Effects of Instruction-Set Extensions on an Embedded Processor: A Case Study on Elliptic-Curve Cryptography over $\mathrm{GF}(2^{\mathrm{m}})$

Sandro Bartolini, *Member*, *IEEE*, Irina Branovic,
Roberto Giorgi, *Senior Member*, *IEEE*, and Enrico Martinelli

Abstract—Elliptic-Curve cryptography (ECC) is promising for enabling information security in constrained embedded devices. In order to be efficient on a target architecture, ECCs require accurate choice/tuning of the algorithms that perform the underlying mathematical operations. This paper contributes with a cycle-level analysis of the dependencies of ECC performance from the interaction between the features of the mathematical algorithms and the actual architectural and microarchitectural features of an ARM-based Intel XScale processor. Another contribution is the cycle-level analysis of a modified ARM processor that includes a word-level finite field polynomial multiplier (poly_mul) in its data path. This extension constitutes a good trade-off between applicability in a number of contexts, the simplicity of integration within the processor, and performance. This paper points out the most advantageous mix of elliptic curve (EC) parameters both for the standard ARM-based Intel XScale platform and for the one equipped with the poly_mul unit. In particular, the latter case allows for more than 41 percent execution time reduction on the considered benchmarks. Last, this paper investigates the correlation between the possible architectural organizations of a processor equipped with poly_mul unit(s) and EC benchmark performance. For instance, only superscalar pipelines can exploit the features of out-of-order execution and only very complex organizations (for example, four way superscalar) can exploit a high number of available ALUs. Conversely, we show that there are no benefits in endowing the processor with more than one poly_mul, and we point out a possible trade-off between performance and complexity increase: A two-way in-order/out-of-order pipeline allows $+50$ percent and $+90$ percent of Instructions per Cycle (IPC), respectively. Finally, we show that there are no critical constraints on the latency and pipelining capability of the poly_mul unit for the basic EC point multiplication.

Index Terms—Cryptography, elliptic curves, instruction-set extension, processor architecture, performance evaluation, embedded systems.

1 Introduction

Elliptic-curve Cryptography (ECC) [32], [33] is becoming increasingly interesting [23] as an alternative to "standard" public-key methods. It uses shorter keys at a security level equivalent to other "traditional" public-key algorithms, which can translate into faster implementations and reduced consumption of energy and bandwidth. These features make ECC promising, especially for constrained embedded systems [21], [22], [23], [43], [44].

The choice of ECC implementation parameters (for example, curve coordinates and algorithms, finite-field representation, and arithmetic) is a nontrivial task because the number, variety, and interdependency of such parameters are much higher than in widely used public-key systems (such as RSA). Moreover, some parameter values have to be avoided for security reasons, as also pointed out by NIST recommendations [5]. One important choice is the type of underlying finite field upon which the elliptic curve operations are based. Many ECC implementations use binary finite fields $\mathrm{GF}(2^{\mathrm{m}})$ [18], [20], [22], [33] in which field elements are usually represented as binary polynomials. Several others use prime fields $\mathrm{GF}(\mathrm{p})$ [33]. Software implementations have been studied in a number of papers and books (for example, [14], [16], [15], [17]). These evaluate extensive sets of algorithms in terms of the number of operations [16] and/or actual performance on fixed platforms [14], [15], [17]. In this paper, we reconsider the most promising of these software implementations (see Section 2) and we evaluate in detail the effects of varying architectural parameters and the benefits of specific architectural support for the most time-consuming operation. Arithmetic on binary fields is not natively supported in the Instruction Set of typical embedded processors. This means that a large portion of the ECC execution time is spent in field multiplications, especially in $\mathrm{GF}(2^{\mathrm{m}})$. Recently, proposals have appeared for extending instruction sets for the support of polynomial multiplication in $\mathrm{GF}(2^{\mathrm{m}})$ [13], [19], [20], [21], [22]. In this paper, first we analyze the performance of pure-software ECC implementations on an ARM-based Intel XScale architecture that is widely used in embedded devices (for example, PDAs). Our investigation methodology is based on cycle-level simulation, which allows us to explore various architectural solutions and the performance contributions of the different processor and memory hierarchy units. We explore various alternatives for finite

• *The authors are with the Dipartimento di Ingegneria dell'Informazione, Università di Siena, via Roma 56, 53100-Siena, Italy. http://www.dii.unisi.it/~{bartolini, giorgi, enrico}.*

Manuscript received 28 July 2006; revised 14 May 2007; accepted 21 Aug. 2007; published online 4 Oct. 2007.
Recommended for acceptance by Ç. Koç.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number TC-0295-0706.
Digital Object Identifier no. 10.1109/TC.2007.70832.

field arithmetic over $\mathrm{GF}(2^{\mathrm{m}})$, coordinates, and scalar point multiplications. We evaluate the performance of several combinations and identify the fastest combination for the considered architecture. As finite field multiplication accounts for a big fraction of the overall time, we evaluate the effects of including of a 32 $\times$ 32-bit finite field polynomial multiplier into the data path of the processor and, accordingly, we add the MULGF [13] instruction to the processor for triggering the computation. This particular processor extension can be useful in a variety of contexts (for example, cryptosystems and algorithms) because it constitutes a good trade-off between simplicity, performance, and generality.

The experimental results obtained in this work highlight the fastest ECC implementation choices on an ARM-based architecture from both perspectives: pure software implementation and using a processor that natively supports word-level polynomial multiplication. In addition, the availability of "soft-core" processors on FPGAs (for example, NIOS-II [35]) permits an easy implementation of architectural support for new instructions, thus making the findings of this paper directly applicable to custom solutions. Programmers can immediately take advantage of such instruction-set extensions, for example, through compiler pragmas.

This paper is organized as follows: In Section 2, we give minimal background to understand ECC and its implementation choices. In Section 3, we describe the experimental setup and the considered benchmarks. In Section 4, the results for pure software implementation are discussed and, in Section 5, we show the benchmark results when a word-level polynomial multiplier is available and a sensitivity analysis of the possible design choices for its architectural organization. Section 6 summarizes the related work and Section 7 concludes this paper.

2 Elliptic-Curve Cryptography Design Space

Elliptic curves can be studied over many fields [32], but, for cryptographic purposes, finite fields (known as Galois fields (GFs)) are usually employed: Binary extended $\mathrm{GF}(2^{\mathrm{m}})$ or prime fields $\mathrm{GF}(\mathrm{p})$ are typically employed [17], [33]. In this paper, we will focus on the first, $\mathrm{GF}(2^{\mathrm{m}})$, as it can yield to very efficient implementations [15]. An elliptic curve over a binary finite field is the set of points (x, y) that satisfy the equation $y^2 + xy = x^3 + ax^2 + b \quad ab, x, y \in GF(2^m), \quad b \neq 0,$ together with the "point at infinity," denoted as O. The addition operation defined on the curve points allows the calculation of integer multiples of a point: Given a point $\mathrm{P} = (\mathrm{x}, \mathrm{y})$ and an integer k, [k]P (that is, the scalar multiplication operation) produces another point Q on the same curve. Scalar multiplication can be naturally implemented through repeated doubling and addition of the point P and it has security features similar to exponentiation in discrete-logarithm cryptosystems.[1] The basis of security in ECCs relies on this [k]P operation. Elliptic curve point addition and doubling can be calculated with a number of additions, multiplications, squarings, and inversions in the underlying binary finite field. For example, in the case of affine coordinates (see below in this section), EC addition and doubling are calculated as in [8]:

$$
\begin{aligned}
&P = (x_1, y_1), \quad Q = (x_2, y_2), \quad P + Q = (x_3, y_3),\\
&x_3 = \lambda^2 + \lambda + a + x_1 + x_2,\\
&y_3 = \lambda(x_1 + x_3) + x_3 + y_1,\\
&\lambda = \begin{cases} \frac{y_2 + y_1}{x_2 + x_1} & if\ P \neq Q\ (addition)\\ x_1 + \frac{y_1}{x_1} & if\ P = Q\ (doubling). \end{cases}
\end{aligned}
$$

Various efficient methods for finite field [36], [37] and elliptic curve [16], [38], [39] arithmetic have been developed. Handling finite field elements in software requires multiprecision arithmetic. When standard processors are employed, each m-bit operand is stored as an array of w-bit long words (where w typically is 8, 16, 32, or 64 bits, matching the host machine word size). $\mathrm{GF}(2^{\mathrm{m}})$ arithmetic can use many different representations of field elements. $\mathrm{GF}(2^{\mathrm{m}})$ can be viewed as a vector space of dimension m over GF(2), where its elements are binary m-tuples with respect to a certain basis. In this work, we use polynomial basis representation, which is more efficient for software implementations [15]. In this case, the addition of two $\mathrm{GF}(2^{\mathrm{m}})$ elements is done as a bitwise XOR between operands and the product of two field elements is obtained by first multiplying them as polynomials, which results in a polynomial of degree less than or equal to $2m - 1$, and then calculating the rest of the division with the irreducible polynomial of the field (*reduction*). There are different proposals for both multiplication and reduction algorithms. We consider the following methods for multiplication in $\mathrm{GF}(2^{\mathrm{m}})$:

- **Karatsuba-Ofman** [10], [11]. This approach uses a recursive divide-and-conquer approach that reduces the number of single-precision (word-size) multiplications by replacing some of them with additions, which are cheaper to execute.
- **Pencil and Paper (p&p)**. This is the modification of the classical shift-and-add multiplication method from Knuth [14]. In each iteration, two words of each operand are multiplied and then the two-word product is XORed with the intermediate result in the appropriate word positions.
- **Montgomery** [12]. Instead of computing $\mathrm{a} \cdot \mathrm{b}$ in $\mathrm{GF}(2^{\mathrm{m}})$, this approach computes $\mathrm{a} \cdot \mathrm{b} \cdot \mathrm{r}^{-1}$ in $\mathrm{GF}(2^{\mathrm{m}})$, where r is a special fixed element of $\mathrm{GF}(2^{\mathrm{m}})$. The selection of $r = x^m$ turns out to be favorable for obtaining fast implementations. However, another Montgomery multiplication by r^2 is needed to obtain the correct result $\mathrm{a} \cdot \mathrm{b}$. This method is suitable for chains of repeated multiplications (for example, exponentiation) so that the correction can be performed only once at the end.
- **Comb** [15]. This method takes advantage of some storage overhead by first computing a lookup table with the product of one multiplicand (for example, a(x)) for all polynomials v(x) of degree less than a chosen w-bit window length. Then, operand b(x) is scanned w-bits at a time.

1. Given k, it is relatively easy to compute $\mathrm{Q} = [\mathrm{k}]\mathrm{P}$, but it is very difficult and time consuming to reverse the operation, that is, finding k knowing only P and Q. Finding such an integer k is equivalent to calculating the elliptic curve discrete logarithm of Q to the base P.

Inversion is the most complex finite-field operation. We evaluated two well-known alternatives: the Almost-Inverse algorithm by Schroeppel et al. [9] and the Extended Euclidean algorithm. If field inversion is much more expensive than multiplication, which is typically the case in software implementations, it is advantageous to represent points by using projective coordinates instead of affine ones [17]. This way, inversions are traded for a number of multiplications. Various projective coordinates have been proposed and we focus on the widespread ones, as summarized in [15]: ***Standard.*** The projective point (X:Y:Z) satisfies the equation $\mathrm{Y}^2\mathrm{Z} + \mathrm{XYZ} = \mathrm{X}^3 + \mathrm{aX}^2\mathrm{Z} + \mathrm{bZ}^3$ and corresponds to the affine point (X/Z, Y/Z). The point at infinity is (0:1:0). We do not use these because they are significantly slower than the others [15], [17]. ***Jacobian*** [42]. The projective point (X:Y:Z) satisfies the equation $\mathrm{Y}^2 + \mathrm{XYZ} = \mathrm{X}^3 + \mathrm{aX}^2\mathrm{Z}^2 + \mathrm{bZ}^6$ and corresponds to the affine point $(\mathrm{X}/\mathrm{Z}^2, \mathrm{Y}/\mathrm{Z}^3)$. The point at infinity is (1:1:0). ***Lopez-Dahab*** [30]. The projective point (X:Y:Z) satisfies the equation $\mathrm{Y}^2 + \mathrm{XYZ} = \mathrm{X}^3\mathrm{Z} + \mathrm{aX}^2\mathrm{Z}^2 + \mathrm{bZ}^4$ and corresponds to the affine point $(\mathrm{X}/\mathrm{Z}, \mathrm{Y}/\mathrm{Z}^2)$. The point at infinity is (1:0:0).

It is common to perform *mixed additions*, in which one of the two points is in affine coordinates and the other is in projective coordinates. Mixed addition decreases the number of finite field operations (in particular, multiplications) needed in point addition. It can be used whenever an EC-operand is a constant known point (for example, elliptic curve base point or precalculated points) and, thus, it can be stored in affine format.

Finally, there are different efficient scalar point multiplication methods ($[k]P$) in use, among which we will focus on the following:

- **Window-NAF** [40]. The integer k (originally t bits) is expressed with $t+1$ bits as $\sum_{i=0}^{t} k_i 2^i$, where $k_i \in \{0, \pm 1\}$, $k_t \neq 0$, and no consecutive digits of k are nonzero. This NAF representation is unique. The NAF representation allows having an average of t/3 nonzero digits of k compared to t/2 in the binary representation. In each bit of k, a doubling of the point P is done, followed by an addition or a subtraction of the point if the bit is 1 or -1, respectively. The algorithm can be improved by windowing and precomputation techniques: Multiplication considers a group of w bits at a time (window) and precalculated multiples of P from a lookup table are then summed/subtracted (we use the reference algorithm provided in [15]).
- **Fixed-Base Comb** [41]. The binary representation of k is divided into w strings of d-bit length, padding k with zeros on the left if needed. The w strings are then written as rows and the columns $(\mathrm{a_{w-1}} \ldots \mathrm{a_0})$ of the resulting rectangle are processed in sequence. To accelerate the computation, the points $a_{w-1}2^{(w-1)d}P + \ldots + a_2 2^{2d}P + a_1 2^d P + a_0 P$ are calculated for all possible bit strings $(\mathrm{a_{w-1}} \ldots \mathrm{a_0})$.
- **Montgomery-[k]P**. Montgomery's results [26] indicate that the x-coordinate of P + Q can be computed from the x-coordinates of P, Q, and $\mathrm{P} - \mathrm{Q}$. The algorithm works on the input point in affine coordinates; thus, it is practically independent of the coordinates used and does not rely on point doublings/additions. López and Dahab [16] describe an optimization of the method proposed in [27]. Such optimization exploits a specific projective representation and uses both x and z coordinates for calculating the affine result more efficiently. In this paper, we refer to this latter implementation as it is has been shown to perform best [16].

TABLE 1
Main Processor Parameters Used in the Experiments

Issue width (instructions)	1 (in-order)
Instruction L1 cache	32 KB/ 32-way/32-byte block size
Data L1 cache	32 KB/ 32-way/32-byte block size
L2 cache	None
Fetch & Decode width (instructions)	1
Functional units	2 ALU, 1 int MULT/DIV
Memory latency (cycles)	24
Memory bus width (bytes)	4

3 Reference Platform, Methodology, and Benchmarks

The reference processor in this evaluation is close to the Intel XScale architecture [2], which has been adopted by major PDA manufacturers (for example, Toshiba, Fujitsu, and HP) and can be used to develop small cost-effective handheld devices. The Intel XScale includes integer, integer multiply-accumulate (MAC), and memory pipes. The pipeline issues a single instruction per cycle. Instructions are issued in order but may be executed out-of-order through a scoreboard mechanism that tracks to-be-finished instructions. Although a single instruction may be issued per clock cycle, all three pipelines (MAC, memory, and main execution) may be processing instructions simultaneously if some of them require more than one cycle to complete.

We evaluate the performance of our ECC benchmarks by using the `sim-outorder` simulator of the SimpleScalar toolset [3]. `sim-outorder` performs cycle-level timing simulation of the modeled architecture and can take into account speculative execution, branch prediction, and cache misses. SimpleScalar for the ARM architecture supports the ARM7 integer instruction set (a subset of XScale's 5TE ISA). The timing of the model has been validated against a Rebel NetWinder Developer workstation [1] by the developers of the simulator. All of the benchmarks were compiled using the arm-linux-gcc cross-compiler included in the SimpleScalar package, with -O2 optimization enabled. We modified the `sim-outorder` simulator to gather the performance metrics of each code region (function or groups of functions), aiming at isolating the contribution of the core for each benchmark. The details of the processor configuration are given in Table 1.

We used the MIRACL C library [4] as the starting point for developing our ECC benchmark set. This library consists of routines that cover multiprecision finite field arithmetic and some algorithms for EC operations. We extended the library with a number of algorithms for field multiplication, scalar EC point multiplication and coordinates support. Our implementations, where applicable, allow for exploring the

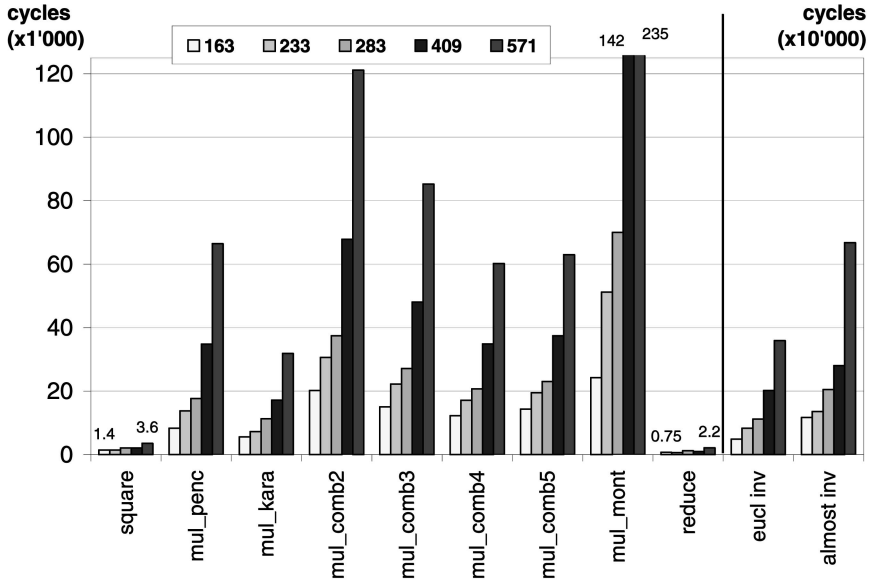


Fig. 1. Execution time of main finite field operations on our reference platform (cycles $\times$ 1,000; for inversion, $\times$ 10,000 is shown). The times for squaring ("square"), multiplication ("mul_*"), modulo reduction ("reduce"), and inversion ("*_inv$") are reported. All timings include modular reduction.

parameters of the algorithms (for example, window width). Finite fields and elliptic curves were chosen according to the NIST standard [5]: for binary polynomial fields, the suggested key sizes are 163, 233, 283, 409, and 571 bits, corresponding to increasing security levels. For each recommended field, this standard also specifies the parameters for initializing the curve (for example, the base point G of the curve) and the irreducible polynomial (usually trinomial or pentanomial). To evaluate the performance of an ECC cryptosystem, we used the following basic set of benchmarks, which comprise the typical operations performed in public-key protocols:

- Diffie-Hellman key exchange (**ecdhb**). This calculates and shares a private key with a partner according to the Diffie-Hellman scheme [6].
- Digital signature generation/verification (**ecdsignb**, **ecdsverb**). This implements the NIST DSA standard [5], calculates a 160-bit digest of a message file, signs it by using a private key, and writes the signature into a file. A verification benchmark calculates the message hash and compares it with the decrypted signature.
- ElGamal encryption/decryption (**ecelgencb**, **ecelgdecb**). This encrypts/decrypts a random message on the curve by using the ElGamal algorithm [7].

We also used a test benchmark consisting of a single scalar EC point multiplication ([k]P) for focused evaluations of various finite field and elliptic curve alternative algorithms and different coordinate systems.

4 Pure-Software Implementation and Evaluation

4.1 Finite-Field Operations

In this section, we show the execution speed of the finite field operations implemented in our library. For multiplication, we illustrate the results for the Karatsuba-Ofman [10], p&p [14], Montgomery [12], and Comb [15] algorithms. For the Comb method, we take into account various window sizes (from 2 to 5 bits). The execution time is measured in clock cycles on our reference simulation platform. Fig. 1 highlights the most advantageous multiplication and inversion algorithm for our architecture and our library for the key sizes suggested in the NIST recommendation [5]. In addition, the figure helps evaluate the relative speed of finite field operations. Timings of all operations include modular reduction.

The squaring operation is between four and nine times faster than the fastest multiplication approach (Karatsuba in Fig. 1) for key sizes going from 163 to 571 bits. The reduction operation (also shown separately) is from 7.5 to 15 times faster than the Karatsuba multiplication for the same key size range: This is mainly due to the special form of irreducible polynomials used in ECC and implies that it is not expected to be a time-critical operation. The fastest Comb multiplication (4-bit window) is roughly two times slower than the Karatsuba method for every key size. A 5-bit window is slightly slower, with a similar performance to a 4-bit one. Beyond this size, the precomputation overhead exceeds the benefits of a larger window size. The Montgomery technique is quite inefficient when implemented in software, exhibiting a slowdown of four to seven times in comparison to the Karatsuba method. Part of this originates in the need for two consecutive Montgomery multiplications to get the correct finite field result when performing a single modular multiplication. Even if we compare a single Montgomery multiplication and therefore obtain half the shown cycles, this method would achieve more or less the same performance as the Comb method and much less performance than Karatsuba, which both also comprise the cycles for reduction (Fig. 1).

Euclidean inversion is roughly 10 times slower than the Karatsuba multiplication: This is in line with existing studies [15]. The Almost-Inverse technique is slower than euclidean inversion, which is also in our library: One

TABLE 2
EC Operations in Terms of Finite-Field Operations

Coordinates	EC point add			EC mixed point add			EC point double		
	M	**I**	***T***	**M**	**I**	***T***	**M**	**I**	***T***
Affine	2	1	*2*	-	-	-	2	1	*2*
Projective Jacobian	14	-	*6*	10	-	*6*	5	-	*2*
Projective Lopez-Dahab	14	-	*7*	9	-	*3*	4	-	*2*

$\mathrm{GF}(2^{\mathrm{m}})$ *operation count and number of temporary variables (finite field elements) for EC point addition and doubling (M = Multiplications, I = Inversions, T = # of Temporary Variables).*

Almost-Inverse operation costs as much as 18 Karatsuba multiplications. For this reason, in the rest of this paper, we will focus on euclidean inversion.

For the multiplication algorithms of our library, Karatsuba-Ofman is the fastest one for every key size. In particular, Karatsuba is 1.5 to 2 times faster than our "p&p" method (the second fastest for 163-409 bits) and 1.9 times faster than Comb with a 4-bit window (the second fastest for 571 bit).

4.2 Elliptic-Curve Coordinate Systems

We have extended the MIRACL library to compare different coordinate systems: affine, Jacobian (projective) [42], and Lopez-Dahab (projective) [30] coordinates. We have substantially verified the results already present in the literature, which we are going to concisely summarize in this section. Each coordinate system uses different formulas to obtain the coordinates of the resulting point as a function of the coordinates of the operand points. According to such formulas, Table 2 (see [15]) shows the number of computationally expensive operations,[2] finite field multiplications (M), and inversions (I) for EC point addition, mixed-coordinates point addition, and doubling. Table 2 also reports the storage overhead as the maximum number of temporary variables $\mathrm{T} \in \mathrm{GF}(2^{\mathrm{m}})$ simultaneously needed during the computation. The Lopez-Dahab coordinates are the most performing, while the affine ones have the smallest memory footprint. Affine coordinates may be a viable choice in memory-constrained devices, even if GF inversion may induce long processing. In addition, it could also be favorable if specific hardware for inversion is available.

4.3 Elliptic-Curve Scalar Multiplication

In this section, we analyze the complexity of three well-known algorithms for computing EC scalar multiplication [k]P on our platform—Window-NAF [17], [15] (from a 3-bit up to a 5-bit window width), Montgomery-[k]P (projective case) [16], and Fixed-Base Comb [41] (from a 3-bit up to a 5-bit window width)—that we introduced in Section 2. The Window-NAF and Fixed-Base Comb methods rely on the precomputation of specific multiples of the point P and, therefore, we investigate on the precomputation overhead too. Table 3 analyzes these techniques in case of a 163-bit key size and, where applicable, for a 4-bit window. Table 3 illustrates the complexity of each method in terms of EC operations, finite-field $\mathrm{GF}(2^{\mathrm{m}})$ operations (corresponding to the EC operations), word-level (32-bit) polynomial multiplications, and the number of the needed temporary EC points (Pts) and GF elements (GFe).

Based on Table 3, it appears that the number of EC doublings required for calculating [k]P is similar for Window-NAF and Fixed-Base Comb (162 and 164, respectively), while the number of EC additions is 57 percent higher for the latter (52 versus 33). Excluding the precomputation overhead, Fixed-Base Comb needs 41 percent more EC additions than Window-NAF (41 versus 29) but 74 percent fewer EC doublings (41 versus 161). This advantage can be exploited whenever [k]P operations are performed on a known point P through the memorization of the precomputed values. In security protocols, it is quite common to also perform [k]P operations by using the base point G of the cryptosystem in place of P.

Montgomery-[k]P (projective case) [16] for EC scalar multiplication does not rely on basic EC operations but on lower level formulas, working mainly on the affine x-coordinate of the points. Therefore, the results are very similar for both projective and affine coordinates.

As for GF operations, Table 3 confirms (see [15]) that affine coordinates are not interesting in software implementations because 196 and 215 inversions for Window-NAF and Fixed-Base Comb, respectively, account for about 1,960 and 2,150 Karatsuba multiplications, assuming a ratio of about 10 between euclidean inversion and Karatsuba Multiplication (see Fig. 1). In order to make some comparison, we defined Equivalent Finite Field Multiplications (EFFMs) $EFFM = \lceil \frac{1}{4} \cdot s \rceil + m + 10 \cdot i$, where s, m, and i are the numbers of squarings, multiplications, and inversions, as reported in Table 2. Table 3 shows an estimation of benchmark complexity in terms of the number of EFFMs, heuristically calculated according to the results in Fig. 1. As GF squaring weights about 1/4 the time of GF multiplication for a 163-bit field size, affine coordinates require $\lceil 197\mathbf{S}/4 \rceil + 394\mathbf{M} + 196\mathbf{I}^{*}10 = \mathbf{2,404}$ EFFM for the Window-NAF method, whereas projective Lopez-Dahab coordinates require $\lceil 999\mathbf{S}/4 \rceil + 1,266\mathbf{M} + 2\mathbf{I}^{*}10 = \mathbf{1,536}$ EFFMs. Neglecting other details of the algorithms, these results suggest that affine coordinates can be more than 50 percent slower than projective ones.

Using projective coordinates, Window-NAF (wNAF) requires about 16 percent fewer GF multiplications than Fixed-Base Comb and 10 percent fewer GF squarings, which translates into an overall 15 percent fewer EFFMs. The Montgomery-[k]P method appears to be the fastest one

2. We do not consider the number of squaring operations because squaring is much faster than multiplication and the number of squarings performed in the listed EC operations is limited.

TABLE 3
Number of Temporary Variables (EC Points "Pts" and GF elements "GFe") and Operations for Different Scalar Multiplication ([k]P) Methods over $\mathrm{GF}(2^{163})$

[k]P method	Coord.		Temps Pts	Temps GFe	EC operations ADD	EC operations DBL	GF operations SQR	GF operations MUL	GF operations INV	*EFFM*	32-bit MUL
Window NAF (4-bit window) [40]	Affine	pre-computation			4	1	7	14	6		234
		computation	5	5	29	161	190	380	190		6819
		total			**33**	**162**	**197**	**394**	**196**	**2404**	**7053**
	Projective Jacobian	pre-computation			4	1	25	60	1		1031
		computation	5	9	29	161	942	1203	1		21585
		total			**33**	**162**	**967**	**1263**	**2**	**1525**	**22616**
	Projective Lopez-Dahab	pre-computation			4	1	29	63	1		972
		computation	5	9	29	161	970	1203	1		18189
		total			**33**	**162**	**999**	**1266**	**2**	**1536**	**19161**
Montgomery			0	9	n/a	n/a	**810**	**980**	**2**	**1203**	**17553**
Fixed-Base Comb (4-bit window) [41]	Affine	pre-computation			11	123	136	272	135		4072
		computation	16	3	41	41	80	160	80		3677
		total			**52**	**164**	**216**	**432**	**215**	**2636**	**7749**
	Projective Jacobian	pre-computation			11	123	667	762	1		13575
		computation	16	7	41	41	395	739	1		13266
		total			**52**	**164**	**1062**	**1501**	**2**	**1787**	**26841**
	Projective Lopez-Dahab	pre-computation			11	123	681	771	1		11205
		computation	16	7	41	41	435	738	1		11787
		total			**52**	**164**	**1116**	**1509**	**2**	**1808**	**22992**

The field multiplication method is Karatsuba. EFFM represents an estimate of the Equivalent Finite Field Multiplication operations needed to execute the benchmark.

in this comparison because it requires 22 percent fewer GF multiplications (980 versus 1263) and 16 percent fewer GF squarings (810 versus 967) compared to the best projective configuration (that is, Jacobian coordinates and wNAF).

Finally, Table 3 shows the number of word-level (32×32 bits on our platform) polynomial multiplications performed at the lower level of the library during the EC scalar multiplication. GF multiplications are decomposed into word-level multiplications and then executed in an ad hoc software function. Using wNAF, the comparison between the projective coordinate systems in terms of EFFM suggests that Jacobian is practically equivalent to Lopez-Dahab (that is, **1,525** versus **1,536**, respectively). The Montgomery-[k]P method is almost independent of the coordinate system, producing **1,203** EFFMs, which is about 21 percent fewer than the second best mix, that is, the Jacobian coordinates and wNAF.

On the other hand, Fixed-Base Comb needs about three times more temporary EC points than wNAF (16 versus 5). The latter requires at least 15 GF elements in affine coordinates: Ten because of the two coordinates of the five points, plus five more GF elements. In projective coordinates, the wNAF memory requirement grows up to 24 GF elements ($5 \cdot 3 + 9$) for both the Jacobian and Lopez-Dahab coordinates. Conversely, Montgomery-[k]P requires only nine GF elements.

Fig. 2 shows the execution time (millions of cycles) of EC scalar multiplication on our simulation platform for various sets of parameters and the Karatsuba-Ofman GF multiplication for the 163-bit key size. Affine (af-), Jacobian projective (ja-), and Lopez-Dahab projective (ld-) coordinates are shown and all of the considered EC scalar multiplication algorithms are compared. Fig. 2 highlights that the Montgomery-[k]P method is the fastest for all of the coordinate systems: It is more than two times faster in the case of affine coordinates and about 13 percent faster than the second fastest mix (the Lopez-Dahab coordinates and wNAF with a 4-bit window). Fixed-Base Comb (fbComb) is the worst performing point multiplication algorithm in our implementation, but it becomes the fastest in case of projective coordinates when neglecting the precomputation overhead (about 40 percent fewer cycles). Moreover, the fbComb computation time takes advantage of increasing the window size from 3 to 5 bits. The precomputation overhead for the wNAF method is negligible in all of the tested conditions. Both windowed methods deliver the best overall performance in the majority of tests for a 3-bit or 4-bit window size (that is, the size chosen for the results in Table 2).

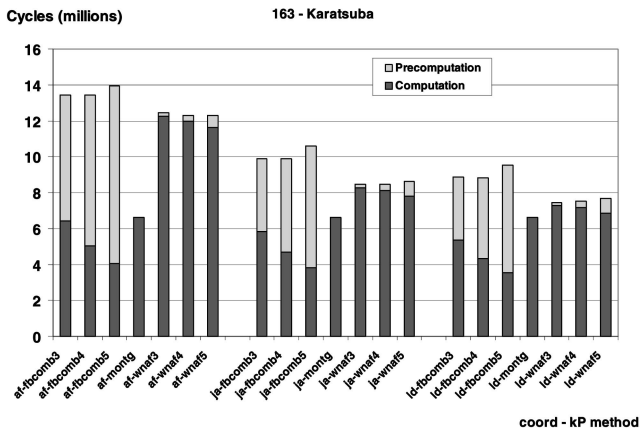


Fig. 2. Execution time of various scalar point multiplication methods, employing the Karatsuba finite-field multiplication on a 163-bit field size. The precomputation and computation quotes are shown.

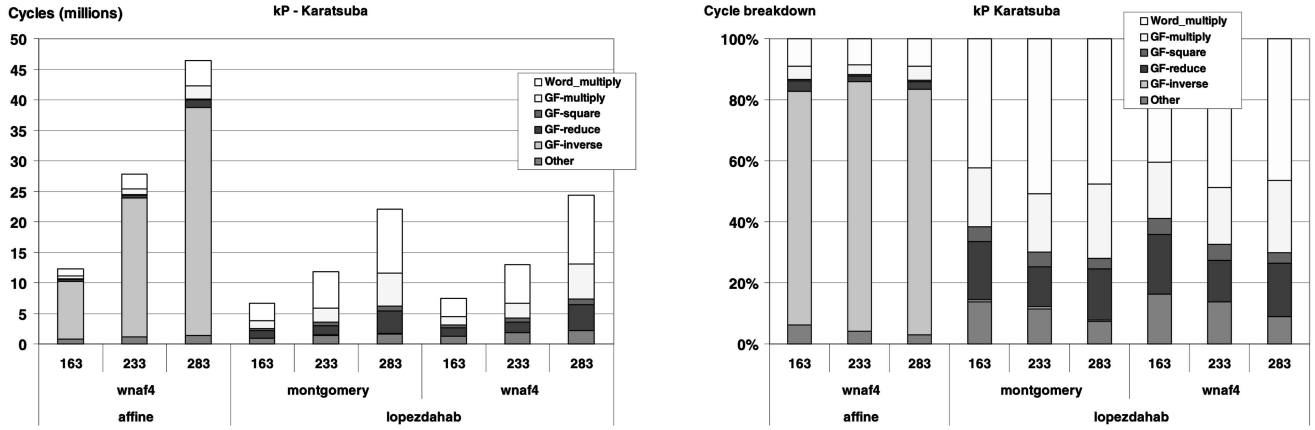


Fig. 3. Cycle breakdown of the EC scalar multiplication algorithms for the most time-consuming activities—word and field multiplication, squaring, modular reduction, inversion, and the remaining activities—in case of affine coordinates with the 4-bit wNAF algorithm (wnaf4 affine), Lopez-Dahab coordinates (wnaf4 lopezdahab) with 4-bit wNAF, and Montgomery-[k]P (Montgomery lopezdahab). In affine coordinates, the execution time is dominated by inversions, while for the Lopez-Dahab coordinates, word-level polynomial multiplication (Word-multiply) accounts for 41 percent to 50 percent of the execution time.

We carried out further experimentations on bigger key sizes. We can summarize the results as follows: For 233-bit and 283-bit cases, the execution time increases by about two and four times, respectively. Moreover, the precomputation overhead, where present, remains almost constant. When neglecting the precomputation, this latter effect lets fbComb perform more or less the same as Montgomery as the key size increases (233 to 283 bits).

Fig. 3 shows the execution time (millions of cycles) for the [k]P benchmark in the case of the two best performing mix of ECC parameters: Lopez-Dahab projective with Montgomery-[k]P and wNAF (4-bit window), respectively. For comparison purposes, affine coordinates with wNAF (a 4-bit window) are also shown. The bars break down the execution time into 32×32-bit polynomial multiplication, multiprecision GF multiplication, squaring, modulo reduction, inversion, and remaining (other) activities. Multiprecision multiplication encompasses the code for decomposing each 163-bit, 233-bit, and 283-bit multiplication into 32×32-bit word-level multiplications and for composing back the result. In the depicted case, the Karatsuba technique is employed. Other activities include temporary operand copying, zeroing, bit shifts, etc. In particular, Fig. 3 (left) shows the absolute execution time, while Fig. 3 (right) shows the normalized breakdown for an easier evaluation of the relative weight of each activity.

Fig. 3 shows that, in affine coordinates, execution is dominated (almost 80 percent) by finite field inversion, while multiplication accounts for only 10 percent to 15 percent of the time, whatever the key size. Conversely, projective coordinates trade inversions for a number of field multiplications (and squarings) and achieve about 50 percent faster execution. This way, the total multiplication activity reaches about 60 percent of the whole time for a 163-bit key size, increasing up to about 70 percent for 283 bits. In particular, the figure shows that the time for 32×32-bit polynomial multiplication takes about 41 percent to 50 percent for the considered key sizes. These differences are due to the effects of multiprecision arithmetic in the specific implementation of the Karatsuba multiplication.

Fig. 3 emphasizes that EC scalar multiplication is very sensitive to the execution efficiency of field multiplication. Moreover, Fig. 3 shows that modular reduction accounts for a 13 percent to 19 percent portion in case of projective coordinates, resulting in a nonnegligible activity. Montgomery-[k]P employs a slightly lower fraction of time than wNAF for auxiliary activities and both require a smaller relative fraction for it as the key size increases. Last, finite field squaring without reduction has a negligible weight in the overall execution.

4.4 Benchmark Results

In this section, we analyze the results of the reference benchmarks and we compare them against the ones already described for EC scalar multiplication. EC benchmarks are based on EC scalar multiplication operations, EC additions, and doublings, as well as point compression operations[3] [17] and, in some cases, integer operations too.

Each benchmark has been carefully structured at the code level and precisely profiled at runtime so that its results are strictly related to the sole core execution. No "noise" on the results is propagated by library or benchmark initialization activities. The results of the reference benchmarks reflect their usage of [k]P operation: In particular, EC digital signature (ecdsign) and El-Gamal decode (ecelgdec) need one [k]P operation, while Diffie-Hellman (ecdh), EC digital signature verification (ecdsver), and El-Gamal encode (ecelgenc) require two [k]P operations. In each benchmark, some more application-level activities are needed. In particular, more inversions are required due to point normalization and point compression operations.

Fig. 4 shows the execution time (processor cycles) comparison between the considered benchmarks for various key sizes. In particular, cycle breakdown is given for

3. Point compression extracts the X coordinate and small information from the Y coordinate of a point in order to save memory in representing curve points. This is useful in secure protocols (for example, Diffie-Hellman) to limit the bandwidth needed for transmitting points over network links [45].

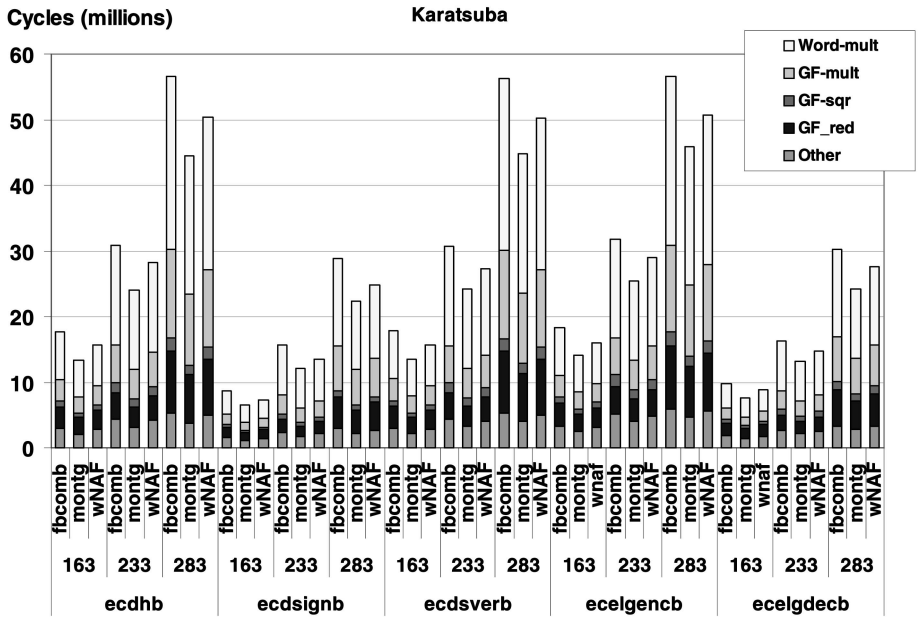


Fig. 4. Cycle breakdown of the considered benchmarks into word-level multiplication (Word-mult), finite field multiplication (GF-mult), squaring (GF-sqr), reduction (GF-red), and the remaining activities (Other). The results are given for the fbComb (4-bit window), Montgomery, and wNAF (4-bit window) [k]P methods and for 163-bit, 233-bit, and 283-bit key sizes.

word-level multiplication, finite field multiplication, squaring, reduction, and other activities. Fig. 4 confirms the performance ordering of the considered [k]P algorithms: Montgomery-[k]P is the fastest choice on our architecture, that is, on average, 13 percent lower execution time for 163 bits and about 11 percent lower for 233 and 283 bits against the wNAF method. The execution time over fbComb is 24 percent lower (163 bits) and 20 percent lower (233 and 283 bits).

Finally, we found that field multiplication takes 57 percent of the execution time (slightly less than the 60 percent that we had in Fig. 3 for a 163-bit [k]P operation). In turn, word-level multiplication takes 39 percent of the execution time (similar to the 40 percent that we found for the 163-bit [k]P operation; see Fig. 3). Moreover, as the security protocols and operations implemented in the benchmarks require the manipulation of points and scalar values, the share of *other* activities increases by about 18 percent, 15 percent, and 10 percent for 163-bit, 233-bit, and 283-bit, respectively, in comparison with the [k]P operation in Fig. 3.

5 Instruction-Set Extension Evaluation

The analysis done for the considered software implementation of the benchmarks and for the various key sizes highlights that a relevant fraction (39 percent to 48 percent) of the benchmark execution time is spent in word-level polynomial multiplication. On specific operations (for example, [k]P), this fraction grows up to 40 percent to 50 percent (Fig. 3).

In general, word-level operations can be suitable for being integrated into the existing data path of a general purpose processor because they usually do not require altering its register file (RF) structure and organization. The same is true for the processor internal buses for operands and, hopefully, for the instruction results. If the combinatorial function to be performed on the word-sized operands does not require excessive space in silicon and if the function result is compatible with the existing data path, it is viable to easily extend the processor with a functional unit that implements such word-level functions in hardware. A different situation exists for integrating functional units wider than the existing data path of a general purpose processor [24], [31]. Usually, a separate RF and return data path have to be deployed, along with a number of new instructions for moving data in and out the wide RF from/toward both memory and the existing RF (for example, Intel SSE [46]).

Word-level polynomial multiplication (32×32 bits to 63 bits in our case) is suitable for an easy noninvasive integration into the data path of an ARM general purpose processor. In fact, its blackbox interface is analogous to the one provided by integer multiplication: Two registers hold the input operands and two registers will receive the higher and the lower words of the result. The choice of this extension represents a good trade-off between the simplicity of integration, generality (that is, applicability in a number of contexts also outside ECC), and performance benefits. From the implementation point of view, polynomial multiplication can be seen as a simplification of integer multiplication when the carries in the partial product accumulation are not propagated. Therefore, the combinatorial circuit needed for its implementation is bounded in complexity by several possible implementations of a word-sized integer multiplier. Proposals also exist [22], [24], [28], [29] for dual-field multipliers. When such a new functional unit is available in a processor, its instruction set has to be augmented by a specific instruction to trigger the hardware computation.

In our test platform, we include one functional unit for a 32×32-bit polynomial multiplication (*poly-mul*) and accordingly insert a new ssembly instruction (MULGF [13]). The

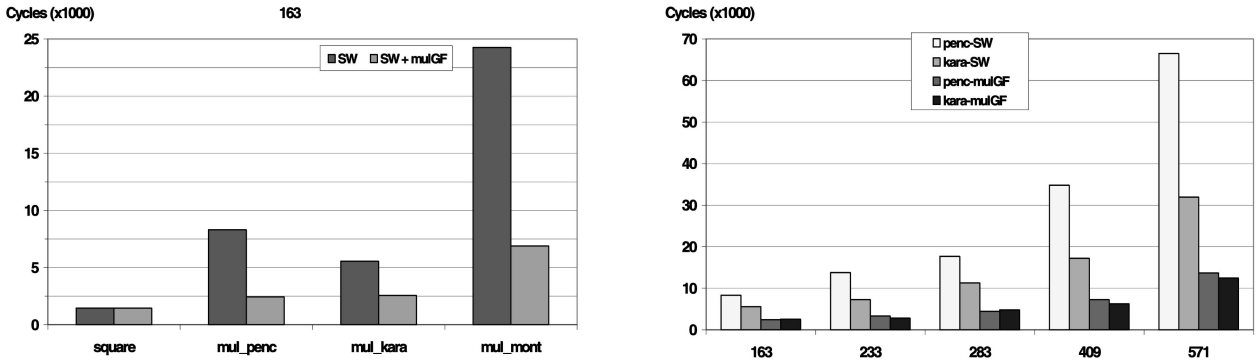


Fig. 5. Left: Execution time of GF squaring and different GF multiplication methods (p&p, Karatsuba, and Montgomery) on $\mathrm{GF}(2^{163})$ for SW and with MULGF instruction (SW + mulGF). Right: Execution time of the p&p and Karatsuba methods of SW and with SW + mulGF for various key sizes. The MULGF instruction let p&p and Karatsuba have very similar performance.

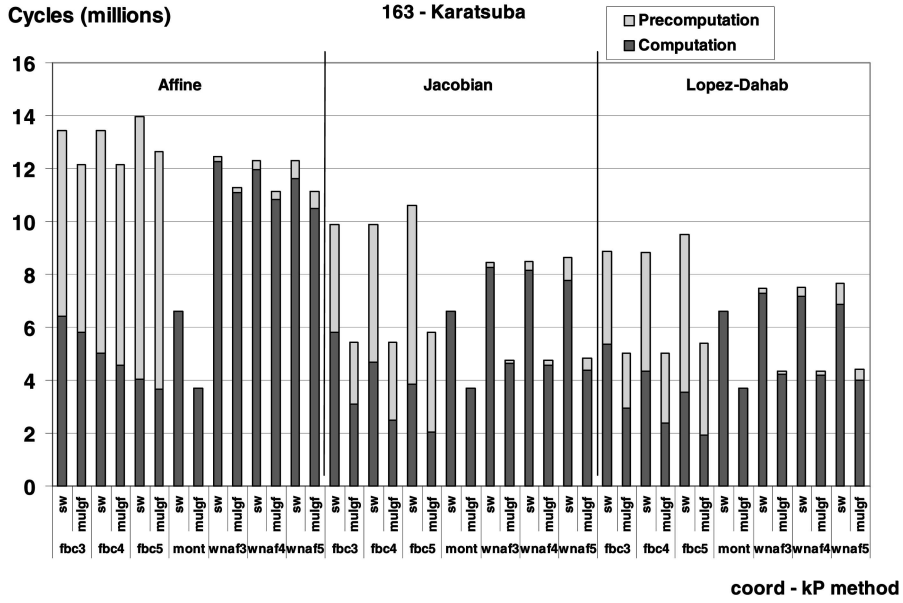


Fig. 6. The total execution time of the [k]P benchmark with a 163-bit key length, different coordinate systems, and [k]P algorithms. The results are shown for both sw and MULGF-supported (mulgf) implementations. The precomputation and computation cycles are highlighted.

cryptolibrary and the benchmarks were modified so that the original software implementation of the poly-mul could be replaced by a single MULGF instruction through assembler inlining. The GCC cross-compiler was modified to support MULGF so that GCC could manage such instructions (for example, register allocation and optimizations) in the same way as the other ones present in the ARM ISA. In particular, we modified the GNU assembler (GAS) source files that manage the target ISA. This way, the benchmarks, including MULGF invocations, could be recompiled (again using the -O2 optimization switch).

5.1 Finite-Field Operations with MULGF

In Fig. 5, we show the effects of the MULGF instruction on finite field multiplications. In particular, Fig. 5a shows that our MULGF instruction reduces the execution time of the 163-bit Karatsuba multiplication from about 5,500 cycles down to 2,550 cycles (-60 percent). Moreover, the figure highlights that the simple p&p method takes good advantage of MULGF, reducing its execution time by 70 percent, which makes its performance almost equal to Karatsuba's. The Montgomery technique also benefits significantly from MULGF, but its performance is still far from favorable. Finally, Fig. 5 (left) shows that squaring does not take advantage of the MULGF instruction. Note also that software squaring is faster than MULGF-supported multiplication.

Fig. 5b analyzes the MULGF effects on the p&p and Karatsuba techniques for different key sizes. Karatsuba is about 15 percent faster for both 233 and 409 bits and 10 percent faster for 571 bits. These results highlight that Karatsuba and p&p deliver very similar performance and the choice between them should be done according to other criteria.

5.2 [k]P Methods with MULGF

In Fig. 6, we show the performance results of the EC scalar multiplication for the same [k]P algorithms and coordinates analyzed in Fig. 2. We compare the pure software implementation without MULGF (sw) and the implementation that uses the MULGF instruction (mulgf) for the Karatsuba multiplication. [k]P results for the p&p method, which are not shown, are practically indistinguishable from the Karatsuba results.

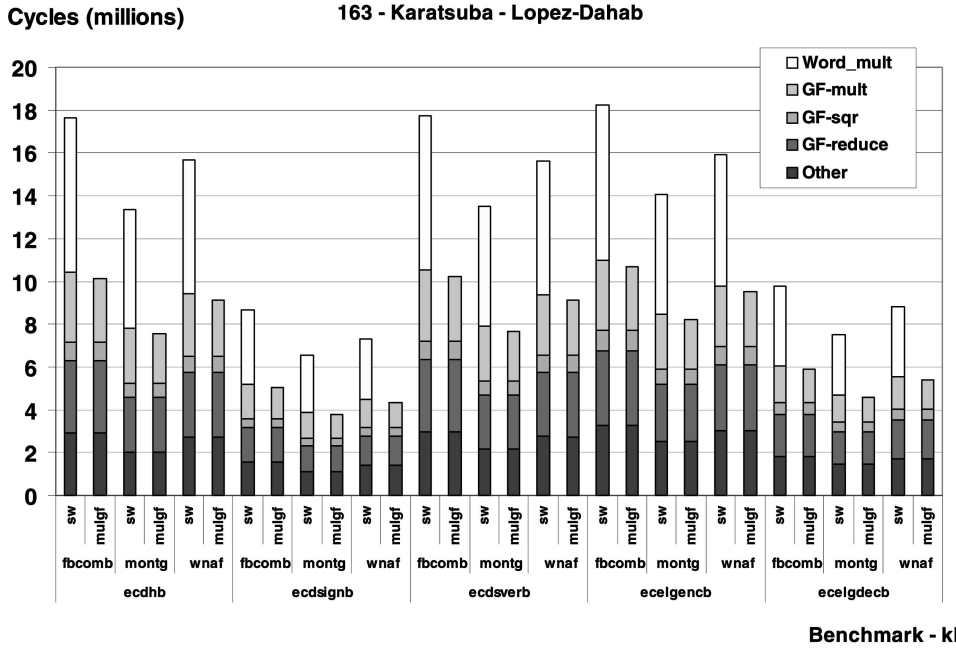


Fig. 7. The total execution time of all benchmarks for the 163-bit key length for the sw and mulgf-supported versions of the benchmarks. The Karatsuba finite field multiplication and Lopez-Dahab coordinates are employed.

From the comparison between the *sw* and *mulgf* results, Fig. 6 highlights that the relative performance of the [k]P methods are also kept when using MULGF: Montgomery-[k]P is the fastest choice, followed by wNAF (4-bit window), and fbComb (4-bit window). Affine coordinates, when not using the Montgomery-[k]P method, deliver worse performance than projective ones in the software implementation and, in addition, take less advantage of MULGF due to the lower number of multiplications performed. This way, the availability of MULGF increases the benefits of employing projective coordinates. The Montgomery [k]P method for the Lopez-Dahab coordinates is still the fastest mix, reducing the execution time by 44 percent, on average, due to the MULGF instruction.

5.3 Benchmark Results with MULGF

In this section, we analyze the overall speedup provided by the MULGF instruction on the considered benchmarks. These results represent the perceived benefits at the application level when the corresponding security protocols are employed. Fig. 7 shows the execution time (millions of cycles) of the benchmarks for the three different [k]P methods using a 163-bit key size and the Lopez-Dahab coordinates for the baseline sw and the processor augmented with mulgf. Note that the time spent in the MULGF instruction execution (even bars) is included in the GF-mult portion. The results in Fig. 7 highlight that, for all benchmarks and for all [k]P algorithms, MULGF allows us to save the share originally devoted to word-level multiplication, resulting in a significant execution time reduction (41 percent on average).

When using the MULGF instruction, the time for the GF-mult portion is slightly reduced (about 8 percent less). This is due to three combined effects. First, the time spent for MULGF is included in GF-mult. Second, MULGF execution substitutes a function call/return, along with the preparation of its parameters/results onto the stack frame. Third, by substituting this function, MULGF increases locality in the instruction cache [21] as the multiplication code (Karatsuba here) is accessed seamlessly.

Fig. 8 (left) confirms the results for various key sizes: an average of 49 percent reduction in the execution time for 233 bits and an average of 48 percent reduction for 283 bits. By increasing the key size, the higher relative weight of word-level multiplication in the total execution time (see Fig. 8 (right)) allows for higher savings than for 163 bits. The slight differences between 233 and 283 bits are mainly due to the different numbers of words representing finite field elements. Fig. 8 (left) shows that the MULGF instruction leaves a cycle breakdown dominated by finite-field multiplication algorithm: from 30 percent up to 40 percent for the considered key sizes.

5.4 Sensitivity Analysis against MULGF Implementation Parameters

Here, we analyze the performance sensitivity against various architectural choices for supporting the MULGF instruction and we explore the processor features that allow for better exploitation of such architectural support. In particular, we explore the effects of adopting one or four units for polynomial multiplication (num_polyMul), two or four ALUs (num_ALU), widths of one, two, or four for the superscalar pipeline (pipe_width), small (32) or large (64) RFs, and in-order/out-of-order execution of instructions [31].

The processor model simulated so far has one poly_mul unit, two ALUs, a single pipeline, in-order issue, and a small RF. The resulting IPCs are 0.68. Fig. 9 shows that no benefit can be obtained from multiple poly_mul units when the pipeline is kept one instruction wide and the execution is in order. This indicates that the instruction flow does not have many MULGF instructions (near each other) able to take advantage of the available units. In addition, the low

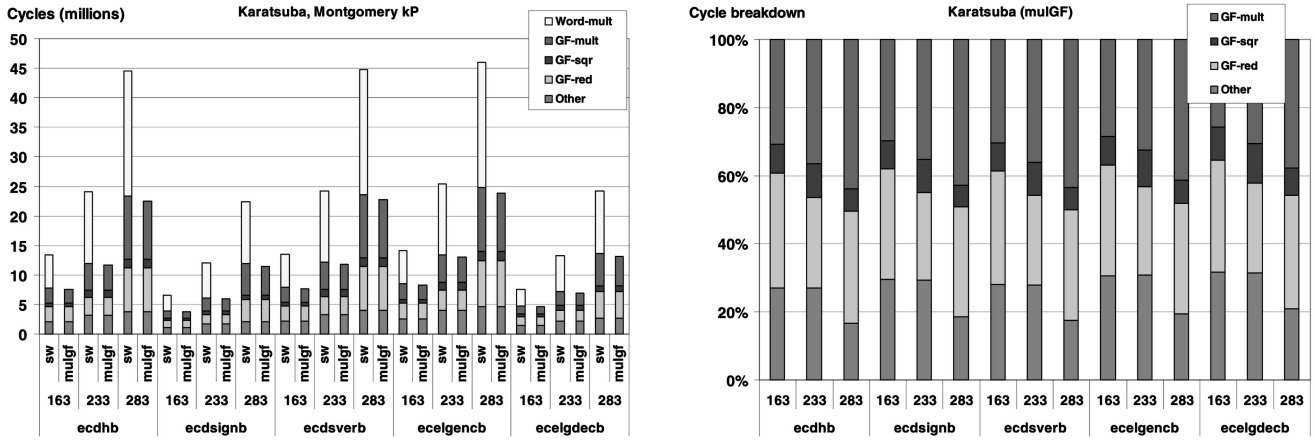


Fig. 8. Left: Overall cycle breakdown of the considered benchmarks, sw, and MULGF-supported implementations in terms of word-level multiplications (Word-mult), finite field multiplication management (GF-mult), finite field squaring (GF-sqr), reduction (GF-red), and other operations (Other). Right: Normalized breakdown of the time spent in the different activities in case of the MULGF-supported implementation.

latency of the poly_mul unit (for example, we assume three cycles as for the integer multiplier) reduces the probability of issuing an MULGF when another is still in execution. Therefore, in a relatively simple embedded processor, deploying more than one poly_mul unit is useless. When the pipeline does not expose parallelism (pipe_width 1), all other parameters have limited effects on performance.

Conversely, a two-way pipeline delivers a minimum of 1.02 IPC for the simplest processor configuration and scales up to 1.29 in case of out-of-order execution and 1.31 when a large RF is employed. These numbers indicate that the single pipeline is the main bottleneck of the processor. For a two instructions wide pipeline, the figure also shows an absolute performance insensitivity to the number of poly_mul units or ALUs: Performance is not limited by such resources. With a four instructions wide pipeline, the simplest configuration delivers 1.11 IPC and scales up to 1.68 for the out-of-order execution for both one and four poly_mul units and two ALUs. Only this high degree of internal parallelism can take advantage of the four-ALU configuration delivering 1.94 IPC for out-of-order and the small RF and 2.21 IPC for out-of-order and the large RF (that is, $+225$ percent over the baseline). In practice, only a very complex architecture employing out-of-order execution would be able to extract a significant number of independent instructions during EC operations to exploit a good portion of the theoretical bandwidth (IPC = 4) of the processor. However, a two instructions wide superscalar processor (especially out-of-order) can achieve good speed-ups: $+50$ percent and $+90$ percent, respectively. With respect to complex processor organizations, we investigated the usage of internal processor resources and verified that none of them was an execution bottleneck. In addition, the external memory does not limit performance: In fact, the miss rates of both instruction and data caches are negligible (one to three misses per 1,000 accesses). These results suggest that the bottleneck resides in the data dependencies among the benchmark instructions.

Moreover, in Fig. 10, we analyze the sensitivity of the overall execution time over a range of poly_mul latencies (one to six cycles) and degrees of pipelining in the unit. In the figure, L_P means that the poly_mul will take L cycles to complete, but, after P cycles, it is possible to issue another

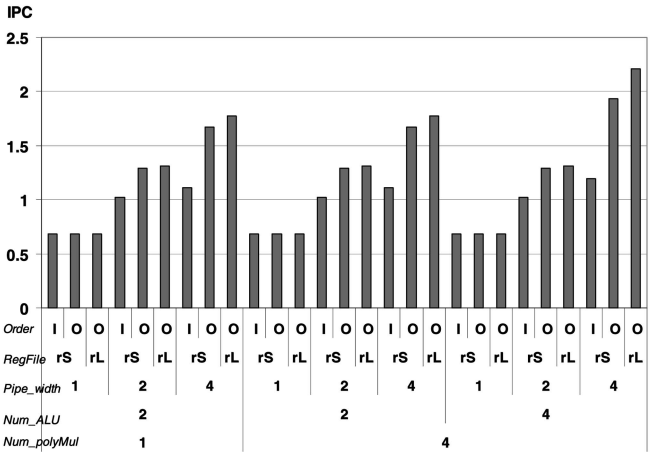


Fig. 9. IPC of the [k]P benchmark for various experimental architectural configurations: the number of poly_muls (num_polyMul), number of ALU (num_ALU), degree of parallelism (pipe_width), large (rL) or small (rS) RF, and issue and execution policy (I = order, O = out of order).

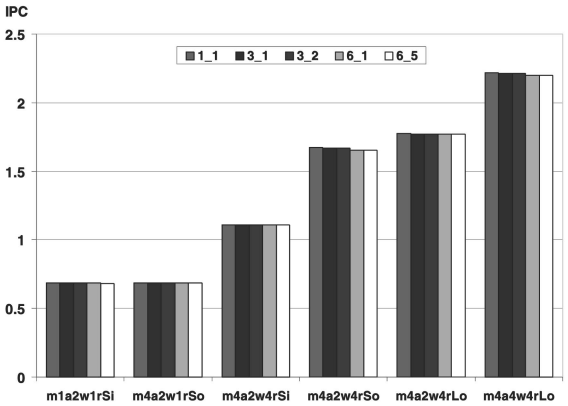


Fig. 10. IPC of the [k]P benchmark for various processor configurations and for different latencies of the poly_mul unit: L_P means that poly_mul takes L cycles to complete and, after P cycles, it can accept a new instruction. P = 1 indicates a fully pipelined unit.

instruction to the unit. For instance, 6_1 is a fully pipelined configuration that can receive one instruction per cycle but takes six cycles to complete each instruction. We assume that a possible real implementation of poly_mul in our platform can be at least as fast as the integer multiplier (three cycles fully pipelined).

We analyze different pipelining abilities for the unit: For one cycle per instruction (L_1) and latencies of three and six cycles, we also analyze two cycles per instruction and five cycles per instruction. The figure highlights a very small IPC sensitivity to the latency of the poly_mul unit (less than 1 percent) for all of the considered processor configurations. In particular, there is almost no sensitivity for the simplest ones. Processor configurations are represented by mXaYwZrK(I/O), which means X poly_mul units (m), Y ALUs (a), Z pipeline width (w), small (S) or large (L) RF (r), and in-order (I) or out-of-order (O) issue and execution. These results highlight that the fast implementation of the poly_mul unit is not as critical as the other architectural choices for the processor.

6 Related Work

An extensive study of the efficient methods for EC arithmetic on binary fields can be found in [15] for NIST-recommended curves and the Intel Pentium II platform. This article served as a reference for our ECC implementation. The work of Weimerskirch et al. [18] presents the implementation of ECC as part of OpenSSL. The analyzed algorithms are specially suited for high-performance devices like large-scale server computers. Field arithmetic implementation does not rely on a specific field size or field polynomial. The authors also present timings of finite field operations on Pentium and SPARC processors. Articles on instruction-set extensions for ECC have recently appeared [19], [20], [21], [22], [24]. All of these propose adopting instruction sets equipped with instructions for word-level polynomial multiplication and, sometimes, for multiply and accumulate. In [20], Großschädl and Kamendje propose and analyze algorithms specially suited for ECC based on the availability of an instruction for polynomial multiplication. They also evaluate the results through a functional model of a 16-bit RISC-like processor with a simple single-issue pipeline. The work of Eberle et al. [22] shows that simple extensions of the data path suffice to efficiently support ECC over $GF(2^m)$ and to outperform ECC over GF(p) on an 8-bit processor. These extensions include an extended integer multiplier that also generates multiplication results for $GF(2^m)$ and a multiply-accumulate instruction for efficient multiprecision multiplications. The implementations described in this paper were evaluated on an Atmel ATmega 128 8-bit microprocessor from Atmel Inc., employing a modern RISC architecture. The authors did not use any optimization of the multiplication method and confirm that there is no need to optimize the squaring instruction since a polynomial multiplier unit can be used for this purpose too. In [19], Fiskiran and Lee evaluate polynomial multiplication and multiply-accumulate instructions for processors with wider word size and multiple-issue execution. The effects of varying the number of functional units and load/store pipes are also considered. The evaluation is performed for the PAX architecture, which supports a minimalist ISA for cryptographic processing. Compared to an optimized software implementation, multiple-issue execution provides a speedup of 3.6 times. The inclusion of a dedicated binary-field multiplier provides about a speedup of 6.5 times and the combined speedup from the new ISA (multiplication only) and superscalar execution reaches about 10 times. Furthermore, by including an instruction for field squaring, this speedup can be increased to about 22 times. Eberle et al. [24] propose a specific extension to a high-performance general-purpose processor intended to be the core of a flexible security server, supporting multiple cryptosystems: RSA, DSA, DH, arbitrary curves, and finite fields (GF(p) and $GF(2^m)$). They propose a 64-bit dual-field multiplier unit and a specific 64-bit data path to support its operands.

To a smaller or greater extent, all of the papers described propose the architecture of specific security processors and thus tend to tune the design mainly to security applications. This choice achieves very good performance on cryptographic operations. Our proposal (see [21]) is orthogonal to these as we aim at maximizing the performance gain through the insertion of minimal modifications to the existing data path of a widespread embedded processor (ARM-based XScale). In particular, we adopt an evaluation methodology that models all of the details of the processor internals so that cycle-accurate performance measurements can be done both on the unmodified and on the extended processors. In addition, the proposed approach exposes the utilization of the internal resources of the processor in order to provide design and configuration indications.

7 Conclusions

In this paper, we have used $GF(2^m)$ ECC applications to investigate the performance of an ARM-based XScale processor and a variant that integrates 32-bit polynomial multiplier units into its data path. Several algorithms have been analyzed for finite field multiplication and EC scalar multiplication ([k]P), as well as for different coordinate systems. For all benchmarks and the library, we performed the cycle breakdown of execution time for different code sections (activities). This way, we highlighted the most important activities that make up the overall execution time of application-level benchmarks, finding that word-level multiplication accounts for the biggest fraction of the execution cycles (40 percent to 48 percent).

We then investigated the performance effects of integrating a word-level polynomial multiplication inside the existing XScale data path and verified the benefits to the benchmarks: an average of 41 percent reduction in execution cycles. We showed that the most time-consuming remaining activities are the management of multiprecision multiplication (~ 35 percent) and modular reduction (~ 25 percent). We explored the effects on performance caused by different design alternatives of the processor data path and by the availability of multiple polynomial multiplication units, finding that the sole adoption of a two instructions wide superscalar data path can be enough to improve IPC by 50 percent and by 90 percent when using out-of-order execution. This could be an interesting trade-off for this kind of applications. We also showed that out-of-order execution is favorable only in the presence of superscalar pipelines (for example, two or four instructions

wide). In addition, adopting more than one polynomial multiplication unit is useless for almost any considered processor organization. In contrast, increasing the number of ALUs is particularly beneficial (+230 percent IPC) only with a very complex processor: a four-way superscalar out-of-order data path with four ALUs and a large register file. Finally, we showed that the polynomial multiplication latency and its pipelining ability do not practically affect the benchmark performance (< 1 percent IPC), leaving freedom to choose such parameters according to other constraints.

ACKNOWLEDGMENTS

The authors wish to acknowledge the anonymous reviewers for their useful comments and suggestions that helped improve this paper. They are particularly grateful to Professor Sally McKee for the precious discussions on this paper. This work was supported by the European Commission in the context of the SARC Integrated Project 27648 (FP6), by the HiPEAC Network of Excellence (FP6), by Contract IST-004408, and by the PAR-2005 Research Program of the University of Siena.

REFERENCES

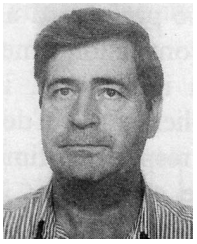
[1] T. Austin, E. Larson, and D. Ernst, "SimpleScalar: An Infrastructure for Computer System Modeling," *Computer,* vol. 35, no. 2, pp. 56-59, Feb. 2002.
[2] Intel® XScale® Core Developer's Manual, http://developer.intel.com, 2007.
[3] SimpleScalar Architectural Simulator, http://www.simplescalar.com, 2007.
[4] MIRACL Big Integer Library, http://indigo.ie/~mscott, 2007.
[5] Nat'l Inst. Standards and Technology (NIST), Digital Signature Standard (DSS), Fed. Information Processing Standards (FIPS) Publication 186-2, Jan. 2000.
[6] W. Diffie and M.E. Hellman, "New Directions in Cryptography," *IEEE Trans. Information Theory,* vol. 22, pp. 644-654, Nov. 1976.
[7] T. ElGamal, "A Public-Key Cryptosystem and Signature Scheme Based on Discrete Logarithms," *IEEE Trans. Information Theory,* vol. 31, no. 4, pp. 469-472, July 1985.
[8] A.J. Menezes, *Elliptic Curve Public Key Cryptosystems.* Kluwer Academic, 1995.
[9] R. Schroeppel, H. Orman, S. O'Malley, and O. Spatscheck, "Fast Key Exchange with Elliptic Curve Cryptosystems," *Proc. Advances in Cryptology,* pp. 43-56, 1995.
[10] A. Karatsuba and Y. Ofman, "Multiplication of Multidigit Numbers on Automata," *Soviet Physics-Doklady,* vol. 7, pp. 595-596, 1963.
[11] S.S. Erdem and Ç.K. Koç, "A Less Recursive Variant of Karatsuba-Ofman Algorithm for Multiplying Operands of Size a Power of Two," *Proc. 16th IEEE Int'l Symp. Computer Arithmetic,* pp. 28-35, June 2003.
[12] P.L. Montgomery, "Modular Multiplication without Trial Division," *Math. of Computation,* vol. 44, pp. 519-521, 1985.
[13] Ç.K. Koç and T. Acar, "Montgomery Multiplication in $GF(2^k)$," *Design, Codes and Cryptography,* vol. 14, no. 1, pp. 59-67, Jan. 1998.
[14] D.A. Knuth, *The Art of Computer Programming 2, Seminumerical Algorithms,* second ed. Addison-Wesley, 1981.
[15] D.R. Hankerson, J.C. López Hernandes, and A.J. Menezes, "Software Implementations of Elliptic Curve Cryptography over Binary Fields," *Proc. Second Int'l Workshop Cryptographic Hardware and Embedded Systems,* pp. 1-24, 2000.
[16] J. López and R. Dahab, "Fast Multiplication on Elliptic Curves over $GF(2^m)$ without Precomputation," *Lecture Notes in Computer Science,* vol. 1717, pp. 316-327, Springer-Verlag, 1999.
[17] I.F. Blake, G. Seroussi, and N.P. Smart, *Elliptic Curves in Cryptography.* Cambridge Univ. Press, 1999.
[18] A. Weimerskirch, D. Stebila, and S. Chang Shantz, "Generic $GF(2^m)$ Implementation in Software and Its Application in ECC," *Proc. Eighth Australasian Conf. Information Security and Privacy,* 2003.
[19] A.M. Fiskiran and R.B. Lee, "Evaluating Instruction Set Extensions for Fast Arithmetic on Binary Finite Fields," *Proc. 15th IEEE Int'l Conf. Application-Specific Systems, Architectures, and Processors,* pp. 125-136, Sept. 2004.
[20] J. Großschädl and G. Kamendje, "Instruction Set Extension for Fast Elliptic Curve Cryptography over Binary Finite Fields $GF(2^m)$," *Proc. 14th IEEE Int'l Conf. Application-Specific Systems, Architectures and Processors,* pp. 455-468, June 2003.
[21] S. Bartolini, I. Branovic, R. Giorgi, and E. Martinelli, "A Performance Evaluation of ARM ISA Extensions for Elliptic Curve Cryptography over Binary Finite Fields," *Proc. 16th IEEE Symp. Computer Architecture and High Performance Computing,* pp. 238-245, Oct. 2004.
[22] H. Eberle, A. Wander, N. Gura, and S. Chang Shantz, "Architectural Extensions for Elliptic Curve Cryptography over $GF(2^m)$ on 8-bit Microprocessors," *Proc. 16th IEEE Int'l Conf. Application-Specific Systems, Architecture Processors,* pp. 343-349, July 2005.
[23] V. Gupta, M. Wurm, Y. Zhu, M. Millard, S. Fung, N. Gura, H. Eberle, and S. Chang Shantz, "Sizzle: A Standards-Based End-to-End Security Architecture for the Embedded Internet," *Pervasive and Mobile Computing J.,* vol. 1, no. 4, pp. 425-445, Dec. 2005.
[24] H. Eberle, N. Gura, S. Chang Shantz, V. Gupta, and L. Rarick, "A Public-Key Cryptographic Processor for RSA and ECC," *Proc. 15th IEEE Conf. Application-Specific Systems, Architectures and Processors,* pp. 98-110, Sept. 2004.
[25] T.H. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein, *Introduction to Algorithms,* second ed. MIT Press and McGraw-Hill, 2001.
[26] P. Montgomery, "Speeding the Pollard and Elliptic Curve Methods of Factorization," *Math. of Computation,* vol. 48, pp. 243-264, 1987.
[27] G.B. Agnew, R.C. Mullin, and S.A. Vanstone, "An Implementation of Elliptic Curve Cryptosystems over $F2^{155}$," *IEEE J. Selected Areas in Comm.,* vol. 11, no. 5, June 1993.
[28] E. Savaş, A.F. Tenca, and Ç.K. Koç, "Dual-Field Multiplier Architecture for Cryptographic Applications," *Conf. Record 37th Asilomar Conf. Signals, Systems, and Computers,* pp. 374-378, Nov. 2003.
[29] J. Großschädl, *A Bit-Serial Unified Multiplier Architecture for Finite Fields GF(p) and* $GF(2^m)$, *Lecture Notes in Computer Science,* vol. 2162, pp. 202-219, Springer-Verlag, 2001.
[30] J. López and R. Dahab, "Improved Algorithms for Elliptic Curve Arithmetic in $GF(2^n)$," Technical Report IC-98-39, Relatório Técnico, Oct. 1998.
[31] J.L. Hennessy and D.A. Patterson, *Computer Architecture: A Quantitative Approach,* third ed. Morgan-Kaufmann, 2003.
[32] N. Koblitz, "Elliptic Curve Cryptosystems," *Math. of Computation,* vol. 48, pp. 203-209, 1987.
[33] V. Miller, "Use of Elliptic Curves in Cryptography," *Proc. Advances in Cryptology '85,* 1985.
[34] M. Brown, D. Hankerson, J. Lopez, and A. Menezes, "Software Implementation of the NIST Elliptic Curves over Prime Fields," *Proc. Cryptology Track of the RSA Conf.,* D. Naccache, ed., pp. 250-265, 2001.
[35] NIOS-II Processor Web site, http://www.altera.com/products/ip/processors/nios2/, 2007.
[36] A. Reyhani-Masoleh and M.A. Hasan, "Low Complexity Bit Parallel Architectures for Polynomial Basis Multiplication over $GF(2^m)$," *IEEE Trans. Computers,* vol. 53, no. 8, pp. 945-959, Aug. 2004.
[37] G.B. Agnew, T. Beth, R.C. Mullin, and S.A. Vanstone, "Arithmetic Operations in $GF(2^m)$," *J. Cryptology,* vol. 6, pp. 3-13, 1993.
[38] A.J. Menezes, I.F. Blake, X. Gao, R.C. Mullin, S.A. Vanstone, and T. Yaghoobian, *Applications of Finite Fields.* Kluwer Academic, 1993.
[39] Ç.K. Koç and B. Sunar, "Low-Complexity Bit-Parallel Canonical and Normal Basis Multipliers for a Class of Finite Fields," *IEEE Trans. Computers,* vol. 47, no. 3, pp. 353-356, Mar. 1998.
[40] J.A. Solinas, "Efficient Arithmetic on Koblitz Curves," *Designs, Codes and Cryptography,* vol. 19, pp. 195-249, 2000.
[41] C.H. Lim and P.J. Lee, "More Flexible Exponentiation with Precomputation," *Lecture Notes in Ccomputer Science,* vol. 839, pp. 95-107, Springer-Verlag, 1994.
[42] D.V. Chudnovsky and G.V. Chudnovsky, "Sequences of Numbers Generated by Addition in Formal Groups and New Primality and Factorization Tests," *Advances in Applied Math.,* vol. 7, pp. 385-434, 1987.

[43] S. Okada, N. Torii, K. Itoh, and M. Takenaka, "Implementation of Elliptic Curve Cryptographic Coprocessor over $GF(2^m)$ on an FPGA," *Proc. Fourth Int'l Workshop Cryptographic Hardware and Embedded Systems,* pp. 25-40, Jan. 2002.
[44] M. Ernst, M. Jung, F. Madlener, S. Huss, and R. Blümel, "A Reconfigurable System on Chip Implementation for Elliptic Curve Cryptography over GF(2n)," *Proc. Fourth Int'l Workshop Cryptographic Hardware and Embedded Systems,* pp. 381-399, Jan. 2003.
[45] V. Miller, "Use of Elliptic Curves in Cryptography," *Proc. Advances in Cryptology '85,* pp. 417-426, 1986.
[46] Pentium-4 IA-32 Intel Architecture Optimization Reference Manual, www.intel.com, 2007.

Sandro Bartolini received the PhD degree in computer engineering from the University of Pisa. He is currently an assistant professor in the Department of Information Engineering at the University of Siena. He took part in various research and industrial projects on advanced computer architecture, embedded systems, cache and memory subsystems, compiler optimization, low-power special hardware for security, and cryptography, which all constitute his research interests. He is an associate editor for the *EURASIP Journal on Embedded Systems* and has been a guest editor of some issues of the *ACM Computer Architecture News* and the *Journal of Embedded Computing*. Since 2002, he has been a co-organizer of the Memory Performance: Dealing with Applications, Systems and Architecture (MEDEA) International Workshop. He is a member of the HiPEAC Network of Excellence, the IEEE, the IEEE Computer Society, and the ACM.

Irina Branovic received the MSc degree in electrical and electronic engineering from the University of Belgrade, Serbia, in 2001 and the PhD degree in information engineering from the University of Siena in 2005. Her current research interests include cryptography and information security, algorithms and architectures for computations in finite fields, and fast implementation of public key cryptographic systems.

Roberto Giorgi received the MS degree in electronics engineering (magna cum laude) and the PhD degree in computer engineering from the University of Pisa. He is an associate professor in the Department of Information Engineering at the University of Siena. He was a research associate at the University of Alabama, Huntsville. He is participating in the European projects High-Performance Embedded-System Architecture and Compiler (HiPEAC) and Scalable Architectures (SARC) in the area of future and emerging technologies. He took part in the ChARM Project, developing part of a software used for the performance evaluation of ARM processor-based embedded systems with cache memory, in cooperation with VLSI Technologies Inc., San Jose (which later became Philips Semiconductor and now NXP). His current research interests include computer architecture themes such as embedded systems, multiprocessors, memory system performance, workload characterization, and high-performance systems for Web and database applications. He is a member of the HiPEAC Network of Excellence and the ACM and a senior member of the IEEE and the IEEE Computer Society.



Enrico Martinelli received the bachelor's degree (cum laude) in electronic engineering from the University of Pisa in 1970. From 1971 to 1994, he was a researcher with the Italian National Research Council (CNR). Since 1994, he has been a full professor in the Department of Information Engineering at the University of Siena and is currently the head of the department. His research interests include computer architectures, in particular the design of high-speed arithmetic devices for special applications, DSP, and cryptography.