2011 20th IEEE Symposium on Computer Arithmetic

Bit-sliced Binary Normal Basis Multiplication

Billy Bob Brumley
Department of Information and Computer Science
Aalto University School of Science
Espoo, Finland
Email: billy.brumley@aalto.fi

Dan Page
Department of Computer Science
University of Bristol
Bristol, UK
Email: page@cs.bris.ac.uk

***Abstract*—The performance of many cryptographic primitives is reliant on efficient algorithms and implementation techniques for arithmetic in binary fields. While dedicated hardware support for said arithmetic is an emerging trend, the study of software-only implementation techniques remains important for legacy or non-equipped processors. One such technique is that of software-based bit-slicing. In the context of binary fields, this is an interesting option since there is extensive previous work on bit-oriented designs for arithmetic in hardware; such designs are intuitively well suited to bit-slicing in software. In this paper we harness previous work, using it to investigate bit-sliced, software-only implementation arithmetic for binary fields, over a range of practical field sizes and using a normal basis representation. We apply our results to demonstrate significant performance improvements for a stream cipher, and over the frequently employed Ning-Yin approach to normal basis implementation in software.**

***Index Terms*—Computer arithmetic, Computations in finite fields, Algorithm design and analysis, Data encryption**

I. Introduction

Focusing exclusively on characteristic two, i.e., $\mathbb{F}_{2^m}$ for some integer m, finite fields represent fundamentally important building blocks within applications such as coding theory (e.g., Reed-Solomon codes) and cryptography (e.g., elliptic curves). Efficient implementation is particularly important in cryptography, which is often deployed in high-volume or high-throughput applications; examples include encryption of VPN traffic and full-disk encryption, both vital components in modern e-commerce. In this context, characteristic two finite fields are often an attractive choice versus alternatives: they afford highly efficient realisation in hardware, and even in software offer relatively low-latency field addition and squaring. Indeed, it is increasingly common to see hardware support for characteristic two finite fields within processor platforms that reflects their popularity and capitalises on their unique properties. Intel provide the most visible example of this, having announced the inclusion of a "carry-less multiplication" instruction called `PCLMULQDQ` in next-generation processor designs. This last trend is exciting for at least two reasons. Firstly, it provides a new processor feature that implementations can take advantage of in interesting ways. This, in turn, highlights the increased importance of cryptography (alongside more traditionally well supported applications such as media) in processor and arithmetic design; additional examples include the Intel AES-NI instruction set. Secondly, it hints at a need for continued research into implementation techniques that do *not* use said feature. That is, a wide range of legacy and non-Intel processor designs will potentially be under-equipped to deal with increased use of $\mathbb{F}_{2^m}$ driven by the popularity and ubiquity of Intel-led innovations such as `PCLMULQDQ`.

This paper addresses a specific question within said context, filling a gap in the literature by exploring bit-sliced software implementation of arithmetic for $\mathbb{F}_{2^m}$ using normal basis representation. Our main contribution is the evaluation of existing normal basis multiplier algorithms, and their application within cryptography, when realised as bit-sliced software implementations. The focus on multiplication is a natural choice due to the dominance of this operation in practical applications. Three main features are highlighted. Firstly, one can use the results to improve the performance of existing primitives; we focus on WG (an eSTREAM candidate stream cipher) as an example. Secondly, one can use the results to improve the performance of field multiplication w.r.t. normal bases over that of the frequently employed Ning-Yin approach. Thirdly, we show that for some field sizes, i.e., some m, and on some platforms, the performance margin between polynomial basis and normal basis field multiplication is much smaller than accepted theory suggests *if* one adopts bit-slicing in the latter case.

Supported in part by the European Commission's Seventh Framework Programme (FP7) under contract number ICT-2007-216499 (CACE).

II. Preliminaries

In this section, we briefly review the concept and applications of bit-slicing.

1) Concept: Considering a processor with a w-bit word size, let x_i denote the i-th bit of a word x where i is termed the index of the bit. Such a processor operates natively on word sized operands. For example, with a single operation one might perform addition of w-bit operands x and y to produce $r = x + y$, or component-wise XOR to produce $r_i = x_i \oplus y_i$ for all $0 \leq i < w$. This ability is restricted, however, when an algorithm is required to perform some operation involving different bits from the same word. For example, one might be required to combine x_i and x_j, where $i \neq j$, using an XOR operation in order to compute the parity of x. In this situation, one is required to shift (and potentially mask) the bits so they are aligned at the same index ready for combination via

1063-6889/11 $26.00 © 2011 IEEE
DOI 10.1109/ARITH.2011.36

a native, component-wise XOR. The technique of bit-slicing offers a way to reduce the associated overhead. Instead of representing the w-bit operand x as one word, we represent x using w words where word i contains x_i aligned at the same fixed index j. As such, there is no need to align bits ready for use in a component-wise XOR operation. Additionally, since native word oriented logical operations in the processor operate on all w bits in parallel, one can pack w different operands (say $x[k]$ for $0 \leq k < w$) into the w words and proceed using an analogy of SIMD style parallelism.

2) Applications: Performance was, and still is, a major motivation for use of bit-sliced software implementation. Such performance advantages are often amplified when a processor has a large word size, i.e., large w, as is often the case for those equipped with SIMD instruction sets, e.g., SSE. Either way, it permits efficient implementation of algorithms that are already of a bit-oriented nature, or can be reformulated to suit. Among the first effective uses of bit-sliced implementation was the work of Biham [1] who used it to implement the DES block cipher. Various researchers have realised bit-sliced implementations of AES [2], [3], [4]. Matsui and Nakajima investigated bit-sliced implementation of the KASUMI block cipher using SSE [5]. Bernstein has studied the implementation of bit-sliced polynomial arithmetic in characteristic two, with application to Edwards-form elliptic curves [6]. Beyond pure performance, one can identify another more subtle advantage of bit-slicing. By, for example, eliminating potentially very large tables used to represent S-box content, a typical bit-sliced implementation will have a smaller data memory footprint; despite the fact that the code memory footprint may slightly increase, the overall effect is usually a net reduction in memory use. Furthermore, elimination of such tables also eliminates the need to execute instructions that access them. Depending on the exact memory hierarchy, this can result in (more) predictable, data-independent execution and thus prevent (or limit) some cache-based side-channel attacks.

III. Normal Basis Multipliers

In this section, we outline a number of existing algorithms for performing multiplication in $\mathbb{F}_{2^m}$ w.r.t. a normal basis; in each case, the goal is to compute the product $C = A \cdot B$ for $A, B, C \in \mathbb{F}_{2^m}$. Our aim in doing so is to highlight both the theoretical *and* practical advantages and disadvantages of each algorithm, and use this to guide and explain our implementation and experimental results in Sec. IV. The overarching feature is the hardware-oriented nature of the algorithms: they naturally operate on bits that represent elements of $\mathbb{F}_2$, and are often even formulated in terms of gates and circuits. This highlights the fact that they are intuitively well suited to bit-sliced software implementation.

A. The Massey-Omura Multiplier

Massey and Omura [7] describe a bit-serial multiplier; the following version appears in many standards such as IEEE P1363. Assume m is odd and denote $p = Tm + 1$ where T is the normal basis type, and $u \in \mathbb{Z}_p^{\times}$ such that $\text{ord}_p(u) = T$. Define the sequence

$$F(2^i u^j \bmod p) = i,\ 0 \leq i < m,\ 0 \leq j < T. \tag{1}$$

One computes coefficient c_0 of the product using

$$c_0 = \sum_{i=1}^{p-2} a_{F(i+1)} b_{F(i)}. \tag{2}$$

To obtain successive coefficients, the operands A and B are simply rotated; that is, for coefficient c_i one uses (2) with inputs $A \lll i$ and $B \lll i$.

B. The "Matrix" Multiplier

Denote the $(m \times m)$-element multiplication matrix, with elements in $\mathbb{F}_2$, by M. The terms of (2) correspond to non-zero elements of M, thus M can be obtained from the F sequence (1) [8, 40.1.9]. Assuming m is odd, start with the zero matrix and add one to $M_{F(p-i),F(i+1)}$ for $1 \leq i < p-1$. With M realised, one computes coefficient c_0 of the product using $c_0 = A \cdot M \cdot B^T$ where the superscript T denotes transposition. This amounts to computing

$$c_0 = \sum_{i=0}^{m-1} a_i \sum_{j=0}^{m-1} M_{i,j} b_j \tag{3}$$

and again rotating operands A and B to obtain successive coefficients c_i. When considering (3), note that each row of M has at most T non-zero entries [9, Lemma 3]. As an aside, we note that this algorithm can also be realised in a more software-oriented fashion. Ning and Yin [10] compute w digits of (3) in parallel by first pre-computing all rotations of A and B in a clever manner, requiring storage for only m words per operand. The resulting algorithm, operating on words (i.e., vectors) rather than bits, is much more natural for a software implementation. We consider this the canonical method in software and use it as the base-line method for our results in Sec. IV.

C. The XESMPO and AESMPO Multipliers

The Massey-Omura and "matrix" multipliers described above are sometimes referred to as Sequential Multipliers with Serial Output (SMSO): in a given clock cycle, a single coefficient of the output is generated in a serial process repeated for m clock cycles in total. In contrast, a Sequential Multiplier with Parallel Output (SMPO) accumulates the m coefficients in parallel during each of the m clock cycles. Reyhani-Masoleh and Hasan [11] describe two examples of SMPOs, namely XOR efficient (denoted XE, with $g = 1$) and AND efficient (denoted AE, with $g = 0$). For simplicity, assume m is odd. They find an expression for the product C as

$$C = \sum_{i=0}^{m-1} \left(a_{i-g} b_{i-g} \beta + \sum_{j=1}^{(m-1)/2} z_{ij} \delta_j \right)^{2^i} \tag{4}$$

where $\delta_j = \beta^{1+2^j} \in \mathbb{F}_{2^m}$ and

$$z_{ij} = \begin{cases} (a_i + a_{i+j})(b_i + b_{i+j}) & \text{if } g = 0 \\ a_i b_{i+j} + a_{i+j} b_i & \text{if } g = 1 \end{cases}$$

and all subscripts are taken modulo m. To realise this in an efficient manner, they fix the index $i = m - 1$ of (4), initialise $C = 0$, and in each clock cycle right-rotate the operands C, A, and B, accumulating the result in C.

D. The Kwon et al. Multiplier

Kwon et al. [12] describe a multiplier for m odd with similar complexity as those of Reyhani-Masoleh and Hasan [11]. The idea begins with an $(m \times m)$-element matrix X with row i corresponding to coefficient i from (3). For instance, the first row is $(X_{0,0}, X_{0,1}, \ldots, X_{0,m-1})$ where $X_{0,i} = a_i \sum_{j=0}^{m-1} M_{i,j} b_j$ and the remaining rows are calculated as in (3) using successive rotations. Then the product is $C = \sum_{i=0}^{m-1} X_i$ transposed, where the X_i are column vectors. They seek to reduce the cost of this sum by reusing partial sums. They accomplish this by first cleverly permuting X into a matrix Y such that, for $m - 1$ out of m elements in a row or column, the inner sum of (3) in $Y_{i,j}$ is the same as that of $Y_{i+k,j+k}$ for some k; that is, the diagonal elements contain $(m - 1)/2$ repeated partial sums. To realise a multiplier, they then fix this diagonal formula, initialise $C = 0$, and accumulate the product in C using successive rotations of A, B, and C. For explicit details on the aforementioned permutation and multiplier architecture, we refer the reader to [12].

E. The Fan-Hasan Multiplier

In a parallel architecture, all of the previously mentioned multipliers require $O(m^2)$ gates to implement. Fan and Hasan [13] give a method to construct parallel multipliers for Type-1 and Type-2 ONB using an $O(m^{\lg 3})$, i.e., subquadratic, number of gates. They employ a change-of-basis implemented with a simple permutation, then compute products w.r.t. this basis using a matrix-vector product. For Type-2 ONB, the matrix decomposes into the sum of two Toeplitz matrices. They then break down each Toeplitz matrix-vector product using a divide-and-conquer approach, and finally output the sum of the result using another change-of-basis permutation. For a more detailed description, we refer the reader to [13].

IV. Evaluation

In this section, we evaluate each hardware-oriented multiplier algorithm described in Sec. III when realised as a bit-sliced software implementation. By doing so, we hope to improve on the state-of-the-art (specifically the Ning-Yin approach) for field multiplication w.r.t. normal bases, and narrow the traditionally large gap between normal and polynomial basis approaches. To evaluate each multiplier, we used a range of experimental platforms. Our rationale for including so many was to exercise various trade-offs within both the multiplier algorithm and implementation via different processor architectures and microarchitectures. Additionally we wanted to explore non-Intel processors as motivated by Sec. I. We begin by outlining our experimental platforms, discussing implementation considerations, and then describe the results themselves.

A. Experimental Platforms

A brief overview of our experimental platforms, roughly in order of computational ability and detailing only pertinent features, is as follows:

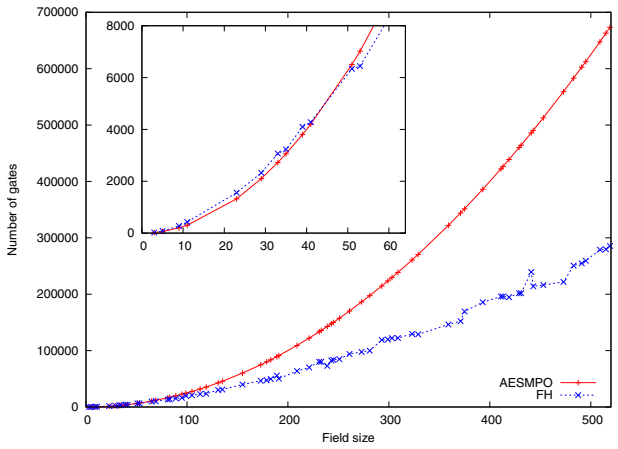
1) An Intel Core2 Duo (Conroe) processor, clocked at 2.4 GHz and based on the Core microarchitecture. It has a 64-bit scalar word size and 128-bit vector word size via SSE-based SIMD instruction set (upto SSSE3). There is a 32 KB L1 I-cache, 32 KB L1 D-cache and 4 MB L2 cache onboard.
2) An AMD Turion 64 X2 (Taylor) processor, clocked at 1.6 GHz. It has a 64-bit scalar word size and 128-bit vector word size via SSE-based SIMD instruction set (upto SSE3). There is a 64 KB L1 I-cache, 64 KB L1 D-cache and 256 KB L2 cache onboard.
3) An Intel Pentium 4 HT (Northwood) processor, clocked at 2.4 GHz and based on the NetBurst microarchitecture. It has a 32-bit scalar word size and 128-bit vector word size via SSE-based SIMD instruction set (upto SSE2). There is a 12 Kμop trace cache, 8 KB L1 D-cache and 512 KB L2 cache onboard.
4) An Intel Atom (Diamondville) processor, clocked at 1.6 GHz. It has a 32-bit scalar word size and 128-bit vector word size via SSE-based SIMD instruction set (upto SSSE3). There is a 32 KB L1 I-cache, 24 KB L1 D-cache and 512 KB L2 cache onboard.
5) An ARM ARM7TDMI processor (simulated via ADS 1.2), clocked at 7 MHz. It has a 32-bit scalar word size. There is no I-cache or D-cache onboard.
6) An Atmel AVR (ATmega128) processor, clocked at 16 MHz. It has an 8-bit scalar word size. There is no I-cache or D-cache onboard.

In all cases except the ARM7TDMI platform, we used vanilla C (with compiler intrinsics for SSE where appropriate) compiled using GCC 4.1.2; for the ARM7TDMI platform we used the ARM C compiler that forms part of ADS 1.2.

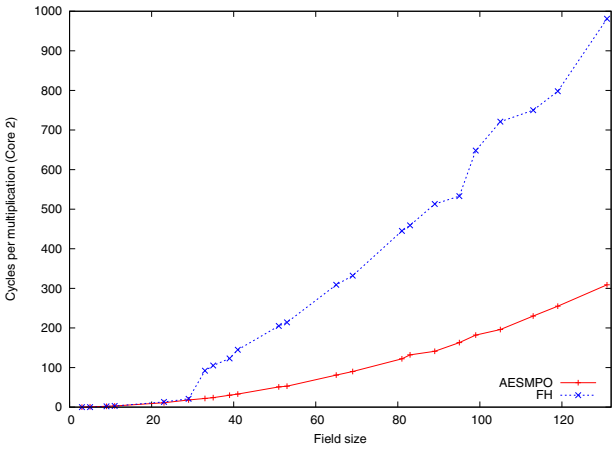
B. Implementation Considerations

The difference in trade-offs implied by implementation of traditionally hardware-oriented algorithms in software demands further investigation and careful consideration, given in the remainder of this subsection.

1) Code Footprint: The code footprint of a software implementation is essentially the number of static instructions within the program image. To explore how this impacts on the efficiency of said implementation, we investigated the Fan-Hasan multiplier in more detail. Our conjecture was that the low asymptotic gate complexity of Fan-Hasan multipliers in hardware should imply a low number of instructions executed dynamically, and hence low latency, in software. Fig. 1a shows the number of gates (i.e., the sum of XOR/AND counts) for our implementation of Fan-Hasan multipliers using odd



(a) Bit-parallel gate counts compared, AESMPO and Fan-Hasan Type-2 multipliers.



(b) Intel Core2 Duo cycles compared, AESMPO and Fan-Hasan Type-2 multipliers.

Fig. 1: A comparison of AESMPO and Fan-Hasan Type-2 multiplier implementations, demonstrating that low gate count in hardware does not imply high performance in software.

Type-2 ONB with $3 \leq m \leq 519$. For comparison, we also include an upper bound (i.e., $f(m) = 2.5m^2$) for the number of gates required by an AESMPO multiplier of the same size. This result clearly shows that for practical m, one can construct Fan-Hasan multipliers with a significantly lower number of gates than AESMPO. To assess whether this advantage translates, we constructed bit-sliced software implementations of the two multipliers. The outstanding feature of these implementations was the large code footprint required by the Fan-Hasan multiplier due to the forced adoption of a parallel approach to generating coefficients. In contrast, one can iterate over a smaller code footprint in a sequential manner for AESMPO. Fig. 1b compares the latency of the implementations and shows that Fan-Hasan multipliers pay a disproportionately large penalty for their large code footprint beyond the $m = 29$ threshold. Consequently, in the remainder of this paper we omit results for the Fan-Hasan multiplier.

2) Instruction Mix: Fig. 2 explores the instruction mix within bit-sliced software implementations of the multipliers on the Intel Core2 Duo platform. The most obvious feature is that multipliers with a higher histogram have to execute more instructions: one might intuitively assume these have higher latency therefore. This assumption is slightly skewed by the cost of said instruction execution however. The second feature is that the multipliers exhibit somewhat different mixes of instruction class: results for the XESMPO and AESMPO multipliers reflect their designed difference in XOR and AND use, while the Massey-Omura and Kwon et al. multipliers exhibits a generally higher proportion of memory access instructions because of their inability to retain more temporary values within the working register set.

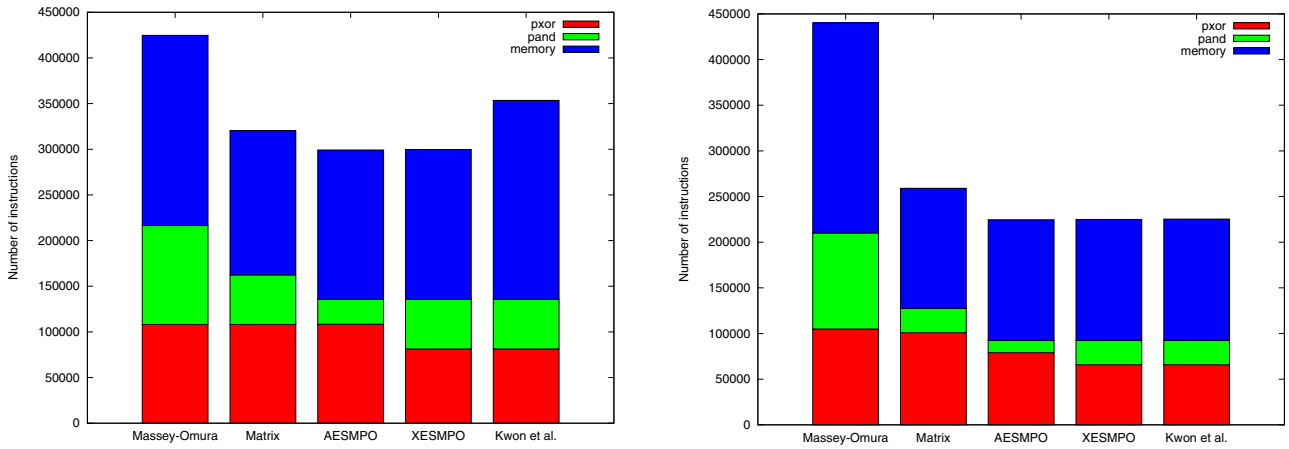
C. Summary and Comparison of Results

To assess performance in terms of latency, we compared bit-sliced software implementations of the previously described multipliers against two standard baseline methods for a range of practical m:

1) To represent the state-of-the-art for normal basis multiplication in software, we generated unrolled implementations of the Ning-Yin [10] approach for all the field sizes under consideration.
2) To represent the state-of-the-art for polynomial basis multiplication in software, we used the benchmark tuning utility in $\texttt{mp}\mathbb{F}_q$ [14] to obtain timings for multiplying two m-bit polynomials in $\mathbb{F}_2[x]$; results do not include a final reduction step. $\texttt{mp}\mathbb{F}_q$ can generate code for a suite of approaches (schoolbook, windowed combing, Karatsuba, etc.) and parameters for a given m, automatically finding the best performing choice for a given platform.

In the case of Ning-Yin and $\texttt{mp}\mathbb{F}_q$, each atomic operation represents computation of the product $C = A \cdot B$ for $A, B, C \in \mathbb{F}_{2^m}$. In the case of bit-slicing, each atomic operation is on a "batch" of w packed operands, representing computation of the w products $C_i = A_i \cdot B_i$ for $A_i, B_i, C_i \in \mathbb{F}_{2^m}$ and $0 \leq i < w$. As such, the results for bit-slicing are scaled so as to compare the per-multiplication latency of all approaches, but also include data conversion to and from bit-slice representation. Note that the batched restriction of bit-slicing in this context means Ning-Yin and $\texttt{mp}\mathbb{F}_q$ have a clear advantage when it comes to single, "one off" multiplications. However, in Sec. V we describe how batched operations can be used in practical applications, and therefore why this comparison is valid. A summary of the results of said comparison, per-platform, is as follows:

1) The Intel Core2 closely reflects the instruction counts shown in Fig. 2; it has a slight preference to XESMPO over AESMPO for Type-2 multipliers and to the Kwon et al. multiplier for Type-4. We lower latency for all Type-2 multipliers compared to polynomial bases for



(a) Type-2 multipliers for $m = 233$.

(b) Type-4 multipliers for $m = 163$.

Fig. 2: A comparison of instruction mix for various multipliers using two fixed field sizes.

field sizes below $m = 53$, and ratios of 1.2 and 1.3 at $m = 65$ and $m = 131$, respectively. We achieved a hefty 11.2-fold average speed-up over Ning-Yin.

2) The AMD Turion shares similar results with the Core2 for Type-4 multipliers. It prefers the "matrix" method for Type-2 fields, outperforming polynomial bases below $m = 35$ and ratios of 1.1 and 1.4 at $m = 65$ and $m = 131$, respectively. We achieved a large 6.4-fold average speed-up over Ning-Yin.
3) The Intel Pentium 4 does not have a clear preference for all field sizes or types. The choice of the "matrix" algorithm or that of Kwon et al. depends on the field size. In contrast to the previous two platforms, the SMPO multipliers perform poorly on the Pentium 4. For Type-2 multipliers, we outperform polynomial bases below a surprising $m = 81$ and at $m \in \{131, 135\}$ as well, with ratios of 1.1, 1.2, and 1.3 at $m = 155$, $m = 173$, and $m = 191$, respectively. The performance ratio never exceeds 1.6. We achieved a large 5.7-fold average speed-up over Ning-Yin.
4) The Intel Atom multiplier results are fairly similar to those of the Core2. We cannot explain the oscillating behavior of $\texttt{mp}\mathbb{F}_q$ on this platform. Nevertheless, with the exception of $m = 113$ we outperform polynomial bases below a sizable $m = 131$ for Type-2 fields. The performance ratio never reaches 2.0. We achieved a 13.3-fold average speed-up over Ning-Yin, the most substantial out of all platforms under consideration.
5) The ARM ARM7TDMI results also strongly reflect the instruction counts from Fig. 2, preferring the SMPOs for both field types. Although more restrained in contrast to the previous platforms due to the smaller registers, we still achieved a respectable 1.8-fold average speed-up over Ning-Yin. We use the $m = 163$ result [15] for polynomial basis multiplication as a single reference point.

TABLE I: Results in thousands of cycles, scalar multiplication for elliptic curve cryptography

m	core2	amd64	p4	atom	arm	atmega128
131	255	410	678	430	4598	25143
173	579	961	1600	1040	10572	56580
179	644	1075	1738	1146	11635	63083
191	798	1337	2241	1414	14319	76507
233	1417	2333	3855	2607	25584	139032
239	1532	2521	4133	2669	27501	149848
251	1873	2979	4780	3107	31962	174027
139	511	800	1013	751	8801	46784
163	859	1287	1631	1213	5421	77509
193	1434	2103	2861	2016	23481	134299
199	1602	2313	3417	2215	26032	150572

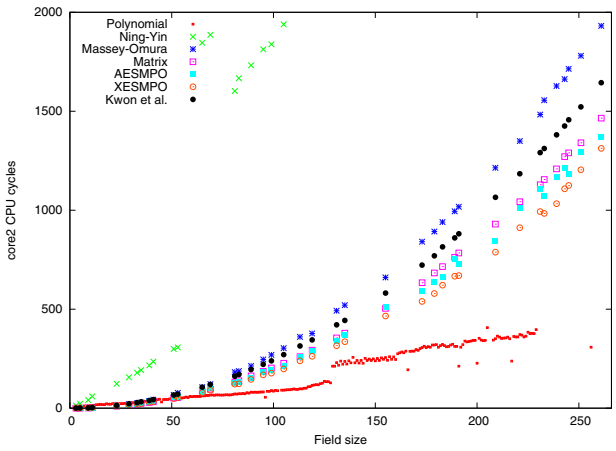
6) The Atmel AVR (ATmega128) has a strong preference to the SMPOs, generously rewarding multipliers with lower XOR/AND operation counts. We achieved a respectable 1.6-fold average speed-up over Ning-Yin. We use the $m = 163$ result [16] for polynomial basis multiplication as a single reference point.

V. Applications

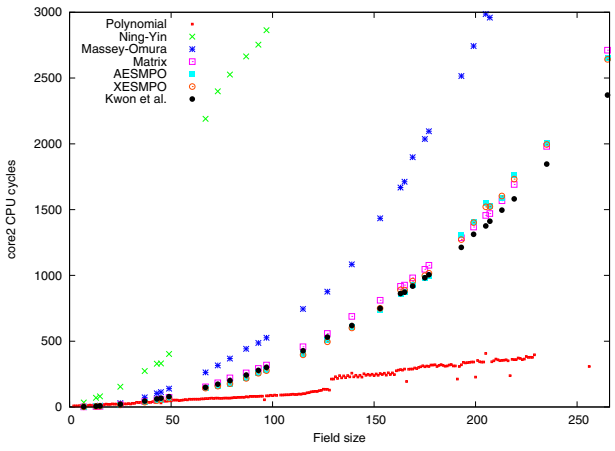
There are numerous practical applications in which the results from Sec. IV can be applied. To demonstrate this, we give two examples: one within asymmetric (i.e., public key) cryptography and the other within symmetric (i.e., secret key) cryptography.

A. Elliptic Curve Cryptography

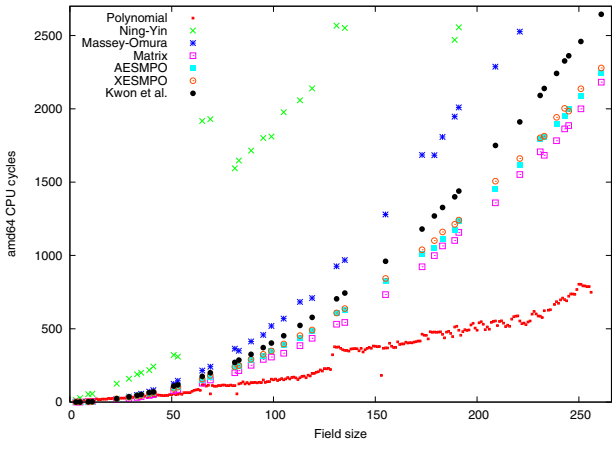
We implemented bit-sliced elliptic curve scalar multiplication using the "Montgomery ladder" [17] and López-Dahab projective coordinates [18]. As noted in Sec. IV, use of bit-slicing demands a "batched" approach to computation. As such, and following [6], our implementation computes the w



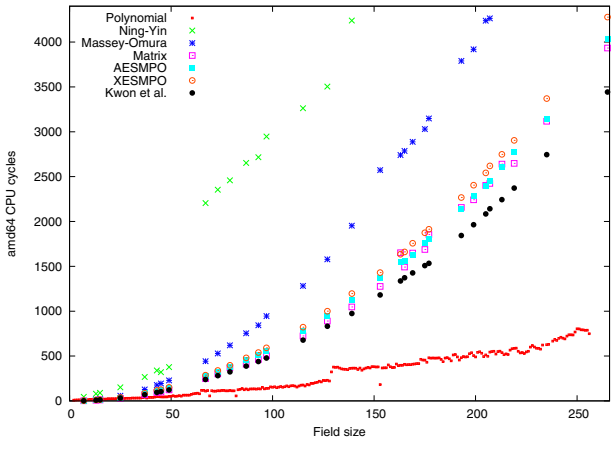
(a) Intel Core 2 Duo, Type-2 fields.



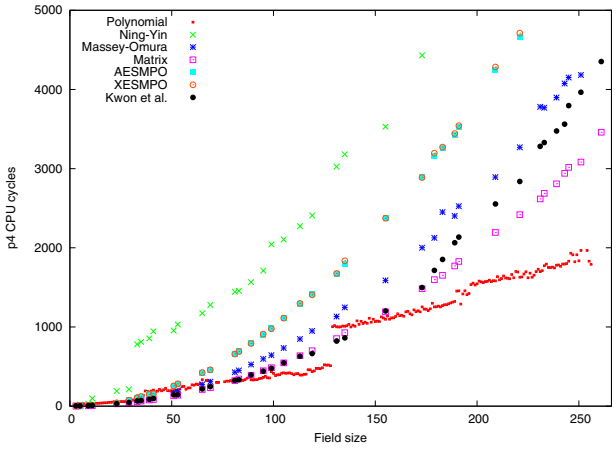
(b) Intel Core 2 Duo, Type-4 fields.



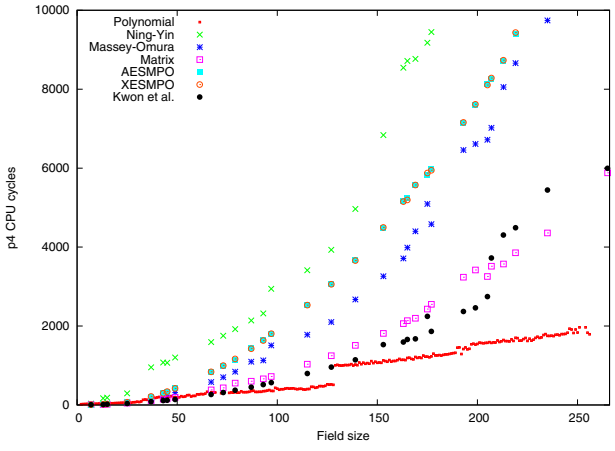
(c) AMD Turion, Type-2 fields.



(d) AMD Turion, Type-4 fields.

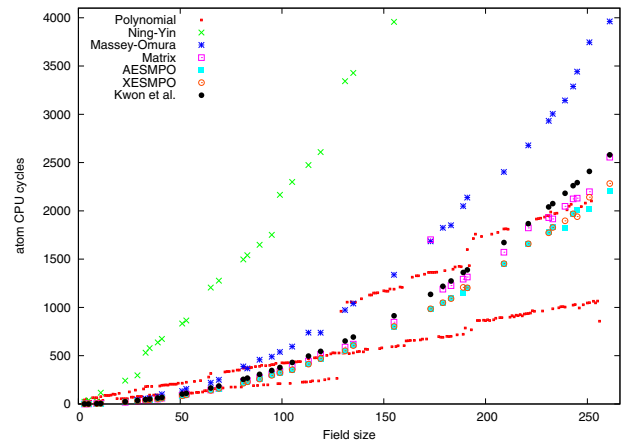


(e) Intel Pentium 4, Type-2 fields.

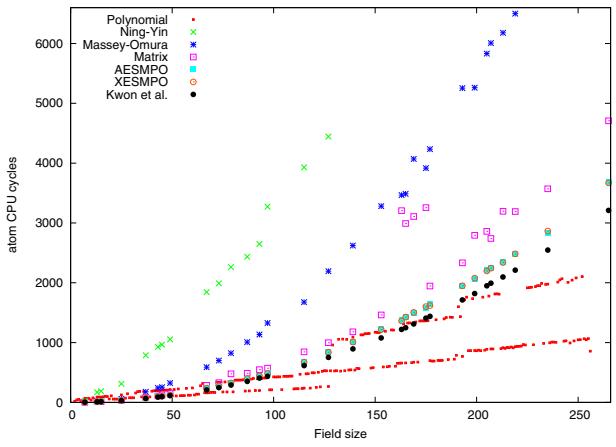


(f) Intel Pentium 4, Type-4 fields.

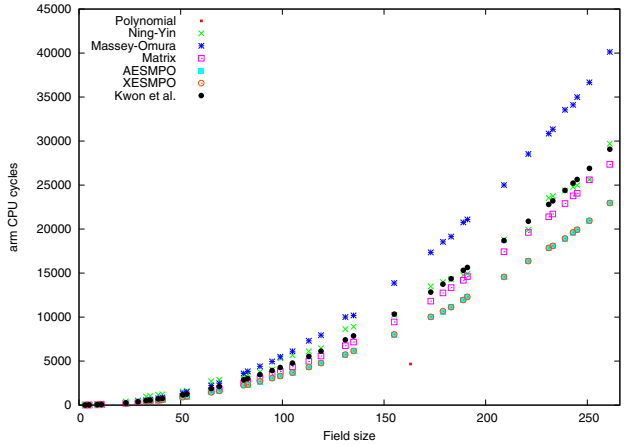
Fig. 3: A performance comparison of bit-sliced software implementations on various (largely unconstrained) experimental platforms.



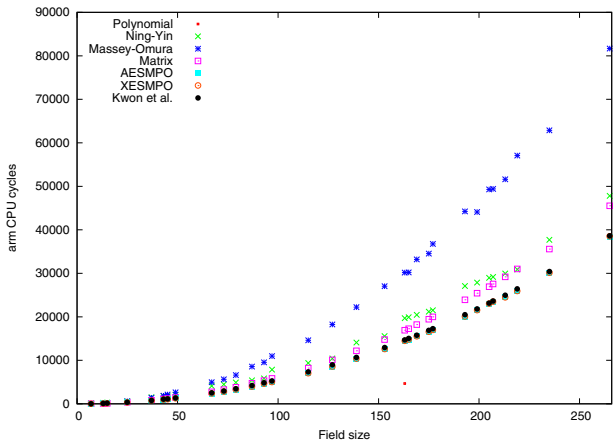
(a) Intel Atom, Type-2 fields.



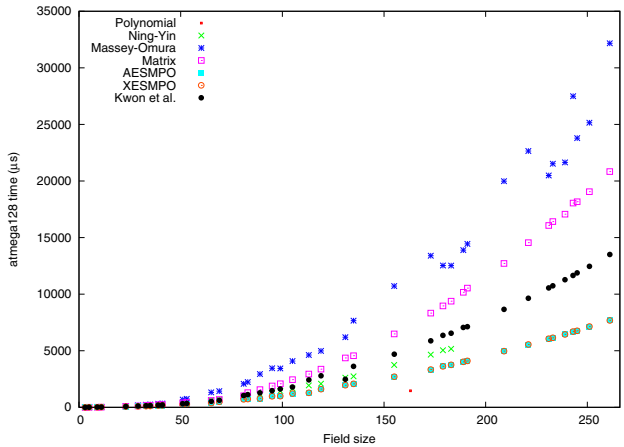
(b) Intel Atom, Type-4 fields.



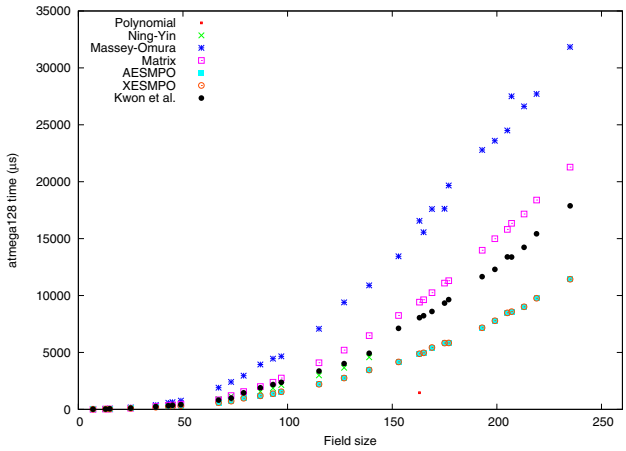
(c) ARM ARM7TDMI, Type-2 fields.



(d) ARM ARM7TDMI, Type-4 fields.



(e) Atmel ATmega128, Type-2 fields.



(f) Atmel ATmega128, Type-4 fields.

Fig. 4: A performance comparison of bit-sliced software implementations on various (fairly constrained) experimental platforms.

TABLE II: Results, WG stream cipher in software. "Serial" and "Bit-sliced" columns give cycles per keystream byte.

Architecture	Serial	Mbps	Bit-sliced	Mbps	Speedup
core2	22116	1.0	1934	11.8	11.5×
amd64	17520	0.7	2656	4.8	6.6×
p4	29967	0.6	4832	4.0	6.2×
atom	33157	0.4	2987	4.3	11.1×
arm	61326	—	33523	—	1.8×
atmega128	787904	—	147898	—	5.3×

scalar multiplications $Q_i = k_i \cdot P_i$ for $0 \leq i < w$. We motivate the use of such an implementation by the heavy workload demanded of modern web-servers: it is eminently feasible that under high-load, such a server could take advantage of batching together w operations relating to w transactions. Following [6], we realise the conditional control-flow based on key bits by using a conditional swap on the operands. Table I outlines the results, *including* the overhead of pre- and post-operation bit-slice conversion. Considering the expected number of multiplications needed, these timings are in-line with the results from the previous section.

B. The WG Stream Cipher

We implemented the WG stream cipher by Nawaz and Gong [19]. The cipher uses arithmetic in $\mathbb{F}_{2^{29}}$ including field multiplications and traces. For comparison to a serial implementation, we used the existing eSTREAM submission for WG, which uses the Ning-Yin method for field multiplication. Focusing on the field arithmetic, this existing code is practically identical to what we generated in Sec. IV-C for comparison to bit-sliced field multiplication. The serial version of WG computes a keystream byte b given key k. Given w keys k_i, our implementation computes the $\frac{w^2}{8}$ keystream bytes $b_{i,j}$ for $0 \leq i < w$ and $0 \leq j < \frac{w}{8}$; that is, $\frac{w}{8}$ keystream bytes under each key. The motivation is similar to that of elliptic curves, but in this case batching becomes even simpler since keystream bytes can be generated ahead of time independent of the plaintext. We give the results in Table II; they are consistent with the field arithmetic results and include keystream conversion from bit-sliced representation. This significant speedup is one example of a practical application of our results in Sec. IV-C for small fields.

VI. Conclusion

Efficient algorithms for arithmetic are a well acknowledged cornerstone of Computer Science in general, and cryptography in particular. In this paper we have investigated, and demonstrated, that considered implementation techniques can help to bridge the gap between theory and practice. Our results show that contrary to intuition, existing hardware-oriented normal basis multiplier algorithms *can* be a useful starting point for software implementations: we rely on the technique of bit-slicing, whose bit-oriented nature is an intuitive and ideal match for said algorithms. For example, in Sec. IV we demonstrated up to an 11-fold improvement in the performance of WG, and up to a 13-fold improvement in performance over the Ning-Yin method for field multiplication. More surprising however, is that the performance margin between polynomial basis and normal basis field multiplication is much smaller than accepted theory suggests *if* one adopts bit-slicing in the latter case. These results motivate further work to exploit this approach, for example by identifying other existing primitives that necessarily use normal bases, or situations where their use might yield an advantage if implemented effectively.

References

[1] E. Biham, "A fast new DES implementation in software," in *Fast Software Encryption (FSE)*, ser. LNCS, vol. 1267. Springer-Verlag, 1997, pp. 260–272.

[2] C. Rebeiro, A. D. Selvakumar, and A. S. L. Devi, "Bitslice implementation of AES," in *Cryptology and Network Security (CANS)*, ser. LNCS, vol. 4301. Springer-Verlag, 2006, pp. 203–212.

[3] R. Könighofer, "A fast and cache-timing resistant implementation of the AES," in *Cryptographers Track, RSA Conference (CT-RSA)*, ser. LNCS, vol. 4964. Springer-Verlag, 2008, pp. 187–202.

[4] E. Kasper and P. Schwabe, "Faster and timing-attack resistant AES-GCM," in *Cryptographic Hardware and Embedded Systems (CHES)*, ser. LNCS, vol. 5747. Springer-Verlag, 2009, pp. 1–17.

[5] M. Matsui and J. Nakajima, "On the power of bitslice implementation on Intel Core2 processor," in *Cryptographic Hardware and Embedded Systems (CHES)*, ser. LNCS, vol. 4727. Springer-Verlag, 2007, pp. 121–134.

[6] D. J. Bernstein, "Batch binary Edwards," in *Advances in Cryptology (CRYPTO)*, ser. LNCS, vol. 5677. Springer-Verlag, 2009, pp. 317–336.

[7] J. K. Omura and J. L. Massey, "Computational method and apparatus for finite field arithmetic," US Patent 4 587 627, 1986.

[8] J. Arndt, *Matters Computational*, Jul. 2009, online draft. [Online]. Available: http://www.jjj.de/fxt#fxtbook

[9] A. Reyhani-Masoleh, "Efficient algorithms and architectures for field multiplication using Gaussian normal bases," *IEEE Trans. Computers*, vol. 55, no. 1, pp. 34–47, 2006.

[10] P. Ning and Y. L. Yin, "Efficient software implementation for finite field multiplication in normal basis," in *ICICS*, ser. LNCS, vol. 2229. Springer, 2001, pp. 177–188.

[11] A. Reyhani-Masoleh and M. A. Hasan, "Low complexity sequential normal basis multipliers over GF(2^{m})," in *IEEE Symposium on Computer Arithmetic*. IEEE Computer Society, 2003, pp. 188–195.

[12] S. Kwon, K. Gaj, C. H. Kim, and C. P. Hong, "Efficient linear array for multiplication in GF(2^{m}) using a normal basis for elliptic curve cryptography," in *CHES*, ser. LNCS, vol. 3156. Springer, 2004, pp. 76–91.

[13] H. Fan and M. A. Hasan, "Subquadratic computational complexity schemes for extended binary field multiplication using optimal normal bases," *IEEE Trans. Computers*, vol. 56, no. 10, pp. 1435–1437, 2007.

[14] P. Gaudry and E. Thomé, "The $\mathrm{mp}\mathbb{F}_q$ library and implementing curve-based key exchanges," in *Software Performance Enhancement for Encryption and Decryption (SPEED)*, 2007, pp. 49–64, http://mpfq.gforge.inria.fr/.

[15] D. J. Park, S. G. Sim, and P. J. Lee, "Fast scalar multiplication method using change-of-basis matrix to prevent power analysis attacks on Koblitz curves," in *WISA*, ser. LNCS, vol. 2908. Springer, 2003, pp. 474–488.

[16] S. C. Seo, D.-G. Han, H. C. Kim, and S. Hong, "TinyECCK: Efficient elliptic curve cryptography implementation over GF(2^{m}) on 8-bit MICAz mote," *IEICE Transactions*, vol. 91-D, no. 5, pp. 1338–1347, 2008.

[17] P. L. Montgomery, "Speeding the Pollard and elliptic curve methods of factorization," *Math. Comp.*, vol. 48, no. 177, pp. 243–264, 1987.

[18] J. López and R. Dahab, "Fast multiplication on elliptic curves over GF(2^{m}) without precomputation," in *CHES*, ser. LNCS, vol. 1717. Springer, 1999, pp. 316–327.

[19] Y. Nawaz and G. Gong, "The WG stream cipher," eSTREAM, ECRYPT Stream Cipher Project, Report 2005/033, 2005.