A Performance Evaluation of ARM ISA Extension for Elliptic Curve Cryptography over Binary Finite Fields

Sandro Bartolini, Irina Branovic, Roberto Giorgi, Enrico Martinelli
Department of Information Engineering, University of Siena, Italy
{bartolini,branovic,giorgi,enrico}@dii.unisi.it

Abstract

In this paper, we present an evaluation of possible ARM instruction set extension for Elliptic Curve Cryptography (ECC) over binary finite fields GF(2^m). The use of elliptic curve cryptography is becoming common in embedded domain, where its reduced key size at a security level equivalent to standard public-key methods (such as RSA) allows for power consumption savings and more efficient operation. ARM processor was selected because it is widely used for embedded system applications. We developed an ECC benchmark set with three widely used public-key algorithms: Diffie-Hellman for key exchange, digital signature algorithm, as well as El-Gamal method for encryption/decryption. We analyzed the major bottlenecks at function level and evaluated the performance improvement, when we introduce some simple architectural support in the ARM ISA. Results of our experiments show that the use of a word-level multiplication instruction over binary field allows for an average 33% reduction of the total number of dynamically executed instructions, while execution time improves by the same amount when projective coordinates are used.

1. Introduction

Cryptographic processing is used in many areas, for instance electronic commerce, securing e-mail and wireless communications. Current uses of public-key cryptography include encryption schemes (like RSA or El-Gamal), digital signature schemes (like Digital Signature Algorithm, DSA), and key agreement methods (like Diffie-Hellman). A common feature of "traditional" public-key schemes is the need to operate on relatively long integer data (1024-4096 bits) to achieve enough security. While performing typical public-key operations such as modular multiplication or exponentiation on a 1024-bit data is not so critical on current desktop computers, the performance of public-key cryptography methods is still critical in embedded environments, especially for application in wireless, handheld internet devices and smart cards with small memory and strict CPU-latency constraints.

Usual solution for improving public-key cryptography performance is the use of coprocessors that accelerate long integer arithmetic operations. Our proposal deviates from expensive solutions, such as co-processors, in order to find a trade-off that could provide performance improvement at little additional cost, by observing what are the mostly important operations, in a RISC-like fashion.

Elliptic Curve Cryptography (ECC) is becoming more and more used as an alternative to "standard" public-key methods. Its major advantage is the fact that it uses shorter keys at security level equivalent to "standard" public-key algorithms, which translates into faster implementations, reduced energy and bandwidth consumption. These characteristics make ECC especially well suited for implementation in embedded systems. For example, security of RSA encryption with 1024-bit key is approximately at the same level as the use of 163-bit key in the case of ECC over $GF(2^m)$ [13]. ECC is already incorporated into two important public-key cryptography standards, FIPS 186-2 [13] and IEEE-P1363 [7].

Elliptic Curves (ECs) can be defined over any field, but for cryptographic purposes, elliptic curves over finite fields are most often used. The performance of ECC is determined by the efficiency of the arithmetic in the underlying finite field (Galois Field over prime field $GF(p)$ or over binary field $GF(2^m)$).

Since it appears that arithmetic in $GF(2^m)$ can be implemented more efficiently in hardware and software than arithmetic in $GF(p)$, elliptic curves over binary finite fields have seen wider use in commercial applications. However, embedded processors do not implement instructions for efficient arithmetic in binary finite fields $GF(2^m)$. For this reason, we studied possible instruction set extensions for improving ECC performance over $GF(2^m)$.

We developed a benchmark set which implements elliptic curve versions of Diffie-Hellman key exchange, DSA, and El-Gamal encryption/decryption.

We chose ARM processor for our case study, since it is widely used for embedded applications. For example, ARM processor core is being developed into 90% of mobile phones and PDAs worldwide [2]. We examined possible instruction set extensions of ARM instruction set useful for ECC over binary finite fields based on the analysis of such typical ECC algorithms.

The performance improvement was verified by using the SimpleScalar architectural simulator for ARM target. Specifically, our simulation with an added instruction for multiplication of two 32-bit polynomials reduces the total number of dynamically executed instructions in average by 33%, and their execution time decreases by the same amount. A 32-bit word finite field polynomial multiplication can be implemented to have similar performance and complexity features as an integer multiplier [10]. Extending instruction set for this purpose therefore offers significant advantages with a limited increase of processor cost and die size.

This paper is organized as follows. In Section 2 we describe ECC algorithms in more detail. Section 3 is devoted to experimental setup and methodology, with results of experiments and discussion of proposed ISA extension for ECC over binary fields presented in Section 4. We considered related work in Section 5, and gave concluding remarks in Section 6.

2. Public-key ECC benchmarks

We considered the ECC version of three most widely used algorithms in public-key cryptography: Diffie-Hellman for key exchange, digital signature algorithm, and ElGamal for encryption/decryption.

Diffie-Hellman key exchange is used to establish a shared key between two parties over a public channel. Although it is the oldest proposal for eliminating the transfer of secret keys in cryptography, it is still generally considered to be one of the most secure and practical public-key schemes. The security of Diffie-Hellman method relays on the difficulty of calculating discrete logarithms (given an element α in finite field *GF*(*p*) and another element y in the same field, find an integer x such that $y = \alpha^x \mod p$).

Digital signature of a document is a cryptographic method for ensuring the identity of the sender and the authenticity of the sent data. Digital signature of a document is information based on both the document and signer's private key. The National Institute of Standards and Technology (NIST) published the Digital Signature Algorithm (DSA) in the Digital Signature Standard [13]. This standard requires use of Secure Hash Algorithm (SHA), specified in the Secure Hash Standard [12]. The SHA algorithm takes a long message and produces its 160-bit digest; this method is known as hashing. The message digest is then digitally signed with the signer's private key; signature can be verified using the sender's public key.

ElGamal is a public-key encryption/decryption scheme based on discrete logarithm problem, e.g. finding modular inverses of exponentiations in finite fields, and as such can be efficiently implemented using elliptic curves. Although RSA cryptosystem [14] has practically become the standard for public-key encryption, it does not offer any particular advantage when used with elliptic curves [4].

For a better understanding of the basic operations involved, we briefly recall ECC information related to our benchmarks. Since ECC over binary fields yields more efficient implementations [15], we focused on such fields. An elliptic curve over binary finite field is defined as the set of points (x, y) that satisfy the Weierstrass equation (in affine coordinates):

$$y^2 + xy = x^3 + ax^2 + b, \qquad a, b, x, y \in \mathrm{GF}(2^m)$$

together with a special point called the "point at infinity". The security of ECC lies in the fact that given two points P and Q on the curve, it is hard to find k (usually a large integer), such that $Q = kP$. This problem, called the elliptic curve discrete logarithm problem, has similar difficulty as solving discrete logarithm in integer fields.

The operation kP is also known as *scalar point multiplication*. The ECC versions of the above algorithms essentially substitute exponentiation with ECC point multiplication. The latter operation is performed in terms of a number of basic EC operations that are *point addition* and *point doubling*. If $P = (x_1, y_1)$ and $Q = (x_2, y_2)$ are two distinct points on the elliptic curve, the sum $P + Q = (x_3, y_3)$, which is also a point on the same curve, is defined by following equations (in affine coordinates):

$$x_3 = \lambda^2 + \lambda + a + x_1 + x_2$$
$$y_3 = \lambda(x_1 + x_3) + x_3 + y_1$$
$$\lambda = \begin{cases} \dfrac{y_2 + y_1}{x_2 + x_1} & \text{if } P \neq Q \\ x_1 + \dfrac{y_1}{x_1} & \text{if } P = Q \end{cases}$$

Various efficient methods for calculating kP in software exist [6] or [4]. Point addition and doubling operations ultimately translate to a certain number of four basic finite field operations (additions,

multiplications, squarings, and inversions), which define the overall efficiency of elliptic curve calculations. Namely, point addition (doubling) require the following $GF(2^m)$ operations:

- two multiplications
- one squaring
- one inversion
- nine (eight for doubling) additions

Elliptic curve operations require use of multi-precision arithmetic over $GF(2^m)$, where m is much larger than word size of the processor (32 bits for ARM). The representation used for elements of the underlying field can have significant impact on the efficiency of elliptic curve cryptosystem. For $GF(2^m)$, *polynomial* and *normal basis* [4] representations are typically used. In this study, we used polynomial representations, since it is well known that it yields more efficient software implementation [6]. In polynomial representation, every element of the binary finite field can be expressed as a binary polynomial of maximum degree $m-1$. The polynomial can be seen as the vector of its coefficients, where each of the coefficients can be either 0 or 1. Such representation is very convenient for performing addition since it requires only simple bit-wise XOR.

We will continue with description of four basic finite field operations, in case of polynomial representation. The product of two field elements can be obtained by first multiplying the two elements as polynomials, which gives as the result a polynomial of the degree less than or equal to $2(m-1)$, and then calculating the rest of the division with the irreducible polynomial (reduction). A polynomial $f(x)$ is said to be irreducible if it cannot be factored in non-trivial polynomials over the same field.

Some DSP, such as Texas Instruments TMS320-C6400, implement the instruction for polynomial multiplication, followed by reduction with the irreducible polynomial. However, these instructions support maximum field size of $GF(2^8)$ as they are used in applications like Reed-Solomon coding.

Since there is no instruction for $GF(2^m)$ multiplication on current processors, this operation has to be done in software or by co-processors. As we are not interested in coprocessor implementations, we focused on software implementations analysis to find which operations could benefit from simple architectural support.

The basic method for multiplication is the "shift-and-add" algorithm. Some smarter approaches for finite filed multiplication are possible. We used the Karatsuba-Ofman algorithm [9], a recursive divide-and-conquer approach for multiplying two multi-precision operands. In practice, this procedure is used to recursively arrive to word-level (32-bit) polynomial multiplication, which is then performed by a variant of window-method [6] on 32-bit words.

By choosing the irreducible polynomial $f(x)$ as a low weight polynomial, i.e. the one with the least possible number of non-zero coefficients, reduction modulo $f(x)$ is a simple operation that is performed in linear time. For cases of practical interest, $f(x)$ is either trinomial or pentanomial. For the reduction of product modulo an irreducible polynomial we used the efficient procedure from [4].

Polynomial squaring is much faster than polynomial multiplication, since it can be obtained by inserting a zero bit between consecutive bits of the binary representation of field polynomial. The speed of squaring can be additionally improved if a pre-computed look-up table is used, however it was not implemented in our benchmark.

Inversion in finite field is the most time-consuming operation. The inversion operation can be avoided during finite field operations when *Jacobean projective* coordinates are used (although one inversion is necessary for point conversion), with the cost of additional multiplications. This approach was used in our benchmarks. To validate the effectiveness of projective coordinates, we compared the results against the case of affine coordinates.

3. Experimental methodology

The performance evaluation of our ECC benchmark set is done using a modified version of sim-profile and sim-outorder simulators of the SimpleScalar toolset [17] for the ARM target. The sim-outorder tool performs a detailed timing simulation of the modeled target. Simulation is execution driven, and accounts for speculative execution, branch prediction, cache misses, and other advanced features (see Table 1).

Table 1: Simulated baseline architecture, modeled after Intel XScale.

Fetch queue (instructions)	4
Branch prediction	8K bimodal, 2K 4-way BTB
Fetch & Decode width	1
Issue width	1 (in order)
ITLB	32 entry, fully associative
DTLB	32 entry, fully associative
Functional units	1 ALU, 1 int MUL/DIV
Instruction L1 cache	32 KB, 32-way
Data L1 cache	32 KB, 32-way
L1 cache hit latency (cycles)	1
L1 cache block size	32 bytes
L2 cache	none
Memory latency (cycles)	24
Memory bus width (bytes)	4

The ARM target of the SimpleScalar set supports the ARM7 integer instruction set, which has been validated against a Rebel NetWinder Developer workstation [3] by the developers of the simulator. The processor the we simulated processor has a configuration that is modeled after Intel XScale architecture [8], adopted by major PDA manufacturers like Toshiba, Fujitsu and HP. Details of the ARM processor configuration are in Table 1.

When simulating "ideal" memory, processor was configured to have non-existing cache and latency of main memory access of 1 cycle, which corresponds to cache hit latency.

The sim-outorder tool was modified to add cycle level function profiling, e.g. to produce exact number of execution cycles for every procedure in the code, based on the output of gcc-objdump. This tool was also modified to support the execution of new instructions. Some unimplemented system calls for ARM target are also added, although they were not critical for the execution of the benchmark.

We used `gcc` cross-compilers for ARM instruction set included with SimpleScalar package [17]. The `gcc` cross-assembler was also modified to recognize newly added instructions. All programs were compiled with -O2 level of optimization.

Since a common approach for implementing public-key cryptography is to use available open-source libraries that offer all basic cryptographic functions, we followed such approach, as a realistic one. In particular, we used MIRACL C library [11]. This library consists of over 100 routines that cover all aspects of multi-precision arithmetic and finite field operations. The library file containing elliptic curve primitives was modified at assembly level to include new instructions. The benchmark set that we implemented makes use of the MIRACLE C library, which does not provide the cryptographic algorithms. The description of our benchmark suite is given in Table 2.

The binary finite fields and elliptic curves used in tests were chosen according to NIST standard [13]. This standard recommends the use of certain curves with strong security properties to ease the interoperability between different implementations of security protocols. For binary polynomial fields, the curves were recommended for key sizes of 163, 233, 283, 409, and 571 bits. Apart from the field size, parameter files in use with our benchmarks specify parameters for initializing the curve, setting the base point on a curve, and setting the irreducible polynomial (trinomial or pentanomial) for reduction of polynomial product. We conducted experiments for the first three fields only, which cover the security requests of nowadays and next future applications.

The notation <*benchmark_acronym*>.*b*<*key_length*> was used for each benchmark, where *b* denotes the use of binary finite field, while *key_length* indicates the field size (e.g., *ecdsign.b233* means that the digital signature generation over $GF(2^{233})$ field is used).

Table 2: Description of our ECC benchmark set.

Benchmark acronym	Benchmark name	Description
ecdh	EC Diffie-Hellman key exchange	Generates a prime suitable for Diffie-Hellman algorithm and calculates a shared key.
ecdsign	EC digital signature generation	Calculates the message digest of a file using sha algorithm, signs the message using the private key and writes the signature into a file.
ecdsver	EC digital signature verification	Calculates the message digest of a file using sha algorithm, then verifies the signature using public-key.
ecelgenc	EC El-Gamal encryption	Encrypts a point on the curve using El-Gamal algorithm.
ecelgdec	EC El-Gamal decryption	Decrypts a point on a curve using El-Gamal algorithm.

4. Analysis of possible instruction set extensions for ECC

The scope of our study was to explore possible extensions of ARM instruction set that are useful for efficient elliptic curve cryptography operations. We firstly analyzed the instruction mix of elliptic curve benchmarks (Figure 1). The instruction mix shows a very large percentage (approximately 40%) of integer instructions, as well as load and store operations (approximately 50%). We expected a similar distribution since finite field operations translate into a large number of logical operations (XOR, shift etc.), and result in a large number of register-memory transfers to operate on *m*-bit data (e.g. *m* ranges from 163 to 283 and is much larger than 32 bit register width in ARM).

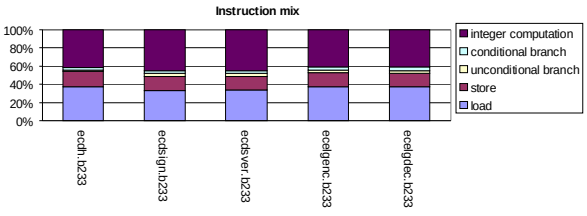

Figure 1: Instruction class distribution for elliptic curve cryptography benchmarks. The results for the key sizes of 163 and 283 bits are almost identical and thus are not reported.

On the simulated ARM architecture, we analyzed which functions are mostly affecting the execution time, through an accurate, cycle level profiling. This analysis was carried out for all ECC benchmark suite that we considered (Figure 2).

Mostly used procedures are:

- *karmul2* (recursive Karatsuba algorithm for $GF(2^m)$ polynomial multiplication),
- *mr_bottom4* (the base case in recursive calls of *karmul2* function),
- *mr_mul2* (word-level polynomial multiplication),
- *add2* ($GF(2^m)$ addition, e.g. XOR operation over *m*-bit polynomials),
- *square2* and *mr_sqr2* ($GF(2^m)$ and word-level squaring respectively),
- *reduce2* ($GF(2^m)$ reduction modulo irreducible polynomial),
- *copy* ($GF(2^m)$ polynomial assignment operation),
- *mr_lzero* (sets to zero a given $GF(2^m)$ element),
- *hashing*, *shs_transform* and *shs_process* (functions used for calculating message),
- *shiftbits*, and *numbits* have obvious meaning.

In particular, the *mr_mul2* procedure, which multiplies two 32-bit binary finite field polynomials and produces a 64-bit product consumes 34% of the total execution time in average for all benchmarks, and reaches 54% for Diffie-Hellman benchmark (*ecdh*). *mr_mul2* procedure is translated into about 400 dynamic instructions (roughly 500 cycles), which correspond to about 12 instructions per bit to perform 32-bit polynomial multiplication.

The difference in *mr_mul2* percentage between coding and decoding is due to the fact that encoding employs one scalar point multiplication more in respect to decoding. The latter shows higher influence by *mr_mul2* than other benchmarks because it is made up exclusively of operations on elliptic curves, without any other significant elaboration: in particular, *ecdh* benchmark uses two scalar EC point multiplications to allow a communication party to form a shared key with the other party.

The operation of El-Gamal algorithm is very similar to Diffie-Hellman one, in fact it uses the party public key to encode the message and the private key to decode it.

The other benchmarks use ECC to perform digital signature operations (signature and verify), and thus comprise a non-negligible activity for non-EC operations. In digital signature algorithm, more than 60% of the total execution time is spent in:

a) 18% in reading message file (*file_reading*) and

b) calculating its hash (about 45%), i.e. 160-bit digest (functions *shs_transform*, *shs_process*, and *hashing* in Figure 2).

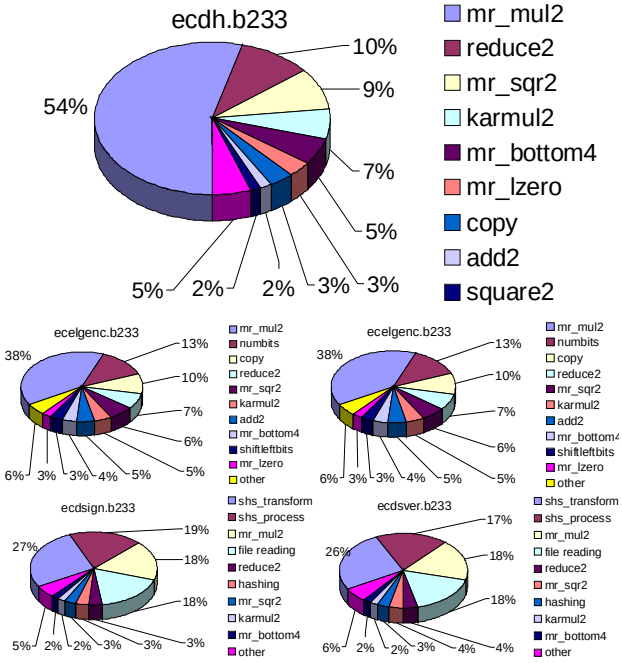

Figure 2: Breakdown of execution time in terms of program functions (cycle level profiling). Coordinates used were projective, the key length is 233 bits.

Figure 2 shows that digital signature and verification benchmarks have a very similar distribution of function usage. This is reasonable because verification procedure is made up of the signature calculation, followed by a comparison of calculated and received digital signatures. However, even in the case of digital signature, the most time-consuming finite field operation is word-level polynomial multiplication (18%, Figure 2). Overall, the impact of addition, squaring, reducing and inversion is at most 20% (*ecdh* benchmark), and is less important than finite field multiplication.

Based on the previous analysis, which showed that the word-level polynomial multiplication is the most time-critical operation, we decided to measure the impact of extending ARM instruction set with the instruction for polynomial word-level multiplication in binary finite fields. We called this instruction MULGF, similar as in [1]. The appropriate calls of C procedure for 32-bit polynomial multiplication in software were substituted with a single MULGF instruction. The MULGF instruction was modeled to have a delay of three cycles, as the integer multiplier unit of ARM

processor. Polynomial multiplication is essentially identical to integer multiplication, except that all carries are suppressed. Proposals for designing flexible multipliers already exist [16]. In our case, we only need the capability of doing bit addition both with and without carry. As in the standard full-adder circuit, the dual field adder has two XOR gates connected serially. Thus, its propagation time is not larger than that of full adder. Their areas differ slightly, but this does not cause a major change in the whole circuit.

The impact of adding the MULGF instruction for word-level polynomial multiplication in finite field is shown in Figures 3 and 4. Number of dynamically executed instructions, as well as the execution time, is lower by approximately one-third in average, with projective coordinates and when all key sizes are taken into account. This average result is in line with expectations, in fact, as the software implementation takes 500 cycles, the 3-cycle hardware implementation allows to reduce by a factor of 3/500 the time spent into word-level polynomial multiplication: from 34% on average (Figure 2) down to a negligible quota (i.e. less than 0.5%).

The improvement in execution time is more significant for Diffie-Hellman (54% in number of instructions and 55% for execution time in $GF(2^{233})$) and El-Gamal algorithms (48% for encryption and 37% for decryption in number of instructions, i.e. 39% and 35% in execution time in $GF(2^{233})$), where 32-bit polynomial multiplication is more used. The improvement for digital signature algorithm is more modest (19% in instruction number and 17% in execution time for the same key length), but still significant.

Given the similarity of Figures 3 and 4, it is clear that memory does not influence significantly ECC performance, when the cache size is 32KB+32KB, and that the working set of ECC algorithms is small.

In order to analyze the benchmark influence on the cache sub-system, we have explored various cache configurations with cache size ranging from a few hundred bytes to 64-KByte. Among them, we have selected here a couple of interesting cases. Firstly, 32-KByte high associativity (i.e. 32 way) I- and D-caches, which are representative of present Intel XScale processors based on ARM cores. Secondly, 1KByte+1KByte direct-mapped I- and D-caches, which match the average working-set size of the considered ECC benchmarks, and thus allow analyzing very precisely the effects on the memory hierarchy before and after the MULGF instruction is added. In particular, figure 5 shows the CPI before and after adding the MULGF instruction for 1KByte+1KByte I- and D-caches. The CPI is divided into two parts: the first one due to processor operation, and the second one is due to memory operations.

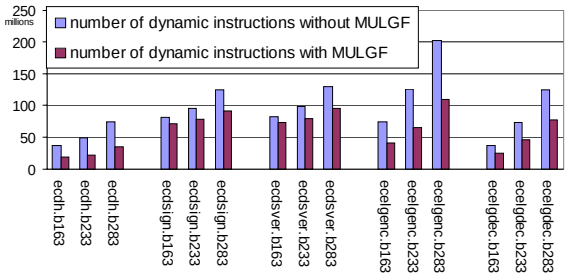

Figure 3: Number of dynamic instructions for projective coordinates before and after adding the MULGF instruction for word-level polynomial multiplication.

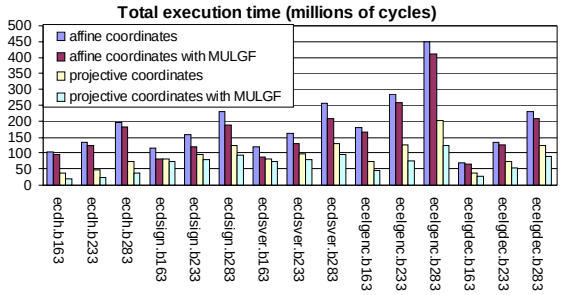

Figure 4: A comparison of the total execution time (millions of cycles) for projective and affine coordinates before and after adding the MULGF instruction.

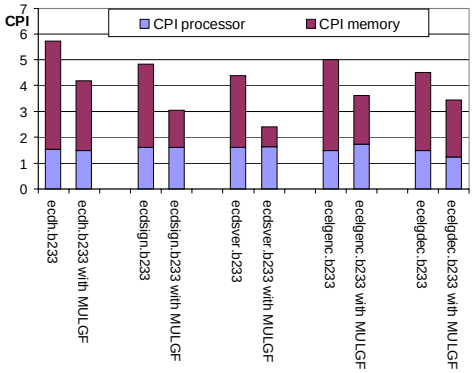

Figure 5: CPI (divided into processor and memory shares) before and after adding MULGF instruction, with 1KB + 1 KB direct-mapped instruction and data cache.

The total CPI is lower in average by 27% after adding MULGF instruction, with largest improvement by 37% and 44% for digital signing and verification, respectively. This highlights two points:

a) average CPI-processor of the ECC benchmarks is similar to the software implementation of MULGF (*mr_mul2*);

b) execution time reduction derives from the lower number of instructions executed (processor elaboration) and from improved "memory" CPI (lower number of accesses to I- and D-cache).

Figure 6 shows instruction and data cache miss rates with and without MULGF support. From Figure 6, we see that while instruction cache miss rates are lower, data cache miss rates grow after adding MULGF instruction. This happens because, after adding MULGF, the instructions of the software implementation of MULGF (*mr_mul2*, Figure 1) are not loaded anymore in the instruction cache, and the benchmark working set can fit better in it.

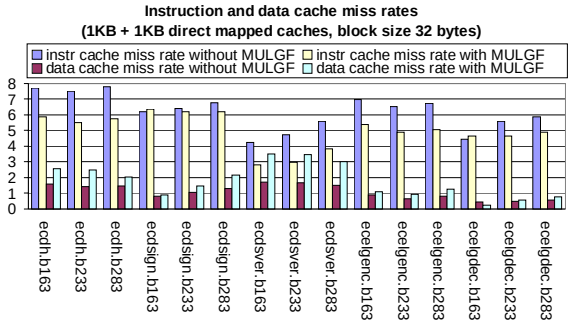

Figure 6: Instruction and data cache miss rates in percentages.

In the data cache, miss rate increases because the number of misses remains practically the same, but cache accesses decrease.

In fact, *mr_mul2* function performs many operations on a few local variables and thus its memory accesses are essentially hits.

Finally, in our analysis we considered also the use of affine coordinates in ECC and studied the performance of affine-ECC with MULGF instruction present (Figure 4). Our experiments show that adding MULGF has smaller impact when affine coordinates are used, because of the smaller number of finite field multiplications used (in affine coordinates, inversion in the finite field cannot be avoided, but some multiplications can be saved).

In case when affine coordinates are used, number of instructions and the execution time are lower, if MULGF is adopted, on average by 13% for all benchmarks. In addition, according to our experiments, projective coordinates are advantageous over affine ones both in a pure software implementation and when MULGF instruction is available.

The improvement obtained by projective-MULGF implementation over affine coordinates without MULGF is approximately fourfold in average for all benchmarks, in both instruction number and execution time (Figure 4).

5. Related work

Proposal of extending instruction set for public-key cryptography purposes can be found in [1]. In that work, some instruction set extensions for Intel and Sparc processors were proposed; among these, the polynomial multiplication in binary finite fields, called MULGF2. However, the effects of extensions were not evaluated. There is an intensive ongoing research in improving the efficiency of elliptic curve operations, as well as their performance analysis. An extensive study of efficient methods for elliptic curve arithmetic in binary finite fields for NIST-recommended curves can be found in [6]. A workload characterization of some public-key and private-key algorithms, including their elliptic curve equivalents for binary polynomial fields is found in [5]. They characterize operations in Diffie-Hellman, digital signature, and El-Gamal elliptic curve methods, and demonstrate that these algorithms can be implemented efficiently with a very simple processor.

Recently, few proposals of scalable dual-field cryptographic processors have appeared. Satoh and Takano in [15] propose ECC processor architecture that can support $GF(p)$ and $GF(2^m)$ fields for arbitrary prime numbers and irreducible polynomials by introducing a dual-field multiplier.

6. Conclusions

We presented an evaluation of instruction set extensions of a typical embedded processor for elliptic curve cryptography that can efficiently replace a coprocessor that is typically used for improving performance of ECC. Based on the analysis of typical elliptic curve cryptography benchmarks, we proposed to extend the ISA for word-level polynomial operations in binary finite fields. We considered the most important instruction that is used in such ISA, namely MULGF, and evaluated its impact on ECC performance for ARM processor. MULGF instruction for word-level polynomial multiplication can be added to existing ARM hardware by integrating a polynomial multiplier into existing datapath. Adding of this instruction is more justified when projective coordinates are used instead of affine, because when projective coordinates are used, finite field inversion is avoided at the price of higher number of finite field multiplications. In such case, the new instruction

improves the execution time on average by 33%, and decreases the number of dynamically executed instructions by 33%. Elliptic curve cryptography algorithms do not require large caches because of their reduced working set size. This information may be useful to design devices like smart-cards where the system runs the same program for its entire lifetime. Adding MULGF instruction lowers the instruction cache miss rate, while data cache miss rate increases because the data working set (and consequently the number of misses) remains the same, while number of accesses decreases. In the future, we intend to examine the effects of adding MULGF instruction when other multiplication algorithms are used, as well as to explore other possible instruction set extensions for elliptic curve cryptography.

Acknowledgments

This work is supported by Italian Ministry of Education, University and Research, under subcontracting of project FIRB "Reconfigurable platforms for wideband wireless communications", protocol RBNE018RFY and by the University of Siena through the project "Innovative processor architectures for embedded multimedia applications" PAR-2003.
We thank the anonymous reviewers for their useful comments.

References

[1] T. Acar, "High-speed algorithms and architectures for number-theoretic cryptosystems", PhD thesis, Oregon State University, 1998.
[2] ARM Web site, http://www.arm.com
[3] T. Austin, E. Larson, D. Ernst, "SimpleScalar: An Infrastructure for Computer System Modeling", *IEEE Computer*, Volume 35, Issue 2, pp. 56-59, 2002.
[4] I.F. Blake, G. Seroussi, N.P. Smart, *Elliptic Curves in Cryptography*, Cambridge University Press, 1999.
[5] A.M. Fiskiran and R.B. Lee, "Workload characterization of elliptic curve cryptography and other network security algorithms for constrained environments", *Proceedings of the 5th IEEE Annual Workshop on Workload Characterization*, pp. 127-137, 2002.
[6] D.R. Hankerson, J.C. Lopez Hernandes, and A.J. Menezes, "Software implementation of elliptic curve cryptography over binary fields", *Cryptographic Hardware and Embedded Systems - CHES 2000*, pp. 1-24, Springer Verlag, 2000.
[7] *IEEE P1363 Standard Specifications for Public-key Cryptography*, http://grouper.ieee.org/groups/1363/
[8] Intel Corporation, *The Intel XScale Microarchitecture Technical Summary*, ftp://download.intel.com/design/intelxscale/XscaleDatasheet4.pdf
[9] A. Karatsuba, Y. Ofman, "Multiplication of multidigit numbers on automata", *Soviet Physics - Koklady*, Vol. 7, pp. 595-596, 1963.
[10] H. Li, C.N. Zhang, "Efficient Cellular Automata Based Versatile Multiplier for $GF(2^m)$", *Journal of Information Science and Engineering 18*, pp. 479-488, 2002.
[11] *Miracl big integer library* Web site, http://indigo.ir/~mscott
[12] National Institute of Standards and Technology, *Secure Hash Standard*, FIPS Publication 180-1, 1995.
[13] National Institute of Standards and Technology, *Digital Signature Standard*, FIPS publication 186-2, 2000.
[14] *RSA Laboratories' FAQs about today's cryptography*, http://www.rsasecurity.com/rsalabs/faq
[15] A. Satoh, K. Takano, "A Scalable Dual-Field Elliptic Curve Cryptographic Processor", *IEEE Transactions on Computers*, Vol. 52, No. 4, pp. 449-460, April 2003.
[16] E. Savas, A.F. Tenca, C.K. Koc, "A scalable and unified multiplier architecture for finite fields $GF(p)$ and $GF(2^m)$", *Cryptographic Hardware and Embedded Systems - CHES 2000*, pp. 227-292, Springer-Verlag, 2000.
[17] *SimpleScalar LLC* Home Page, http://www.simplescalar.com