A Workload Characterization of Elliptic Curve Cryptography Methods in Embedded Environments

I. Branovic, R. Giorgi, E. Martinelli
University of Siena
Via Roma 56 - Siena, Italy
{branovic,giorgi,martinelli}@dii.unisi.it

Abstract

Elliptic Curve Cryptography (ECC) is emerging as an attractive public-key system for constrained environments, because of the small key sizes and computational efficiency, while preserving the same security level as the standard methods

We have developed a set of benchmarks to compare standard and corresponding elliptic curve public-key methods. An embedded device based on the Intel XScale architecture, which utilizes an ARM processor core was modeled and used for studying the benchmark performance. Different possible variations for the memory hierarchy of such basic architecture were considered. We compared our benchmarks with MiBench/Security, another widely accepted benchmark set, to provide a reference for our evaluation.

We studied operations and impact on memory of Diffie-Hellman key exchange, digital signature algorithm, ElGamal, and RSA public-key cryptosystems. Elliptic curve cryptosystems are more efficient in terms of execution time, but their impact on memory subsystem has to be taken into account when designing embedded devices in order to achieve better performance.

1. Introduction

Cryptography algorithms are split into two categories: private-key (symmetric) and public-key (asymmetric). Internet security protocols (e.g. SSL, IPSec) employ a public-key cryptosystem to exchange private keys and then use faster private-key algorithms to ensure confidentiality of streaming data. In private-key algorithms, communicating parties share a common private key, which is used to transform the original message into a ciphered message. The ciphered message is communicated to another side, and the original message is decrypted by using the same private key. Public-key systems, on the other side, do not require exchange of keys. The public key, known to all, can be used for encrypting messages. However, the resulting ciphertext can only be decrypted using the receiver's private key.

Elliptic Curve Cryptography has attracted attention due to the reduced key size at equivalent levels. Because of the favorable characteristics, it has been incorporated into two important public-key cryptography standards, FIPS 186-2 [*NIST00*] and IEEE-P1363 [*IEEE1363-00*]. These standards specify how to use elliptic curves over prime fields $GF(p)$ and binary fields $GF(2^m)$; recommended curves have well-studied properties that make them resistant to known attacks.

Due to expected advances in cryptanalysis and increases in available computing power, both private and public key sizes must grow over time to offer acceptable security. Table 1 [*NIST00*], [*Blake03*] shows expected key-size growth for various private and public-key cryptosystems:

Table 1. Equivalent key size for some cryptosystems.

Public key			private key
ECC key length		RSA key length for approximate equivalent security	length for approximate equivalent security
Prime field	Binary field		
192	163	1024	80
224	233	2048	112
256	283	3072	128
384	408	7680	192
521	571	15360	256

We study elliptic curve analogs of following algorithms for recommended curves in both types of fields:

- Diffie-Hellman key algorithm, used for secure exchange of private keys [*Diffie76*]
- Digital signature algorithm, used for ensuring authenticity of data [*NIST00*]
- ElGamal algorithm, used for encrypting data [*ElGamal85*].

We compared elliptic curve versions of public-key algorithms with corresponding standard versions. We also included RSA public-key algorithm [*RSA02*], since it is a de-facto standard in this area. To provide a refernce for our evaluation, we compared our benchmarks with MiBench/Security, another widely accepted benchmark set for embedded systems [*Guthaus01*].

The rest of the paper is organized as follows: in Section 2, we give information necessary for understanding public-key cryptography methods, as well as principles of ECC. Section 3 gives details on benchmarks used,

Section 4 outlines the methodology, while in Section 5 we present a workload characterization of public-key methods with special emphasis on memory performance. Section 6 presents related work in this area, and Section 7 concludes.

2. Public-Key Algorithms

In a public-key cryptosystem, the private key is always linked mathematically to the public key. Therefore, it is always possible to attack a public-key system by deriving the private key from the public key. The defense against this is to make the problem of deriving private key as difficult as possible.

2.1. Standard Public-Key Methods

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 relays on difficulty of calculating discrete logarithms (given an element α in a finite field F_p and another element y in the same field, find an integer x such that $y = \left|\alpha^x\right|_p$).

Digital signature of a document is a cryptographic means for ensuring the identity of the sender and the authenticity of 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 [*NIST00*]. This standard requires use of Secure Hash Algorithm (SHA), specified in the Secure Hash Standard [*NIST95*]. The SHA algorithm takes a long message and produces its 160-bit digest; this method is known as *hashing*. Hash function is hard to invert, which means that given a hash value, it is computationally extremely difficult to find the original message. The message digest is then digitally signed using the private key of the signer; signature can be verified using the sender's public key.

RSA cryptosystem [*RSA02*] is used in the most popular applications, such as SSL, IPSec, e-commerce systems, e-mail systems (PGP, S/MIME); it has practically become the standard for public-key encryption. RSA encryption is based on the fact that the computational difficulty of finding the private key is equivalent to factoring an integer, which is computationally impossible if it is long enough. RSA key sizes that today offer acceptable level of security are 1024 bits and longer. Public exponent in common use is $2^{16} + 1$ (65537), since it improves the efficiency of algorithm.

ElGamal is a public-key cryptosystem often used as an alternative to RSA. The encryption algorithm is based on discrete logarithm problem, e.g. finding modular inverses of exponentiations in finite field [*ElGamal85*].

2.2 Elliptic Curve Methods

The fundamental operation in RSA and Diffie-Hellman is modular integer multiplication. Unlike standard public-key methods that operate over integer fields, the elliptic curve cryptosystems operate over points on an elliptic curve. Cryptographic algorithms based on discrete logarithm problem can be efficiently implemented using elliptic curves. However, the core of elliptic curve arithmetic is an operation called *scalar point multiplication*, which computes $Q = kP$ (point P multiplied by an integer k gives a point Q on the same elliptic curve). The security of ECC lies in the fact that given P and $Q = kP$, it is hard to find k; this problem has similar difficulty as solving discrete logarithm in integer fields, although at the time being this operation seems harder in elliptic curve groups. Consequently, the same level of security is obtained with smaller key sizes compared with standard public-key methods (Table 1). While it is possible to carry out a brute force approach of computing all multiples of P to find Q, by choosing to operate over a large field, for instance binary field $GF(2^{163})$, k is so large that it becomes infeasible to determine k this way. The (large) random integer k is kept as the private key, while the result Q serves as the corresponding public key.

Elliptic curve can be defined over any field, but for cryptographic purposes, we are interested in elliptic curves over finite fields. Finite fields commonly in use in cryptography are prime and binary fields. In binary field, elements can be represented using polynomial or a normal basis. As it is well known that polynomial basis yields more efficient software implementations [*Hankerson00*], we used it in developing our benchmarks. Since not every elliptic curve offers strong security properties, standards organizations like NIST have published a set of recommended curves [*NIST00*]. The use of these curves is also recommended for easier interoperability between different implementations of a security protocol. For binary polynomial fields, two curves are recommended for key sizes of 163, 233, 283, 409, and 571 bits; for prime fields, for each key size (192, 224, 256, 384, and 521), one curve is recommended.

In the polynomial representation of binary field, each field element can be viewed as polynomial whose coefficients are either 0 or 1. Polynomial addition is defined as simple component-wise XOR of the two polynomials. Polynomial multiplication is also component-wise; the key difference is that multiplication may produce a product polynomial of degree that is greater or equal to the field size. In such case, the product needs to be reduced by the irreducible

polynomial (usually trinomial or pentanomial), defined in [*NIST00*]. Operations on elliptic curves in binary fields imply using finite field operations. For example, doubling of point in a binary field requires ten finite field operations: two multiplications, one squaring, six additions, and one field inversion [*Menezes01*].

3. Benchmark description

In typical portable wireless systems, cryptographic functions are performed in software due to the ease of implementation and flexibility that accompanies the use of software. The various public key cryptography schemes require the use of arithmetic algorithms that operate on operands that are much larger than the microprocessor's word size (e.g., 1024-bit operands on a 32-bit architecture). One common approach for implementing public-key cryptography is to use available open source libraries that offer all basic cryptographic functions. Our benchmarks were written by using *MIRACL* C library procedures for big integer arithmetic [*Miracl02*]. The *MIRACL* library consists of over 100 routines that cover all aspects of multi-precision arithmetic and offers procedures for finite-field elliptic curve operations. All routines have been optimized for speed and efficiency, while at the same time remaining standard, portable C. Algorithms used to implement arithmetic on the *big* data type are taken from [*Knuth81*].

The benchmarks we setup are listed in Table 4. The use of elliptic curve cryptography also involves choosing a finite field, a specific curve on it, and a base point on the chosen curve. The finite fields and the elliptic curves used in our benchmarks are chosen according to NIST standard [*NIST00*]. Elliptic-curve benchmarks in graphs and tables have the following notation:

<benchmark>.<b | p><three digit number>

The abbreviation *b* denotes an elliptic curve over binary field, while *p* denotes use of a prime field. Three-digit number indicates the size of the field; for example, *b163* denotes the use of $GF(2^{163})$ binary field, while *p384* denotes prime field with binary length of prime *p* equal to 384 (also indicated as ||p||). Therefore, the ec-dh.p192, ec-dh.p224, ec-dh.p256, ec-dh.p384, and ec-dh.p521 represent results of simulating elliptic curve Diffie-Hellman key exchange.

NIST curves over a prime field *GF(p)* are of form:

$$y^2 = x^3 - 3x + b \quad \text{where } b \text{ random}^1$$

while the curves over $GF(2^m)$ are of the form

$$y^2 + xy = x^3 + x^2 + b \text{ with } b \text{ random}^2.$$

[1] Prime field operations are defined as integer addition and multiplication modulo p.

The elliptic curve methods of the benchmark use parameter files for initializing the curve, setting the base point on the curve, and setting the irreducible polynomial for multiplication in binary fields. An example of the structure of prime fields' parameter files (*p192.txt, p224.txt, p256.txt, p384.txt, p521.txt*) is given in Table 2, while Table 3 gives the typical structure of binary field parameter file. For standard cryptography benchmarks we use the following notation:

<benchmark>.<key_length>

Table 2. Example of the parameter file for a prime field GF(p), ||p||=384.

prime p (dec)	39402006196394479212279040100143613805079 73927046544666794829340424572177149687032 90472660882589380018616069731123 19
curve term b (hex)	b3312fa7e23ee7e4988e056be3f82d19181d9c6efe8 141120314088f5013875ac656398d8a2ed19d2a85c 8edd3ec2aef
base point x coord. (hex)	aa87ca22be8b05378eb1c71ef320ad746e1d3b628b a79b9859f741e082542a385502f25dbf55296c3a545 e3872760ab7
base point y coord. (hex)	3617de4a96262c6f5d9e98bf9292dc29f8f41dbd289a 147ce9da3113b5f0b8c00a60b1ce1d7e819d7a431d 7c90ea0e5f

Table 3. Example of the parameter file for a binary field GF (2^{283}).

degrees of the irreducible polynomial terms	283 12 7 5 0
curve term b (hex)	27b680ac8b8596da5a4af8a19a0303fca97f d7645309fa2a581485af6263e313b79a2f5
base point coordinate x (hex)	5f939258db7dd90e1934f8c70b0dfec2eed2 5b8557eac9c80e2e198f8cdbecd86b12053
base point coordinate y (hex)	3676854fe24141cb98fe6d4b20d02b4516ff7 02350eddb0826779c813f0df45be8112f4

The following benchmarks are representative of commonly used public-key methods:

- *dh* (Diffie-Hellman key exchange). It reads a prime suitable for Diffie-Hellman from a file, calculates shared key and writes it into a file.
- *ds* (digital signature). It reads three integers suitable for generating keys from file, generates public and private keys and writes them into files, calculates message digest of a file given as argument using the private key, writes signature into a file, and verifies the signature of the file using public key.

[2] In case of binary field, an irreducible polynomial is used when the degree of multiplication product polynomial is greater than field size.

Table 4. Our public-key benchmark suite.

Benchmark acronym	*Benchmark name*	*Input set description*	*Example input set value*
dh	Diffie-Hellman key exchange	key length	1024
ec-dh	Elliptic curve Diffie-Hellman key exchange	elliptic curve parameter file	b163.txt
ds	Digital signature of a file	key length, file to sign	2048, input_small.asc
ec-ds	Elliptic curve digital signature of a file	elliptic-curve parameter file, file to sign	b163.txt, input_small.asc
rsa	RSA encryption/decryption with exponent 65537	key length, small text file	1024, test.asc
elg	ElGamal key generation, encryption and decryption	key length, small text file	1024, test.asc
ec-elg	Elliptic curve ElGamal key generation, encryption and decryption	elliptic-curve parameter file, small text file	p192.txt, test.asc

- *rsa* (RSA encryption/decryption with exponent 2^{16}+1). Reads 1024-bit public key from a file, encrypts file passed as argument, writes ciphered text into a file, reads private key from file, reads ciphered content from a file and decrypts it using private key.
- *elg* (ElGamal key generation, encryption and decryption). It reads a suitable prime from a file and generates a public key and a private key. Encrypts file passed as argument, writes ciphered text into a file, reads private key from file, and finally decirypts the content of a file using private key.
- *ec-dh* (elliptic curve Diffie-Hellman). It reads parameter file for elliptic curve, calculates shared key and writes it into a file.
- *ec-ds* (elliptic curve digital signature). It reads private key and elliptic curve parameters from files, calculates message digest of file given as argument, writes signature to a file, and immediately verifies the signature using the public key.
- *ec-elg* (elliptic curve ElGamal key generation, encryption and decryption). It generates public-private key pair, encrypts the base point on a curve, and then immediately decrypts it.

The source code of the benchmarks, as well as the results of the experiments are available at http://www.dii.unisi.it/~giorgi/basicrypt/.

4. Methodology

The performance evaluation of our public-key cryptography benchmarks is done using the sim-outorder simulators from ARM version of the SimpleScalar toolset [*Ss97*]. The sim-outorder tool performs a detailed timing simulation of the modeled target. Simulation is execution driven, including execution down any speculative path until the detection of a fault, TLB miss, or branch misprediction. The ARM target of the SimpleScalar set supports the ARM7 integer instruction set, with the pipeline and memory system models for the Intel StrongARM SA-1110 [*Austin02*].

The simulated processor configuration is modeled after Intel ARM Xscale architecture [*Intel03*], with details of configuration given in Table 5. The sim-outorder tool was modified to model mini-data (victim) cache of Intel XScale architecture, in order to provide more accurate memory modeling.

The benchmark code was compiled using arm-linux gcc cross-compiler [*Ss02*], with optimization O2 enabled.

We compared the performance of our benchmarks with MiBench/Security benchmarks [*Guthaus01*] (Table 6), available at SimpleScalar Web site [*Ss02*]. MiBench is a set of commercially representative, freely available embedded programs. It offers different categories of real-world embedded applications, among which is the security category. MiBench security category is based on private-key methods; the only public-key application is PGP encryption/decryption, which uses RSA algorithm for signing messages. Due to problems in execution of PGP decode on SimpleScalar-ARM simulator (unimplemented system calls), we simulated only PGP encoding; we expect that PGP decode will have similar performance [*Guthaus01*]. To make the comparison as close as possible, we also used MiBench input files (small ASCII text files) as input for our benchmark simulations, where it was applicable. For reasons of space, we reported only the results of the simulations with shortest keys both for standard and elliptic curve methods (e.g. 1024, b163, p192). The results of simulations with various key lengths are available at http://www.dii.unisi.it/~giorgi/basicrypt.

When presenting our benchmark statistics (Table 7), source lines count included comments. Library files actually used in each benchmark are individuated, and their source lines counted. Static instruction count is obtained by compiling C source and library files with –S option, which produces assembly files, and by counting number of lines in assembly files. Static executable size is the number of bytes occupied on disk, while all dynamic instruction counts, loads and stores are obtained as sim-outorder simulation statistics.

5. Workload Characterization

In this Section, we characterize selected benchmarks with particular emphasis on their memory behavior.

In Table 7, static and dynamic figures for standard cryptography algorithms (dh, ds, elg) and their ECC equivalent (ec-dh, ec-ds, ec-elg) are reported. Additionally, we considered rsa algorithm as a commonly used program for public-key encryption/decryption.

To allow a fair comparison, we set the key length at a value which implies the same level of security, as discussed in Introduction (see Table 1): 1024 bits for standard public-key cryptography, 192 bits for prime field based ECC, and 163 bits for binary field based ECC.

In order to provide a direct comparison with a widely known benchmark suite, we also included the MiBench/Security suite in our experimental setup. Similarly to Table 7, we report in Table 8 statistics for MiBench/Security. In this case, it has to be observed that for some applications it would not make sense to try a "same-level of security" comparison. For example, sha is a hashing algorithm (no key length is involved). Moreover the execution time of Blowfish algorithm (bf) does not have a great dependence on key length. In fact, Blowfish only uses the key (40 to 448 bits) to set up an internal "working key" which is a fixed size structure [*Schneier96*]. For Rijndael (rj), we do not have knowledge of implementations that use less then 128 bits as key length. Therefore, the MiBench/Security benchmark comparison has to be regarded as reference to other algorithms commonly used in security applications.

Initially, we analyzed the type and number of operations. Almost all benchmarks (except ec-elg and rj) have more than 50% integer ALU operations, therefore they are very computation intensive. *Using ECC involves a lower number of dynamically executed instructions* (fourth column, Table 7), compared with the same statistics for standard cryptography. The percentage of memory operations is practically the same for ECC algorithms and standard cryptography (Figure 3).

The number of memory references is higher in standard cryptography than in ECC (last two columns, Table 7), but a further analysis is needed to see if they really contribute to the total execution time. In fact, the actual performance of these benchmarks depends also on the characteristics of instruction and data access patterns. In particular, branch prediction schemes could have different impact on performance. In Figure 4, 5, 6, and 7 we analyzed these factors.

Branch prediction is one of the factors influencing instruction pattern generation. We simulated three different schemes to see if we can achieve any performance improvement: not taken, an 8k 2-level predictor, and an 8k bimodal predictor. All predictors used a 2k BTB except the not taken strategy. Figure 4 shows the results for these schemes. The most efficient predictor is the bimodal. We used this predictor in the rest of our study.

In Figure 5 – on the left – where we selected 24 cycles for memory latency and 1KB for Level-1 Instruction Cache and 1KB for Level-1 Data Cache, it appears that ECC algorithms take comparable time to execute than their corresponding standard version.

When memory latency is increased to 96 cycles (right side of the Figure 5) the execution time is always higher for ECC algorithms than in the case of standard methods.

In Figure 5, we separated the contribution to execution time due to memory stall (upper portion of bars). Memory stalls account for a high percentage of the execution time. For example, in case of 96 cycles, more than 80% for ec-dh and ec-elg and more than 60% for ec-ds. This means that a particular care has to be taken for the memory subsystem, when considering the implementation of ECC algorithms. This is particularly true for mobile systems such as PDA or wireless phones, where memory could be not very fast and caches have a small size due to power constraints.

Even if ECC uses a lower number of memory operations, the working set is larger or the locality of instruction and data accesses is somewhat worse than in standard cryptography.

Table 5. Simulated architecture.

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 int ALU, 1 int MUL/DIV
Instruction L1 cache	32 kB, 32-way
Data L1 cache	32 kB, 32-way
L1 cache hit latency	1 cycle
L1 cache block size	32 B
L2 cache	none
Mini-data cache	2 kB
Memory latency (cycles)	24 , 96
Memory bus width (bytes)	4

Table 6. MIBENCH/SECURITY benchmarks.

Benchmark acronym	*Benchmark name*	*Input set description*	*Input set value*
bf.enc	Blowfish encrypt	file to encrypt	input_small.asc
bf.dec	Blowfish decrypt	file to decrypt	output_small.enc
rj.enc	Rijndael encrypt	file to encrypt	input_small.asc
rj.dec	Rijndael decrypt	file to decrypt	output_small.enc
sha	SHA	file to hash	input_small.asc
pgp.enc	PGP encode	small text file	test.asc
pgp.dec	PGP decode	small text file	test.enc

Both latter problems can be overcome through the use of larger caches. Therefore, we considered a more detailed analysis of the caches. As our goal is to analyze this situation in the case of embedded systems, we setup typical configurations of XScale processor, with only Level-1 Instruction+Data split caches and no Level-2 cache.

To analyze the reasons of the higher stall time of Figure 5, we report in Figure 6 a detail of the Data and Instruction MPI in the case of 1-Kbyte caches. The Misses-Per-Instruction (MPI) metric is useful as it provides a figure that is directly proportional to the CPI (Cycles Per Instruction) contribution due to memory stall [*Kessler91*]. To determine appropriate caches for these algorithms, we considered cache sizes from 256-bytes through 32 kB (Figure 7).

The cache size range is appropriate for our case as the working set size is rather small (as typical in embedded systems applications [*Guthaus01*]).

For a 32K-bytes cache size the MPI approaches zero.

To reduce the execution time, we should have at least 16 kB of instruction cache and 2 kB of data cache available for these applications.

If the constraints of our system design required a slower (lower-power) main memory, the stall time due to memory access could be even higher (Figure 5, right portion, where main memory latency is 96 cycles).

For MiBench/Security, we observe that private-key algorithms (bf and rj) have a much higher number of data misses (Figure 6, 1 kB cache size), while the public-key based pgp.enc nicely compares with same values as the standard cryptography benchmarks that we selected. For Instruction misses, MiBench private-key algorithms (bf, rj) compare more directly with ECC algorithms rather than with standard public-key methods (dh, ds, elg).

Table 7. Public-key benchmark statistics (cache size 1 kB, memory latency 24, optimization O2).

Benchmark name	Source lines (application +app_library)	Instruction count		Static executable size (bytes)	Dynamic executable size (bytes)		Loads dynamic	Stores dynamic
		static (application+app_library)	dynamic		text	data		
dh.1024	59/8365	214/20226	57,229,355	117,865	257,916	33,874,376	21,312,670	12,908,525
ec-dh.b163	99/10821	349/30884	23,101,446	120,094	258,236	33,874,792	6,288,459	3,782,031
ec-dh.p192	89/9356	306/21217	37,766,470	119,738	258,204	33,874,744	13,506,830	8,688,365
ds.1024	246/8522	806/20773	116,430,745	119,473	311,188	33,928,020	38,270,187	26,441,914
ec-ds.b163	330/10978	1157/31431	88,948,893	122,285	312,036	33,928,948	25,523,628	18,783,618
ec-ds.p192	330/9513	1009/24701	109,024,645	121,069	311,796	33,928,724	34,748,326	25,005,446
rsa.1024	215/8365	776/20226	30,812,587	118,319	232,038	33,848,504	11,147,041	6,824,826
elg.1024	192/8365	699/20226	231,072,055	118,127	259,388	33,876,060	87,473,709	52,744,618
ec-elg.b163	99/10821	384/30884	74,021,406	118,363	258,332	33,874,920	18,339,259	11,276,216
ec-elg.p192	97/9356	333/21217	71,062,489	118,395	258,284	33,874,904	18,048,378	11,258,092

Table 8. MIBENCH/SECURITY: Benchmark statistics (cache size 1 kB, memory latency 24, optimization O2).

Benchmark name	Source lines	Instruction count		Static executable size (bytes)	Dynamic executable size (bytes)		Loads dynamic	Stores dynamic
		static	dynamic		text	data		
bf.enc	2,302	7,749	52,410,681	968,691	190,900	33,808,760	19,971,638	17,391,227
rj.enc	1,773	12,134	24,907,400	998,449	199,364	33,874,440	10,829,945	3,789,546
sha	269	793	13,286,108	955,216	187,540	33,801,380	2,296,237	963,951
pgp	34,858	73,244	39,105,600	1,451,988	217,088	450,560	8,609,025	4,690,129

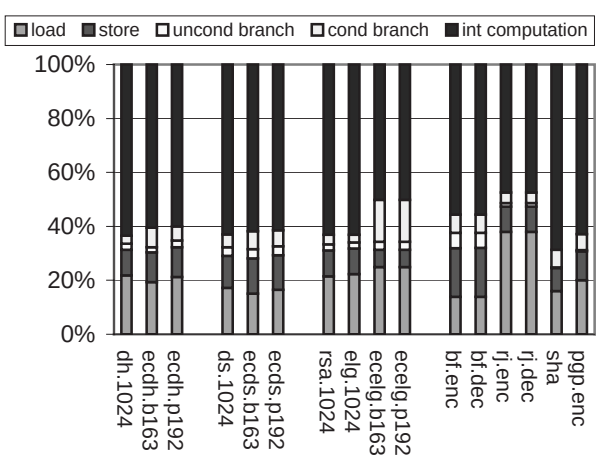

Figure 3. Dynamic instruction class profile.

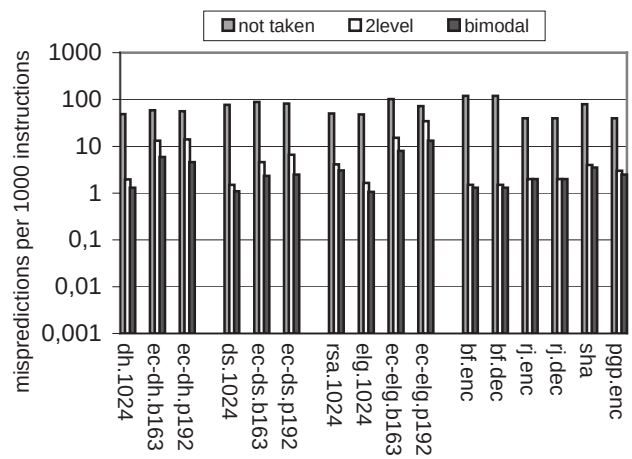

Figure 4. Branch prediction rates per 1000 instructions for several schemes.

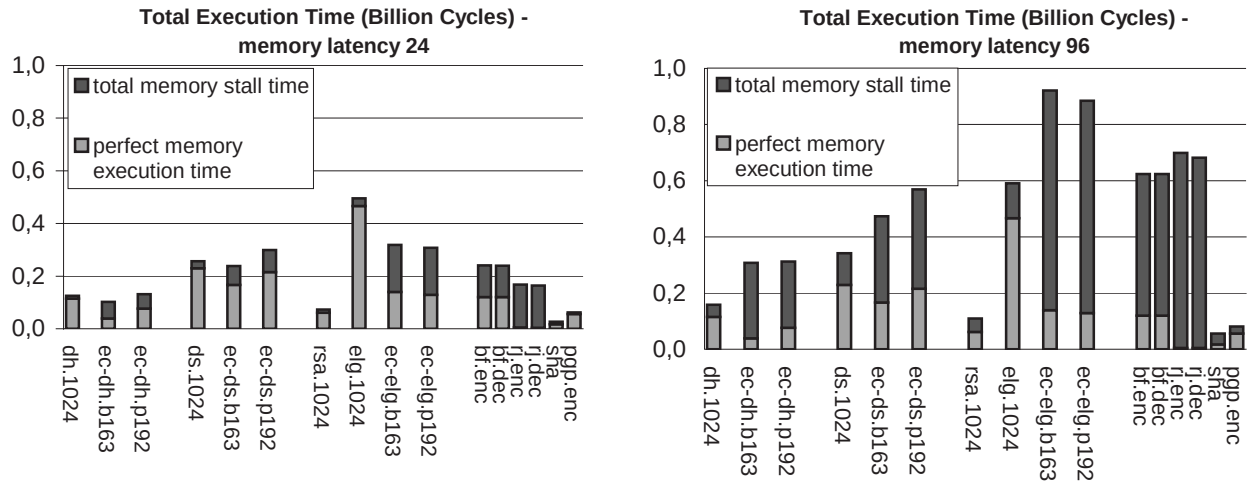

Figure 5. Total execution time (cache 1kB + 1kB) with memory latency of 24 and 96 cycles.

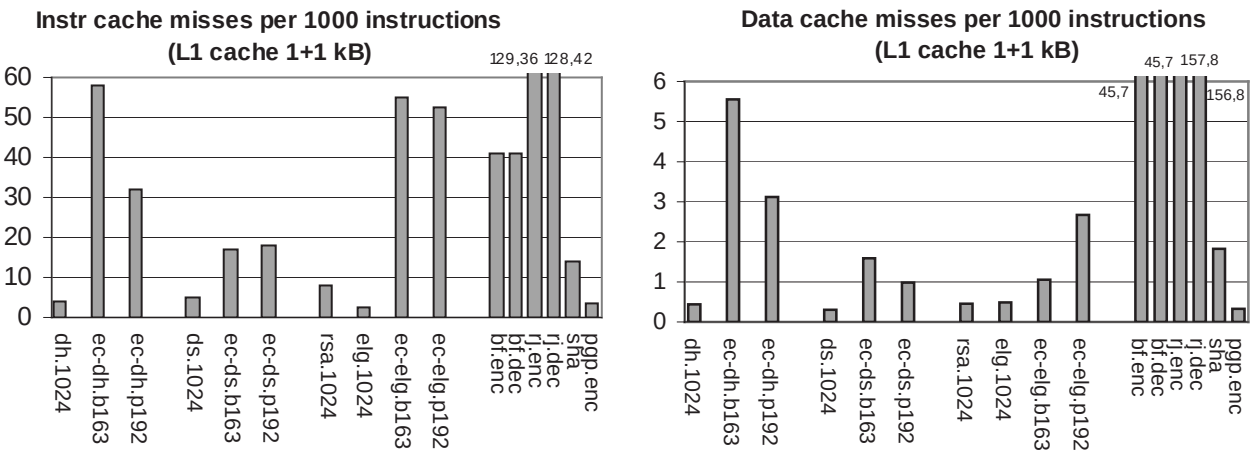

Figure 6. Instruction and data cache misses per 1000 instructions (L1 cache 1+1 KB) and comparison with MiBench.

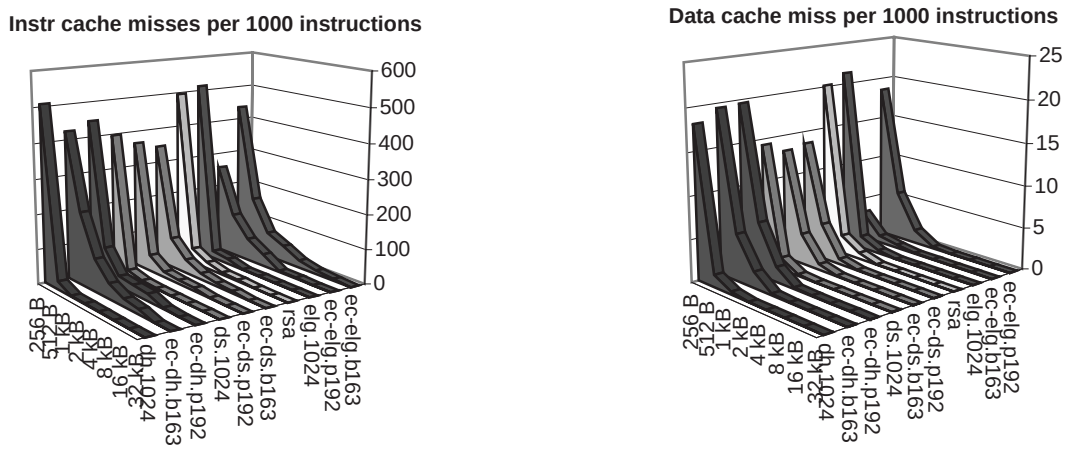

Figure 7. Instruction and data cache misses per 1000 instructions.

6. Related Work

There is an intensive research ongoing in improving the efficiency of elliptic curve operations, as well as their performance analysis. A workload characterization of some public-key and private-key algorithms, including their elliptic-curve equivalents for binary polynomial fields is found in [*Fiskiran02*]. They characterize operations in Diffie-Hellman, digital signature, and ElGamal elliptic curve methods, and demonstrate that all these algorithms can be implemented efficiently with a very simple processor. [*Hankerson00*] presents an extensive and careful study of the software implementation of NIST-recommended elliptic curves over binary fields. In [*Gupta02*], the authors give the first estimate of performance improvements that can be expected by adding ECC support in SSL protocol.

In [*Guthaus01*] MiBench is compared with SPEC2000 benchmarks, which characterize a workload for general-purpose computers. The common characteristics of security applications are low cache miss rate, more than 50% integer ALU operations, and low level of parallelism. In [*Milenkovic03*], MiBench suite and SimpleScalar simulator for ARM target are used for a performance evaluation of typical cache design issues for embedded systems.

7. Conclusions

The main contributions of our paper are: i) setup of kernel benchmark set for studying elliptic curve and standard public-key methods and ii) studying the performance of public-key methods in embedded environments.

We found that using ECC cryptography involves a higher number of dynamically executed instructions. However, the locality of instruction and data accesses is worse than in standard cryptography.

Instruction and data locality matters to ECC performance and appropriate caches should be adopted in order to keep total execution time at acceptable levels. The importance of memory stall and thus the importance of appropriate caches is more relevant in the case of ECC cryptography than in the case of standard cryptography. Branch mispredictions do not affect performance significantly, as the percentage of branch instructions is decreased.

References

[*Austin02*] T. Austin, E. Larson, D. Ernst, "SimpleScalar: An Infrastructure for Computer System Modelling", *IEEE Computer* , Volume 35, Issue: 2, Feb. 2002, pp. 59–67.

[*Blake03*] V. Gupta, S. Blake-Wilson, B. Moeller, C. Hawk, "ECC Cipher Suites for TLS", Internet draft, June 2003, http://www.ietf.org/internet-drafts/draft-ietf-tls-ecc-03.txt.

[*Diffie76*] W. Diffie, M. E. Hellman, "New Directions in Cryptography", *IEEE Trans. on Information Theory*, Vol. IT-22, Nov. 1976, pp. 644-654.

[*ElGamal85*] T. ElGamal, "A public key cryptosystem and a signature scheme based on discrete logarithms", *IEEE Trans. on Information Theory*, Vol. 31, 1985, pp. 469-472.

[*Fiskiran02*] A. M. Fiskiran, R. B. Lee, "Workload Characterization of Elliptic Curve Cryptography and other Network Security Algorithms for Constrained Environments", *Proc. of 5th IEEE Workshop on Workload Characterization (WWC-5)*, Nov. 2002, pp. 127-137.

[*Gupta02*] V. Gupta, S. Gupta, S. C. Chang, "Performance Analysis of Elliptic Curve Cryptography for SSL", *WiSe'02*, Atlanta, USA, 2002.

[*Guthaus01*] M. Guthaus, J. Ringerberg, T. Austin, T. Mudge, R. Brown, "MiBench: A free, commercially representative embedded benchmark suite", *Proc. of 4th Workshop on Workload Characterization*, Dec. 2001.

[*Hankerson00*] D. Hankerson, J. Lopez, A. Menezes, "Software Implementation of Elliptic Curve Cryptography over Binary Fields", *Proc. of CHES 2000 Conference*, Springer-Verlag, 2000, pp. 1-24.

[*IEEE1363-00*] IEEE Standard Specifications for Public-Key Cryptography, 1363-2000, IEEE Computer Society, Jan. 2000, http://grouper.ieee.org/groups/1363/

[*Intel03*] Intel Corporation, "The Intel Xscale Microarchitecture Technical Summary", ftp://download.intel.com/design/intelxscale/XscaleDatasheet4.pdf

[*Kessler91*] R. E. Kessler, "Analysis of Multi-Megabyte Secondary CPU Cache Memories", Ph.D. Thesis, Univ. of Wisconsin, Computer Sciences, Tech. Report 31032, 1991.

[*Knuth81*] D. E. Knuth, *The Art of Computer Programming, Vol. 2: Seminumerical Algorithms*, Addison-Wesley, 1981.

[*Menezes01*] A. Menezes, *Elliptic Curve Public Key Crypto-systems*, Kluwer Academic Publishers, Boston, USA, 2001.

[*Milenkovic03*] A. Milenkovic, M. Milenkovic, N. Barnes, "A Performance Evaluation of Memory Hierarchy in Embedded Systems", *Proc. of 35th SSST*, Mar. 2003.

[*Miracl02*] Miracl big integer library Web site, http://indigo.ie/~mscott/

[*NIST95*] NIST, Secure Hash Standard, FIPS pub180-1, 1995.

[*NIST00*] Digital Signature Standard, National Institute of Standards and Technology, FIPS pub186-2, Jan. 2000.

[*RSA02*] RSA Laboratories' FAQs about Today's Cryptography, http://www.rsasecurity.com/rsalabs/faq

[*Schneier96*] B. Schneier, Applied Cryptography – Protocols, Algorithms and Source Code in C, 2nd ed., New York, J. Wiley & Sons, 1996.

[*Ss97*] D. C. Burger, T. M. Austin, "The SimpleScalar Tool Set, Version 2.0", Tech. Report CS-TR-97-1342, University of Wisconsin-Madison, June 1997.

[*Ss02*] SimpleScalar LLC, http://www.simplescalar.com