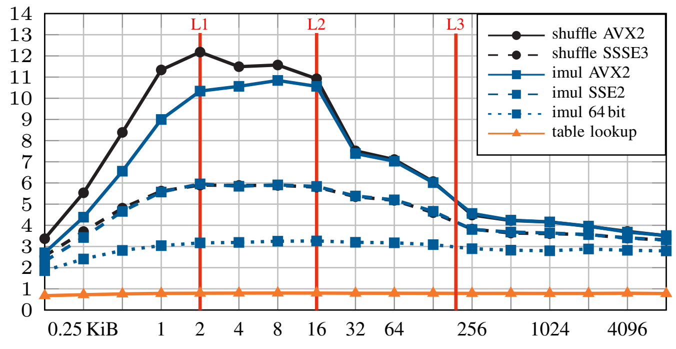
*2) Adaptation for GF($2^2$):* The shuffle operation selects multiplication results from 16 different values. In order to obtain 16 multiplication results when multiplying a constant with a field element of GF($2^2$), we consider two elements at once and perform the multiplication with $M$ words of 2 bit each simultaneously. This way, we obtain low and high tables of $4 \times 16$ elements. However, each of these tables now yield two multiplication results per lookup.



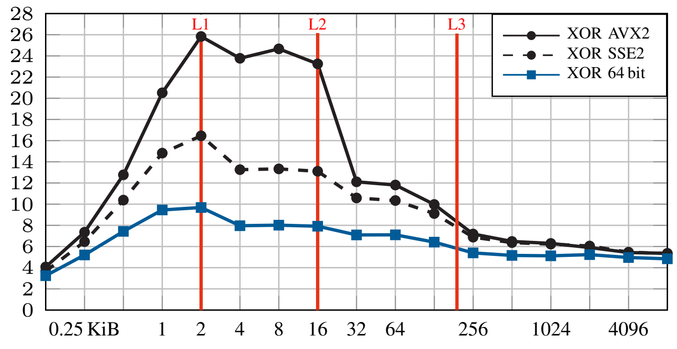
(a) GF($2^8$)



(b) GF($2^4$)



(c) GF($2^2$)



(d) GF(2)

Figure 2. Encoding throughput [Gbit/s] for packet sizes varying from 128 B to 8 MiB in a generation of size 16 on an Intel Core i3-4010U (1.7 GHz, 256 KiB L2 per core, 3 MiB shared L3). The vertical (red) lines indicate the packet sizes at which the working set exceeds the size of the respective cache level.

*3) Other field sizes:* The shuffle algorithm described in this section can be adapted to other field sizes as well. For GF($2^8$), each word is split into its low and high nibble. These are multiplied individually with all field elements, yielding two tables of $256 \times 16$ entries each. Adaptations for GF($2^{16}$) and GF($2^{32}$) are also described in [13].

## IV. Measurements

We compare the random linear encoding throughput according to (4) using a generation of size $N = 16$. The generation of the pseudo random numbers needed for random linear coding is part of the measurements, i.e., the performance of the PRNG[4] has an influence on the encoding throughput. The measurements[5] are repeated for both the shuffle and imul algorithms at packet sizes ranging from 128 B to 8 MiB. As the baseline performance, we use full table lookups, i.e., all possible multiplication results are calculated in advance and stored in a table. This approach is described in [20] and found to be more efficient than log/antilog tables for GF($2^8$).

The packet size has a significant impact on the performance since the overhead for checking the constant factor (multiplication by 0 and 1 are handled separately), setup of any constants/bitmasks needed for the respective algorithm, and generation of random numbers are incurred only once per invocation of the MUL/MADD routines. Consequently, larger packet sizes result in less overhead and thus better performance. This rule of thumb holds true as long as the working set, i.e., any memory regions where data is read from or written to, fits into the processor's caches. At some point, performance degrades as the throughput between caches and memory becomes the limiting factor.

Section IV-A discusses the results on an Intel Core i3-4010U running at 1.7 GHz, which is an ultra low voltage mobile processor (15 W thermal design power) based on the Haswell microarchitecture [21, Chapter 2.1] supporting all SIMD extensions up to and including AVX2. Section IV-B discusses the same measurements performed on a Samsung Exynos 5 Octa[6] running at 1.4 GHz, which is an ARMv7-based Cortex A15 SoC [22] supporting the NEON SIMD extensions and is commonly found in tablet PCs. All measurements are conducted single-threaded.

### A. Intel Core i3-4010U

Consider the results for GF($2^8$) shown in Figure 2a. The imul algorithm without SIMD extensions is on par with the table lookups. The SSE2 implementation is approximately 55 % faster than table lookups and was found to vary significantly between different microarchitectures. The reason is that the performance of instructions on GPRs and their counterparts on SIMD registers may differ. The performance gain between the SSE2 and AVX2 implementations is approximately a factor of two as expected since the register width is doubled.

The shuffle algorithm using SSSE3 (AVX2) instructions is up to 7 (15) times faster than table lookups. Provided that the necessary instruction set extensions are supported, this

[4]We use a linear congruential generator, i.e., $x_{n+1} = (ax_n + c) \bmod m$ with $m = 2^{32}$, $a = 214013$, and $b = 2531011$. Only bits 30 . . . 15 of the result are being used.

[5]Binaries are compiled using `gcc-4.8.1` (x86) and `gcc-4.8.2` (ARM), respectively, with `-O2` and `-funroll-loops` turned on.

[6]ODROID-XU Lite development board, http://www.hardkernel.com

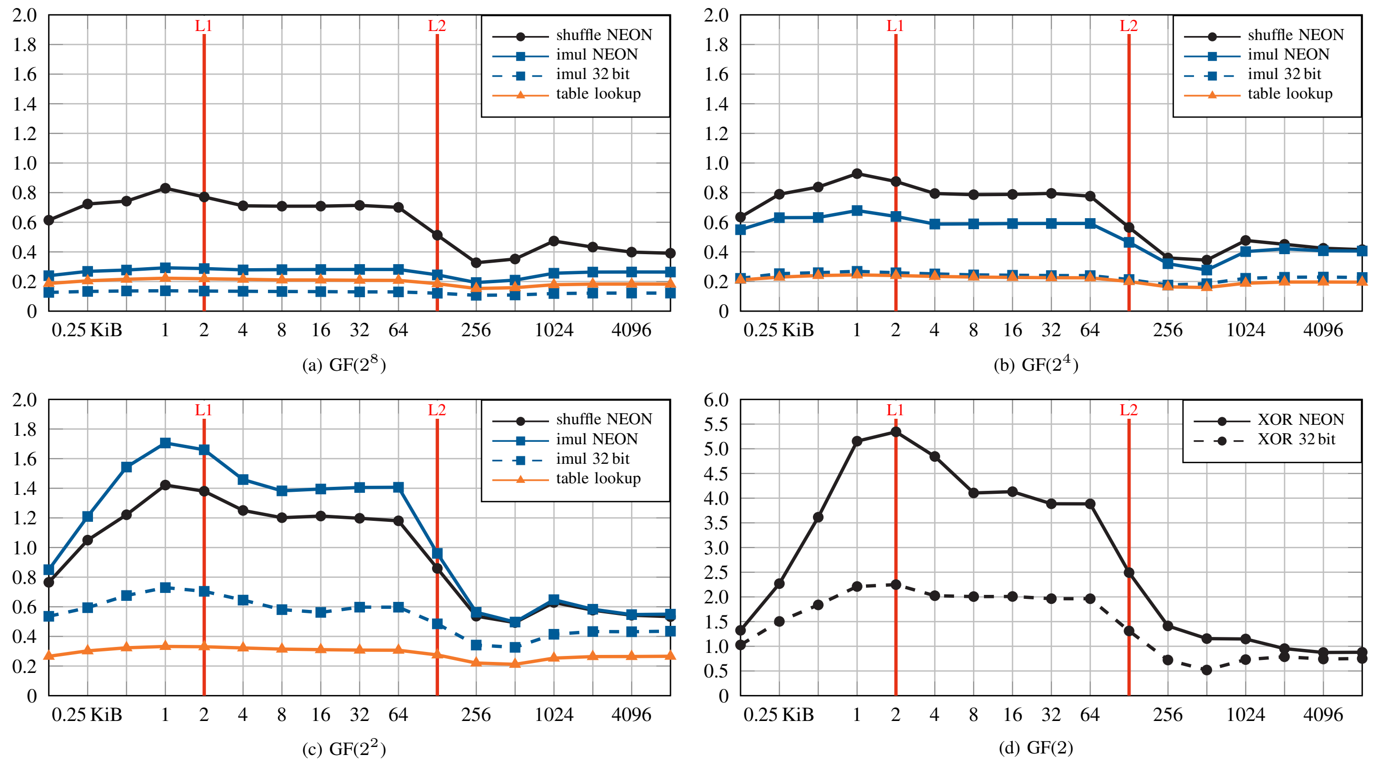


Figure 3. Encoding throughput [Gbit/s] for packet sizes varying from 128 B to 8 MiB in a generation of size 16 on a Samsung Exynos 5 Octa (1.4 GHz, 2 MiB shared L2). The vertical (red) lines indicate the packet sizes at which the working set exceeds the size of the respective cache level.

algorithm is the most efficient one we are currently aware of. However, it also shows the most significant dependency on the packet size for reasons discussed above. As the packet size increases, the relative overhead decreases. At some point, the working set, i.e., a generation of 16 packets plus one packet buffer for the result, exceeds the size of the L2 cache, which is 256 KiB for the Core i3-4010U. The corresponding frame size is indicated by the second vertical (red) line in Figure 2a. Note that the L2 cache size is already exceeded by one frame size since 16 frames of 16 KiB each exactly fill the L2 cache but an additional buffer for the result is needed. Also note that the L1 cache has almost no impact on performance since the Haswell microarchitecture can load 64 B per cycle from the L2 cache [23], resulting in more than 100 GiB/s on L2 hits.

For packet sizes between 16 KiB and 128 KiB, the working set fits into the L3 cache, which is 3 MiB in size. However, the L3 cache also holds the contents of the L2 caches (256 B per core) and acts as a non-coherent victim cache for the integrated GPU, which has a dedicated 512 KiB cache [24]. Therefore, up to $1/3$ of the L3 cache is not usable. Since the working set is slightly larger than 2 MiB, performance already starts degrading at 128 KiB. Repeating the tests on an Intel Core i7-4650U, which has 4 MiB of L3 cache, does shift the performance impact to a packet size of 256 KiB as expected.

The results for GF($2^4$) depicted in Figure 2b show a similar behavior: The imul algorithm is considerably faster since it needs only half the operations compared to GF($2^8$). The minimal increase in throughput of the shuffle algorithm is a result of the smaller field size, i.e., the probability to draw 0 or 1 as a random coefficient is larger, which reduces the MADD operation to no operation or XOR, respectively. In case of GF($2^2$) shown in Figure 2c, all algorithms show a significant throughput gain. Again, the imul algorithm needs only half the operations compared to GF($2^4$), making it competitive with the shuffle algorithm. The performance increase of the shuffle implementations is again solely due to the smaller field size.

Finally, Figure 2d shows the results for GF(2) (note the different scaling of the ordinate). The algorithms degenerate to XOR operations for $c = 1$ and no operation for $c = 0$. Consequently, there is no direct comparison between the algorithms. Since SSSE3 instructions are not needed here, this feature set is omitted in the figure. Without SIMD extensions, coding operations over GF(2) are 15 times faster than table lookups on GF($2^8$). Using AVX2, this gap is reduced to a bit more than a factor of three, which is a very good result considering that half the encoding operations over GF(2) are MADDs by 0.

## B. *Samsung Exynos 5 Octa*

Compared to the results of the previous section, Figure 3 shows that the encoding throughput on ARM-based systems is almost one order of magnitude lower, which is expected as the ARM architecture is optimized for low power consumption instead of high performance. Considering the results for GF($2^8$) shown in Figure 3a, the imul algorithm performs worse than table lookups. This is due to the lack of 64 bit support on the ARMv7 architecture, which cuts the performance of the imul algorithm approximately in half. As the field size decreases, the imul algorithm becomes increasingly more efficient. Over GF($2^2$), it is 15 –20 % faster than the shuffle algorithm.

The Samsung Exynos 5 Octa shows two performance degradations, one when the working set approaches the 32 KiB L1 cache and a second one when it exceeds the 2 MiB L2 cache. It is interesting to observe that the encoding throughput increases again at a packet size of 1 MiB, which is a phenomenon observed with all field sizes. Currently, we do not have a conclusive explanation for this anomaly. However, a Samsung Exynos 4412 (Cortex-A9) does not show this behavior, which is why we assume reasons specific to the Samsung Exynos 5 Octa rather than the ARM architecture in general.

### C. Remarks

Varying the generation size has two effects. First, the encoding throughput decreases nearly proportionally to the generation size provided that packets are arranged in a consecutive memory region. Second, packet sizes at which caches start to saturate change accordingly, e.g. doubling the generation size to $N = 32$ would lead to a performance hit starting at a packet size of 8 KiB instead of 16 KiB in Figure 2.

The PRNG affects the encoding throughput. The overhead due to random number generation is larger for small packets because less time is spent in the actual encoding routines. An analysis of the hardware performance counters using OProfile [25] revealed an overhead due to random number generation of 17 % for 128 B packets and only 2 % for 2048 B packets. We also determined that using the `rand_r` function provided by the `glibc` increases this overhead by a factor of three since it uses a cascade of three linear congruential generators compared to a single one as used in our implementation. Using the non-reentrant variant `rand` further increases the overhead due to a more complex, trinomial PRNG.

The algorithms evaluated in this section are designed as low-level arithmetic functions. Parallel versions are not considered because the overhead for threading and locking would cause an immense performance penalty. Instead, parallelization may occur at a higher level, e.g. by implementing coding sessions as separate threads. In this case, the total encoding throughput scales well with the number of threads as long as the memory bandwidth does not become the limiting factor. For instance, the performance for GF($2^8$) on the Intel Core i3-4010U used for the measurements in Section IV-A nearly doubles when starting two encoders on different generations in parallel.

## V. Conclusion

With scalar implementations on general purpose registers, all field sizes larger than GF(2) are limited to less than 1 Gbit/s on an Intel Core i3-4010U and less than 300 Mbit/s on a Samsung Exynos 5 Octa. Using SIMD extensions easily increases performance by factors of 15 (AVX2) and 5 (NEON), respectively, which significantly reduces the gap to GF(2). Using AVX2 instructions, encoding throughput over GF($2^8$) can achieve up to 98 % of the throughput over GF(2) without vectorization. This comes at cost of less than 20 hardware-dependent lines of code using C instrinsics [1, 13, 18]. We further showed that in the absence of SIMD extensions or without support for shuffling, the imul algorithm is a well-performing and hardware-independent alternative to table lookups.

Given the trend to heterogeneous microarchitectures where both CPU and (integrated) GPU coherently access the same virtual address space, we might see a similar gain when low-level arithmetic functions are offloaded to integrated graphics processors in the near future.

## References

[1] S. Günther and M. Riemensberger, "Supplemental material: libmoepgf," http://moep80211.net/plink/netcod2014.

[2] R. Ahlswede, N. Cai, S.-Y. Li, and R. Yeung, "Network Information Flow," *IEEE Transactions on Information Theory*, vol. 46, no. 4, pp. 1204–1216, Jul. 2000.

[3] M. Pedersen, J. Heide, P. Vingelmann, and F. Fitzek, "Network Coding over the $2^{32} - 5$ Prime Field," in *IEEE International Conference on Communications*, Jun. 2013.

[4] M. Wang and B. Li, "How Practical is Network Coding?" in *IEEE International Workshop on Quality of Service*, Jun. 2006, pp. 274–278.

[5] A. Paramanathan, M. Pedersen, D. Lucani, F. Fitzek, and M. Katz, "Lean and Mean: Network Coding for Commercial Devices," *IEEE Wireless Communications*, vol. 20, no. 5, pp. 54–61, Oct. 2013.

[6] C. Cooper, "On the Distribution of Rank of a Random Matrix over a Finite Field," *Random Struct. Algorithms*, vol. 17, no. 3-4, pp. 197–212, Oct. 2000.

[7] J. Heide, M. V. Pedersen, F. Fitzek, and T. Larsen, "Cautious View on Network Coding – From Theory to Practice," *Journal of Communications and Networks*, vol. 10, no. 4, pp. 403–411, Dec. 2008.

[8] H. Shojania, B. Li, and X. Wang, "Nuclei: GPU-accelerated Many-Core Network Coding," in *IEEE International Conference on Computer Communications*, Apr. 2009, pp. 459–467.

[9] H. Shojania and B. Li, "Pushing the Envelope: Extreme Network Coding on the GPU," in *IEEE International Conference on Distributed Computing Systems*, Jun. 2009, pp. 490–499.

[10] X. Chu, K. Zhao, and M. Wang, "Practical Random Linear Network Coding on GPUs," in *IFIP International Conference on Networking*, 2009, vol. 5550, pp. 573–585.

[11] P. Vingelmann and F. Fitzek, "Implementation of Random Linear Network Coding using NVIDIA's CUDA Toolkit," in *Networks for Grid Applications*, ser. Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering, 2010, vol. 25, pp. 131–138.

[12] S. Lee and W. Ro, "Accelerated Network Coding with Dynamic Stream Decomposition on Graphics Processing Unit," *Computer Journal*, vol. 55, no. 1, pp. 21–34, Jan. 2012.

[13] J. Plank, K. Greenan, and E. Miller, "Screaming Fast Galois Field Arithmetic using Intel SIMD Instructions," *USENIX Conference on File and Storage Technologies*, vol. 11, Aug. 2013.

[14] K. Feng, W. Ma, W. Huang, Q. Zhang, and Y. Gong, "Speeding Up Galois Field Arithmetic on Intel MIC Architecture," in *Network and Parallel Computing*, ser. Lecture Notes in Computer Science, 2013, vol. 8147, pp. 143–154.

[15] A. Nagarajan, M. Schulte, and P. Ramanathan, "Galois Field Hardware Architectures for Network Coding," in *ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, Oct. 2010.

[16] M. V. Pedersen, J. Heide, and F. H. Fitzek, "Kodo: An Open and Research Oriented Network Coding Library," in *Networking 2011 Workshops*, ser. Lecture Notes in Computer Science, V. Casares-Giner, P. Manzoni, and A. Pont, Eds., 2011, vol. 6827, pp. 145–152.

[17] D. Hankerson, A. Menezes, and S. Vanstone, *Guide to Elliptic Curve Cryptography*, 1st ed., Jan. 2004.

[18] H. Anvin, "The Mathematics of RAID-6," 2004.

[19] *ARM Architecture Reference Manual, ARMv7-A and ARMv7-R Edition*, ARM, 2012.

[20] K. Greenan, E. Miller, and T. Schwarz, "Optimizing Galois Field Arithmetic for Diverse Processor Architectures and Applications," in *MASCOTS*, 2008, pp. 257–266.

[21] *Intel 64 and IA-32 Architectures Optimization Reference Manual*, Intel, Jul. 2013.

[22] *Cortex-A15 Technical Reference Manual*, ARM, 2011.

[23] A. Fog, "The Microarchitecture of Intel, AMD, and VIA CPUs," Feb. 2014, http://www.agner.org.

[24] *Intel Open Source Graphics Programmer's Reference Manual (PRM) for the 2013 Intel Core Processor Family*, Intel, Dec. 2013.

[25] J. Levon, P. Elie, M. Johnson, S. Suthikulpanit, W. Deacon, G. Allard, D. Hansel, and R. Richter, "OProfile," 2013, http://oprofile.sourceforge.net.