Analysis of Efficient Techniques for Fast Elliptic Curve Cryptography on x86-64 based Processors

Patrick Longa, and Catherine Gebotys
Department of Electrical and Computer Engineering,
University of Waterloo, Canada,
{plonga,cgebotys}@uwaterloo.ca

Abstract. In this work, we analyze and present experimental data evaluating the efficiency of several techniques for speeding up the computation of elliptic curve point multiplication on emerging x86-64 processor architectures. In particular, we study the efficient combination of such techniques as elimination of conditional branches and incomplete reduction to achieve fast field arithmetic over $\mathbb{F}_p$. Furthermore, we study the impact of (true) data dependencies on these processors and propose several generic techniques to reduce the number of pipeline stalls, memory reads/writes and function calls. We also extend these techniques to field arithmetic over $\mathbb{F}_{p^2}$, which is utilized as underlying field by the recently proposed Galbraith-Lin-Scott (GLS) method to achieve higher performance in the point multiplication. By efficiently combining all these methods with state-of-the-art elliptic curve algorithms we obtain high-speed implementations of point multiplication that are up to 31% faster than the best previous published results on similar platforms. This research is crucial for advancing high-speed cryptography on new emerging processor architectures.

Keywords. Elliptic curve cryptosystem, point multiplication, field arithmetic scheduling, incomplete reduction, data dependence, pipeline stall, x86-64 instruction set, software implementation.

1 INTRODUCTION

Elliptic Curve Cryptography (ECC), discovered independently by Miller [38] and Koblitz [30] in mid 80's, has gained widespread acceptance in recent years, taking over a central role in public-key cryptography that was previously exclusive to the classic RSA. This technological shift is partially explained by ECC's reduced key length requirement to achieve certain security level. The latter brings many benefits such as reduced memory footprint, lower power consumption and faster execution time, among others.

Point multiplication, defined as $[k]P$, where the point P has order r and is on an elliptic curve E over a prime field $\mathbb{F}_p$ (i.e., $P \in E(\mathbb{F}_p)$) and $k \in [1, r-1]$ is an integer, is the central and most time-consuming operation in ECC over prime fields. Hence, its efficient realization has gained increasing importance for the industry and research communities and a plethora of methods have been proposed for speeding up this operation at its various computational levels. For instance, different studies have proposed methods using efficient arithmetic representations for the scalar [41][44][11][31], efficiently computable endomorphisms [20][19], fast precomputation schemes [36][34], efficient point formulae [7][8][25][37] and long integer modular arithmetic [29][40][9][47], and improved curve forms with fast arithmetic [6][12][26][27]. Still, these research efforts usually need to be complemented with further analysis and actual implementations on different platforms that permit one to assess their practical effectiveness in the real world. Accordingly, many studies in that direction have focused on efficient implementations on constrained 8-bit microcontrollers [21][45], 32-bit embedded devices [46][17], Graphical Processing Units (GPUs) [43], processors based on the Cell Broadband Engine Architecture (CBEA) [10], 32-bit x86-based processors [5][4], among others. Nevertheless, there are very few studies focusing on the analysis of efficient techniques for high-speed ECC point multiplication especially targeting the most recent x86-64 based processors, and this work tries to fill that gap.

Modern CPUs from the notebook, desktop and server classes are decisively adopting the 64-bit x86 instruction set (a.k.a. x86-64) developed by AMD [1]. The most relevant features of this new instruction set are the expansion of the general-purpose registers (GPRs) from 32 to 64 bits, the execution of arithmetic and logical operations on 64-bit integers and an increment in the number of GPRs, among other enhancements. In addition, these processors usually exhibit a highly pipelined architecture, improved branch predictors and complex execution stages that offer parallelism at the instruction level. Thus, this increasingly high complexity brings new paradigms to the software and compiler developer.

In this work, we analyze several techniques and evaluate their effectiveness to devise highly efficient field and point arithmetic for ECC over prime fields on architectures based on the x86-64 ISA. Specifically, we study the impact of branch misprediction for modular reduction and demonstrate quantitatively the benefit of eliminating conditional branches in modular addition, subtraction and multiplication/division by small constants. Moreover, we optimally combine this approach with the well-known technique of incomplete reduction (IR) [47] to achieve further cost reductions. Also, we analyze the influence of deeply pipelined architectures in the ECC point multiplication execution. In particular, we notice that the increased number of stages in certain pipeline architectures can make (true) data dependencies between contiguous field operations particularly expensive because these can potentially stall the execution for several clock cycles. These dependencies fall in the category of read-after-write (RAW), which are typically found between several field operations when the result of an operation is required as input by the following operation. In this work, we demonstrate the potentially high cost incurred by these dependencies, which is hardly avoided by compilers and dynamic schedulers in processors, and propose *three* techniques to reduce its effect: field arithmetic scheduling, merging of field operations and merging of point operations.

The techniques above are applied to modular operations using a prime p, which are used for performing the $\mathbb{F}_p$ arithmetic in ECC over prime fields. However, these techniques are generic and can also be extended to different scenarios using other underlying fields. For instance, Galbraith et al. [19] recently proposed a faster way to do ECC that exploits an efficiently computable endomorphism to accelerate the execution of point multiplication over a quadratic extension field (a.k.a. GLS method). Accordingly, we extend our analysis to $\mathbb{F}_{p^2}$ arithmetic and show that the proposed techniques also lead to significant gains in performance in this case.

Our extensive tests assessing the techniques under analysis cover at least one representative x86-64 based CPU from each processor class: 1.66GHz Intel Atom N450 from the notebook (and netbook) class, 2.66GHz Intel Core 2 Duo E6750 from the desktop class, and 2.83GHz Intel Xeon E5440 and 2.6GHz AMD Opteron 252 from the server (and workstation) class.

Finally, to assess their effectiveness for a full point multiplication, the proposed techniques are applied to state-of-the-art implementations using Jacobian and (extended) Twisted Edwards coordinates on the targeted processors. Our measurements show that the proposed optimizations (in combination with state-of-the-art point formulas/coordinate systems, precomputation schemes and exponentiation methods) significantly speed up the execution time of point multiplication, surpassing with considerable margins previous state-of-the-art implementations. For instance, we show that a 256-bit point multiplication for the case of Jacobian and (extended) Twisted Edwards coordinates can be computed in only 337000 and 281000 cycles, respectively, on one core of an Intel Core 2 Duo processor. Compared to the previous results of 468000 and 362000 cycles (respect.) by Hisil et al. [27], our results achieve improvements of about 28% and 22% (respect.). In the case of the GLS method, for Jacobian and (extended) Twisted Edwards coord., we compute one point multiplication in about 252000 and 229000 cycles (respect.) on the same processor, which compared to the best previous results by Galbraith et al. [18][19] (326000 and 293000 cycles, respect.) translate to improvements of about 23% and 22%, respectively.

This work extends significantly the analysis and results presented by the authors in [35].

Our work is organized as follows. In Section 2, we briefly describe relevant features of x86-64 based processors, elliptic curves over prime fields and the recently proposed GLS method. In Section 3, we analyze the impact of combining the incomplete reduction technique with elimination of conditional branches to achieve high-performance field arithmetic. In Section 4, we analyze the effect of (true) data dependencies between contiguous field operations on different processors and propose several practical techniques to minimize it. In Section 5, we extend the proposed techniques to quadratic extension fields and study their impact when using the GLS method. Finally, in Section 6, we present our timings for point multiplication and compare them to the best previous results.

2 PRELIMINARIES

2.1 x86-64 based Processor Architectures

For a background in computer architectures and experimental analysis on processors based on the x86-64 ISA, readers are referred to [28] and [15][16], respectively.

Modern CPUs from the notebook, desktop and server classes are rapidly adopting the x86-64 ISA proposed by AMD [1]. This new instruction set involves GPRs of 64 bits, arithmetic and logical operations on 64-bit integers, an increment in the number of GPRs, among other enhancements. Most importantly, modern processors based on this architecture exhibit deep pipelines with a high number of stages. For instance, experiments presented in [16] suggest that Intel Atom, Intel Core 2 Duo and AMD processors have pipelines with 16, 15 and 12 stages, respectively. There are *two* aspects related to the latter that are of special interest in this work: the high cost of branch mispredictions and data dependencies.

Branch Predictors and Conditional Branches:

The performance of branch predictors can be evaluated through the equation:

$$CPI = CPI_{ideal} + (\%Branch \times \%Branch\_misprediction \times penalty)\,, \qquad (1)$$

where:
CPI: cycles per instruction.
CPI_{ideal} : ideal CPI without hazards. Typically, equal to 1 for non-superscalar processors.
$\%Branch$: percentage of instructions that are branches.
$\%Branch\_misprediction$: percentage of unsuccessful predictions.
penalty: cost in cycles per misprediction. Roughly, equal to the number of stages in the pipeline.

Let us consider the following conditioned executing statements:

```
if condition
   execute1
else execute2
```

which is typically translated to the following pseudocode using conditional branches:

```
if condition branch to label1
execute2
branch to label2
label1:
  execute1
label2:
   ...
```

If the predictor guesses correctly whether to branch to **label1** or **label2** most of the time, the penalty introduced by mispredictions is minimal. Sophisticated branch predictors such as local and global branch predictors combined with 2-level adaptive techniques (found in processors from Intel and AMD) can obtain about 97% of guesses correct for certain applications. This translates to small penalties with CPIs only increased by 9% for example (assuming that $\%Branch = 0.1$, $penalty = 15$ and $CPI_{ideal} = 0.5$ in (1)). Otherwise, the penalty can be extremely high. Following the example above, the CPI increases by 250% if $\%Branch\_misprediction = 0.5$.

Unfortunately, the last scenario is expected to happen in certain field operations, such as addition and subtraction, in which the reduction step (typically implemented with a conditional branch) is required 50% of the time in a “random” pattern. Hence, it is better to eliminate conditional branches in these circumstances, as already implemented in some crypto libraries [22]. There are *two* possible solutions to achieve this:

- Using look-up tables: two values, 0 and the actual value required by the reduction step, are pre-stored and then selected accordingly during modular reduction using indexed indirect addressing mode.
- Through branch predication: using *predicated* move instructions (e.g., `cmov` in x86) to load 0 or the actual value required by the reduction step.

The previous approaches follow the same idea: the reduction step is performed every time. If the reduction is not actually required it is performed with the value 0, which does not affect the final result. These techniques have some additional advantages. They tend to reduce the code size and allow a more flexible scheduling of instructions that can lead to faster execution times.

In Section 3.2, we analyze the impact of eliminating conditional branches during modular reduction, and present experimental data quantifying the gain in performance on x86-64 based CPUs. Moreover, we efficiently combine this approach with the incomplete reduction technique.

Data Dependencies:

Let *i* and *j* be the computer orders of instructions I_i and I_j in a given program flow. We say that instruction I_j depends on instruction I_i if [28]:

$$[W(I_i)\cap R(I_j)]\cup[R(I_i)\cap W(I_j)]\cup[W(I_i)\cap W(I_j)]\neq\varnothing, \tag{2}$$

where $R(I_x)$ is the set of memory locations or registers read by I_x and $W(I_x)$ is the set of memory locations or registers written by I_x. We can distinguish *three* cases:

- True (data) dependence (or Read-After-Write, RAW): if $i<j$ and $W(I_i)\cap R(I_j)$, i.e., if I_j reads something written by I_i.
- Anti-dependence (or Write-After-Read, WAR): if $i<j$ and $R(I_i)\cap W(I_j)$, i.e., if I_i reads a location later updated by I_j.
- Output dependence (or Write-After-Write, WAW): if $i<j$ and $W(I_i)\cap W(I_j)$, i.e., if both I_i and I_j write the same location.

Modern out-of-order processors and compilers deal relatively well with anti- and output dependencies through register renaming. However, true or RAW dependencies cannot be removed in the strict sense of the term and are more dangerous to the performance of architectures exploiting *Instruction-Level Parallelism* (ILP).

In the case that instructions I_i and I_j , where $i<j$, have a RAW dependence and are “close” to each other such that a hazard is imminent, the pipeline needs to stall a number of cycles proportional to the time it takes I_i to complete its pipeline latency. There are *two* approaches to minimize the appearance of pipeline stalls: by instruction scheduling and using data forwarding. In particular, the former can be taken over by the compiler, the out-of-order processor or the programmer (or a combination of these). In Section 4, we discuss

several software-based techniques that minimize the number of pipeline stalls caused by RAW dependencies between consecutive field operations on processors with deep pipelines such as x86-64 based CPUs.

2.2 Elliptic Curve Cryptography

For a background in elliptic curves, the reader is referred to [24]. The standard elliptic curve (also known as short Weierstrass curve) over a prime field $\mathbb{F}_p$ has the equation:

$$E_w: \ y^2 = x^3 + ax + b \,, \tag{3}$$

where $a,b \in \mathbb{F}_p$ and $\Delta = 4a^3 + 27b^2 \neq 0$. However, different curve forms exhibiting faster group arithmetic have been studied during the last few years. A good example is given by Twisted Edwards. This curve form, proposed in [3], is a generalization of Edwards curves [12] and has the equation:

$$E_{tedw}: ax^2 + y^2 = 1 + dx^2y^2 \,, \tag{4}$$

where $a,d \in \mathbb{F}_p$ are distinct nonzero elements.

The points on equations (3) or (4) and the point at infinity, denoted by $\mathcal{O}$, form an *abelian* group $(E(\mathbb{F}_p),+)$ with a group law mainly consisting of *two* basic point operations: doubling (2*P*) and addition (*P*+*Q*) of points. In this setting, the main operation is known as point multiplication, which is denoted by $[k]P$, where $P \in E(\mathbb{F}_p)$, and can be seen as the computation $[k]P = P + P + \ldots + P$, where *P* is added $(k-1)$ times.

Because affine coordinates (point representation using (x, y) coordinates; denoted by $\mathcal{A}$) are expensive over prime fields due to costly field inversions, the use of projective coordinates with the form $(X:Y:Z)$ is preferred. In this work, we have chosen the following coordinate systems for assessing the techniques under analysis:

- Jacobian coordinates (denoted by $\mathcal{J}$), where each projective point $(X:Y:Z)$ corresponds to the affine point $(X/Z^2, Y/Z^3)$, $Z \neq 0$. In this case, the standard equation (3) acquires the form $Y^2 = X^3 + aXZ^4 + bZ^6$.
- Combined homogenous/extended Twisted Edwards coordinates (denoted by $\mathcal{E}/\mathcal{E}^e$) [26]. In the extended version $\mathcal{E}^e$, the auxiliary coordinate *T* is added to the homogenous representation $(X:Y:Z)$ such that each projective point $(X:Y:Z:T)$ corresponds to $(X/Z, Y/Z, 1, T/Z)$ in affine, where $T = XY/Z$. With $\mathcal{E}/\mathcal{E}^e$ coordinates, the projective form of equation (4) is given by $(aX^2 + Y^2)Z^2 = Z^4 + dX^2Y^2$.

State-of-the-art formulas using $\mathcal{J}$ and $\mathcal{E}/\mathcal{E}^e$ coordinates can be found in [32] and [26], respectively, and their costs are summarized in Table 1. Although variations to costs displayed in Table 1 exist (for instance, those obtained by trading multiplications for squarings [32][13]), these sometimes involve an increased number of "small" operations such as additions, subtractions and multiplications/divisions by constants. On various platforms (including x86-64 based processors), that extra cost may not be negligible. Formulas in Table 1 have been selected so that the *overall* cost is minimal on the targeted platforms. The complete set of revised formulas with *optimal* number of multiplications and squarings and *minimal* number of "small" operations have been compiled in Appendix A.

TABLE 1
Costs of point operations on Weierstrass and Twisted Edwards curves [1].

Point Operation	Coord.	Weierstrass ($a = -3$)	Coord.	Twisted Edwards ($a = -1$)
Doubling	$2\mathcal{J} \to \mathcal{J}$	4M + 4S	$2\mathcal{E} \to \mathcal{E}$	4M + 3S
Mixed addition	$\mathcal{J} + \mathcal{A} \to \mathcal{J}$	8M + 3S	$\mathcal{E}^e + \mathcal{A} \to \mathcal{E}$	7M
General addition	$\mathcal{J} + \mathcal{J} \to \mathcal{J}$	11M + 3S (1)	$\mathcal{E}^e + \mathcal{E}^e \to \mathcal{E}$	8M
Mixed Doubling-Addition	$2\mathcal{J} + \mathcal{A} \to \mathcal{J}$	13M + 5S	$(2\mathcal{E})^e + \mathcal{A} \to \mathcal{E}$	11M + 3S
General Doubling-Addition	$2\mathcal{J} + \mathcal{J} \to \mathcal{J}$	16M + 5S (1)	$(2\mathcal{E})^e + \mathcal{E}^e \to \mathcal{E}$	12M + 3S

(1) Using cached values.

The Galbraith-Lin-Scott (GLS) Method:

In this method by Galbraith et al. [19], ECC computations are performed on the quadratic twist of an elliptic curve over $\mathbb{F}_{p^2}$ with an efficiently computable homomorphism $\psi(x, y) \to (\alpha x, \beta y)$, $\psi(P) = \lambda P$. Then, following [20], $[k]P$ is computed as a multiple point multiplication with form $[k_0]P + [k_1](\lambda P)$, where k_0 and k_1 have approx. half the bitlength of *k*.

For the case of the Weierstrass form, given equation (3) defined over $\mathbb{F}_p$, the quadratic twist E'_w over $\mathbb{F}_{p^2}$ of $E_w(\mathbb{F}_{p^2})$ is given by the equation:

$$E'_w : y^2 = x^3 + \mu^2 a x + \mu^3 b\,, \tag{5}$$

where μ is a non-square in $\mathbb{F}_{p^2}$. Following [19], we fix $p \equiv 3 (\mathrm{mod}\, 4)$ and $\mu = 2 + i \in \mathbb{F}_{p^2}$ such that $i = \sqrt{-1} \in \mathbb{F}_p$ and $\psi(x, y) = (\frac{\mu}{\mu^p} \cdot \overline{x}, \sqrt{\mu^3 / \mu^{3p}} \cdot \overline{y})$, where $\overline{x}$, $\overline{y}$ denote the Galois conjugates of x, y, respectively.

For the case of Twisted Edwards, given equation (4) defined over $\mathbb{F}_p$, the quadratic twist E'_{tedw} over $\mathbb{F}_{p^2}$ of $E_{tedw}(\mathbb{F}_{p^2})$ is given by the equation:

$$E'_{tedw} : \mu a x^2 + y^2 = 1 + \mu d x^2 y^2\,, \tag{6}$$

where μ is a non-square in $\mathbb{F}_{p^2}$. Following [18], we fix $p \equiv 3 (\mathrm{mod}\, 4)$ and $\mu = 2 + i \in \mathbb{F}_{p^2}$ such that $i = \sqrt{-1} \in \mathbb{F}_p$ and $\psi(x, y) = (\sqrt{\mu^p / \mu} \cdot \overline{x}, \overline{y})$.

Since for our case $\mathbb{F}_{p^2} = \mathbb{F}_p(\sqrt{-1})$, i.e., $i = \sqrt{-1} \in \mathbb{F}_p$, elements in $\mathbb{F}_{p^2}$ can be represented by $a + bi$, where $a, b \in \mathbb{F}_p$. For instance, an $\mathbb{F}_{p^2}$ multiplication, as suggested in [19], can be performed using Karatsuba method [29] as $(a + bi) \cdot (c + di) = (ac - bd) + (bc + ad)i = (ac - bd) + ((a + b)(c + d) - ac - bd)i$, which requires 3 $\mathbb{F}_p$ multiplications and

[1] Field operations: I = inversion, M = multiplication, S = squaring, Add = addition, Sub = subtraction, Mul*x* = multiplication by *x*, Div*x* = division by *x*, Neg = negation.

5 $\mathbb{F}_p$ additions/subtractions.

Galbraith et al. showed that, in practice, the new method runs about 16% faster than the best previous implementation due to Gaudry and Thomé [23] on an Intel Core 2 Duo. For complete details and the security implications, the reader is referred to [18][19].

In this work, we analyze the performance of the field and point arithmetic exploiting our optimizing techniques on *two* "traditional" implementations (on Weierstrass and Twisted Edwards curves) and *two* implementations using the GLS method (again, one per curve). For the traditional case, we have written the underlying field arithmetic over $\mathbb{F}_p$ using hand-written assembly language. In this case, we consider for maximal speed-up a pseudo-Mersenne prime of the form $p = 2^m - c$, where $m = n \cdot w$ on an *w*-bit platform, $n \in \mathbb{Z}^+$, and c is a "small" integer (i.e., $c < 2^w$). These primes are highly efficient for performing modular reduction a prime *p*, and support other optimizations such as elimination of conditional branches. On the other hand, for the GLS method we reuse the very efficient modules for field arithmetic over $\mathbb{F}_{p^2}$ provided with the crypto library MIRACL [42]. In this case, $\mathbb{F}_{p^2}$ arithmetic provided by MIRACL considers a Mersenne prime with the form $2^t - 1$ (i.e., *t* is prime).

3 Optimizing the Modular Reduction

In this section, we evaluate the performance gain of *two* techniques, namely incomplete reduction and elimination of conditional branches, and combine them to devise highly efficient field arithmetic with very fast modular reduction for operations such as addition, subtraction and division/multiplication by constants. We also show that incomplete reduction is not exclusive to addition/subtraction and can be easily extended to other operations, and that subtraction does not necessarily benefit from incomplete reduction when *p* is a smartly chosen pseudo-Mersenne prime. All tests described in this section were performed on our assembly language module implementing the field arithmetic over $\mathbb{F}_p$ and compiled with GCC version 4.4.1.

3.1 Incomplete Reduction (IR)

This technique was introduced by Yanik et al. [47]. Given two numbers in the range $[0, p-1]$, it consists of allowing the result of an operation to stay in the range $[0, 2^s - 1]$ instead of executing a complete reduction, where $p < 2^s < 2p - 1$, $s = n \cdot w$, w is the basic wordlength (typically, $w = 8, 16, 32, 64$) and *n* is the number of words. If the modulus is a pseudo-Mersenne prime of the form $2^m - c$ such that $m = s$ and $c < 2^w$, then the method gets even more advantageous. In the case of addition, for example, the result can be reduced by first discarding the carry bit in the most significant word and then adding the correction value *c*, which fits in a single *w*-bit register. Also note that this last addition does not produces an overflow because $2 \times (2^m - c - 1) - (2^m - c) < 2^m$. The procedure is illustrated for the case of modular addition in Algorithm 3.1(b), for which the reduction step described above is performed in Step 3. As can be seen in Algorithm 3.1(a), a complete reduction requires additionally the execution of Step 4 that performs a subtraction $r - p$ in case

$p \leq r < 2^m$, where r is the partial result from Step 2.

Yanik et al. [47] also showed that subtraction can benefit from IR when using a prime p of arbitrary form. However, we show in the following that for primes of special form, such as pseudo-Mersenne primes, that is not necessarily the case.

Algorithm 3.1 Modular addition with a pseudo-Mersenne prime

INPUT: integers $a, b \in [0, p-1]$, $p = 2^m - c$, $m = n \cdot w$, where $n, w, c \in \mathbb{Z}^+$ and $c < 2^w$

OUTPUT: $r = a + b \pmod{p}$ or $r = a + b \pmod{2^m}$

(a) With Complete Reduction

```
1. $carry = 0$
2. For $i$ from $0$ to $n-1$ do
   2.1. $(carry, r[i]) \leftarrow a[i] + b[i] + carry$
3. If $carry = 1$
   3.1. $carry = 0$
   3.2. $(carry, r[0]) \leftarrow r[0] + c$
   3.3. For $i$ from $1$ to $n-1$ do
        3.3.1. $(carry, r[i]) \leftarrow r[i] + carry$
4. Else
   4.1. $borrow = 0$
   4.2. For $i$ from $0$ to $n-1$ do
        4.2.1. $(borrow, R[i]) \leftarrow r[i] - p[i] - borrow$
   4.3. If $borrow = 0$
        4.3.1. $r \leftarrow R$
5. Return $r$
```

(b) With Incomplete Reduction

```
1. $carry = 0$
2. For $i$ from $0$ to $n-1$ do
   2.1. $(carry, r[i]) \leftarrow a[i] + b[i] + carry$
3. If $carry = 1$
   3.1. $carry = 0$
   3.2. $(carry, r[0]) \leftarrow r[0] + c$
   3.3. For $i$ from $1$ to $n-1$ do
        3.3.1. $(carry, r[i]) \leftarrow r[i] + carry$
4. Return $r$
```

Algorithm 3.2 Modular subtraction with a pseudo-Mersenne prime and complete reduction

INPUT: integers $a, b \in [0, p-1]$, $p = 2^m - c$, $m = n \cdot w$, where $n, w, c \in \mathbb{Z}^+$ and $c < 2^w$

OUTPUT: $r = a - b \pmod{p}$

```
1. $borrow = 0$
2. For $i$ from $0$ to $n-1$ do
   2.1. $(borrow, r[i]) \leftarrow a[i] - b[i] - borrow$
3. If $borrow = 1$
   3.1. $carry = 0$
   3.2. For $i$ from $0$ to $n-1$ do
        3.2.1. $(carry, r[i]) \leftarrow r[i] + p[i] + carry$
4. Return $r$
```

Modular Subtraction. Let us consider Algorithm 3.2. After Step 2 we obtain the completely reduced value $r = a - b$ if $borrow = 0$. If, otherwise, $borrow = 1$ then this bit is discarded and the partial result is given by $r = a - b + 2^m$, where $b > a$. This value is incorrect, because it has the extra addition with 2^m. In step 3.2, we compute $r + p = (a - b + 2^m) + (2^m - c) = a - b - c + 2^{m+1}$, where $2^m < a - b - c + 2^{m+1} < 2^{m+1}$ since

$-2^m + c < a - b < 0$. Then, by simply discarding the final carry from Step 3.2 (i.e., by subtracting 2^m) we obtain the correct, completely reduced result $a - b - c + 2^{m+1} - 2^m = a - b + p$, where $0 < a - b + p < p$. Since Algorithm 3.2 gives the correct result without evaluating both values of *borrow* after Step 2 (similarly to the case of *carry* in Alg. 3.1(b)), there is no need for incomplete reduction in this case.

Nevertheless, there are other types of "small" operations that may be benefited by the use of IR. We analyze in the following the cases that are useful to the setting of ECC over prime fields.

Modular Addition $\boldsymbol{a + b \,(\mathrm{mod}\, p)}$ with IR, where $\boldsymbol{a \in [0, p-1]}$ and $\boldsymbol{b \in [0, 2^m - 1]}$. In this case, after addition we get $0 \le a + b \le 2^{m+1} - c - 2$, where $2^m < 2^{m+1} - c - 2 < 2^{m+1}$ for practical values of *m*. Thus, if there is no final carry the result *r* is incompletely reduced such that $r \in [0, 2^m - 1]$, as wanted. Otherwise, for the case $2^m \le a + b \le 2^{m+1} - c - 2$ we discard the carry and add the correction value c such that $0 < c \le a + b - 2^m + c \le 2^m - 2 < 2^m$ to obtain an incompletely reduced result $r \in [0, 2^m - 1]$. Consequently, Algorithm 3.1(b) also allows adding two terms where one of them can be in incompletely reduced form.

Modular Multiplication by 3 with IR, where $\boldsymbol{a \in [0, p-1]}$. If this operation is performed by executing $a + a + a \,(\mathrm{mod}\, p)$, internally, the first addition $r = a + a \,(\mathrm{mod}\, p)$ can be left incompletely reduced using Algorithm 3.1(b). Then, following the proof in the previous subsection, the final result $r + a \,(\mathrm{mod}\, p) \in [0, 2^m - 1]$ can be obtained by adding the incompletely reduced value *r* with the completely reduced operand *a*.

Modular Division $\boldsymbol{a/2 \,(\mathrm{mod}\, p)}$ with IR, where $\boldsymbol{a \in [0, 2^m - 1]}$. This operation is illustrated when using IR by Alg. 3.3(b). If the value a is even, then a division by 2 can be directly applied through Steps 3 and 4, where $(carry, r[i]) \leftarrow (carry, r[i])/2$ represents the concurrent assignments $r[i] \leftarrow \left\lfloor (carry \cdot 2^{(i+1).w} + r[i])/2 \right\rfloor$ and $carry \leftarrow r[i] (\mathrm{mod}\, 2)$. In this case, if $a \in [0, 2^m - 2]$ then the result $r \in [0, 2^{m-1} - 1]$ is completely reduced since $2^{m-1} - 1 << 2^m - c$ for practical values of *m*, such that $c < 2^w$ and $w < m - 1$. If, otherwise, the operand *a* is odd, we first add p to a in Step 2.2 to obtain an equivalent from the residue class that is even. Then, $2^m - c + 1 < p + a < 2^{m+1} - c - 1$, where the partial result has $m + 1$ bits maximum and is stored in $(carry, r)$. The operation is then completed by dividing by 2 through Steps 3 and 4, where the final result $2^{m-1} - (c-1)/2 < (p+a)/2 < 2^m - (c+1)/2$. Hence, the result is incompletely reduced because $2^m - c \le 2^m - (c+1)/2 \le 2^m - 1$. If the result needs to be completely reduced then, for the case that $(p+a)/2 \in [p, 2^m - \lceil (c+1)/2 \rceil]$, one needs to additionally compute a subtraction with *p* such that $0 \le (p+a)/2 - p < (c-1)/2 < 2^m - c$, as performed in Steps 6 and 7 of Alg. 3.3(a).

It is also interesting to note that in the case that input a is in completely reduced form, i.e., $a \in [0, p-1]$, after Step 4 in Alg. 3.3(b) we get $2^{m-1} - (c+1)/2 < (p+a)/2 < 2^m - c$, which is in completely reduced form.

Algorithm 3.3 Modular division by 2 with a pseudo-Mersenne prime

INPUT: integers $a \in [0, 2^m - 1]$, $p = 2^m - c$, $m = n \cdot w$, where $n, w, c \in \mathbb{Z}^+$ and $c < 2^w$

OUTPUT: $r = a/2 \pmod p$ or $r = a/2 \pmod{2^m}$

(a) With Complete Reduction

```
1. $carry = 0$
2. If $a$ is $odd$
   2.2. For $i$ from $0$ to $n-1$ do
        2.2.1. $(carry, r[i]) \leftarrow a[i] + p[i] + carry$
3. $(carry, r[n-1]) \leftarrow (carry, r[n-1])/2$
4. For $i$ from $n-2$ to $0$ do
   4.1. $(carry, r[i]) \leftarrow (carry, r[i])/2$
5. $borrow = 0$
6. For $i$ from $0$ to $n-1$ do
   6.1. $(borrow, R[i]) \leftarrow r[i] - p[i] - borrow$
7. If $borrow = 0$
   7.1. $r \leftarrow R$
8. Return $r$
```

(b) With Incomplete Reduction

```
1. $carry = 0$
2. If $a$ is $odd$
   2.2. For $i$ from $0$ to $n-1$ do
        2.2.1. $(carry, r[i]) \leftarrow a[i] + p[i] + carry$
3. $(carry, r[n-1]) \leftarrow (carry, r[n-1])/2$
4. For $i$ from $n-2$ to $0$ do
   4.1. $(carry, r[i]) \leftarrow (carry, r[i])/2$
5. Return $r$
```

To evaluate in practice the advantage of using incomplete reduction, we implemented in assembly language both versions with and without IR of each operation discussed in this section. In Table 2, we summarize our results on the targeted Intel and AMD processors.

TABLE 2
Cost (in cycles) of modular operations when using incomplete reduction (IR) against complete reduction (CR); $p = 2^{256} - 189$.

Modular Operation	Atom N450			Core 2 Duo E6750			Opteron 252		
	IR	CR	Cost reduction (%)	IR	CR	Cost reduction (%)	IR	CR	Cost reduction (%)
Addition	31	45	31%	20	25	20%	13	20	35%
Multiplication by 2	27	40	33%	19	24	21%	10	17	41%
Multiplication by 3	43	69	38%	28	43	35%	15	23	35%
Division by 2	57	61	7%	20	25	20%	11	18	39%

As can be seen in Table 2, in our experiments using the pseudo-Mersenne prime $p = 2^{256} - 189$ we obtain significant reductions in cost ranging from 7% to up to 41% when using IR.

It is important to note that, because multiplication and squaring may accept inputs in the range $[0, 2^m - 1]$, an operation using IR can precede any of these two operations. Thus, the reduction process (which is left "incomplete" by the operation using IR) is fully completed by these multiplications or squarings without any additional cost. If care is taken when implementing point operations, virtually all additions and multiplications/divisions by small constants can be implemented with IR because most of them have results that are later required by multiplications or squarings only. See Appendix A for details about the scheduling of field operations $\mathbb{F}_p$ suggested for point formulas using Jacobian and (extended) Twisted Edwards coordinates.

3.2 Elimination of Conditional Branches

Conditional branches may be expensive in several modern processors with deep pipelines if the prediction strategy fails in most instances in a particular implementation. Recovering from a mispredicted branch requires the pipeline to flush, wasting several clock cycles that may increase the overall cost significantly. In particular, the reduction portion of modular addition, subtraction and other similar operations is traditionally expressed with a conditional branch. For example, let us consider the evaluation in Step 3 of Algorithm 3.1(b) for performing a modular addition with IR. Because $a,b\in[0,p-1]$ and $2^m-p=c$ (again considering $p=2^m-c$ and $m=s$), where c is a relatively small number such that $2^m\approx p$ for practical estimates, the possible values for *carry* after computing $a+b$ in Step 2, where $(a+b)\in[0,2p-2]$, are (approximately) equally distributed and describe a "random" sequence for all practical purposes. In this scenario, only an average of 50% of the predictions can be correct in the best case. Similar results are expected for conditional branches in other operations (see Algorithms 3.1-3.3).

To avoid the latter effect, it is possible to eliminate conditional branches by using techniques such as look-up tables or branch predication (cf. §2.1). In Fig. 3.1, we illustrate the replacement of the conditional branch in Step 3 of Alg. 3.1(b) by a predicated `move`

FIGURE 3.1

Steps 3 and 4 of Alg. 3.1(b) for executing modular addition using IR, where $p=2^{256}-189$. The conditional branch is replaced by (a) `cmov` instruction (initial values `%rax=0`, `%rcx=189`) and (b) look-up table using indexed indirect addressing mode (preset values `%rax=0`, `(%rcx)=0`, `8(%rcx)=189`). Partial addition $a+b$ from Step 2 is stored in registers `%r8-r11` and final result is stored in x`(%rdx)`. x86-64 assembly code uses AT&T syntax.

(a)

```
> ⋮
> cmovnc %rax,%rcx
> addq %rcx,%r8
> movq %r8,8(%rdx)
> adcq $0,%r9
> movq %r9,16(%rdx)
> adcq $0,%r10
> movq %r10,24(%rdx)
> adcq $0,%r11
> movq %r11,32(%rdx)
> ret
```

(b)

```
> ⋮
> adcq $0,%rax
> addq (%rcx,%rax,8),%r8
> movq %r8,8(%rdx)
> adcq $0,%r9
> movq %r9,16(%rdx)
> adcq $0,%r10
> movq %r10,24(%rdx)
> adcq $0,%r11
> movq %r11,32(%rdx)
> ret
```

instruction (Fig. 3.1(a)) and by a look-up table with indexed indirect addressing (Fig. 3.1(b)). In both cases, the strategy is to perform an addition with 0 if there is no carry-out (i.e., the reduction step is not required) or an addition with $c = 189$, where $p = 2^{256} - 189$, if there is carry-out and the computation $(a + b - 2^{256}) + 189$ is necessary. On the targeted CPUs, branch predication performs slightly better in most cases. This conclusion is platform-dependent and, in our case, may be due to the faster execution of `cmov` in comparison to the memory access required by the look-up table approach.

To quantify in practice the difference in performance obtained by implementing modular arithmetic with and without conditional branches, we tested both schemes on the targeted Intel and AMD processors. The results are summarized in Table 3. For addition, subtraction and division by 2, we use Algorithms 3.1(a), 3.2 and 3.3(a), respectively. In the case of addition and division by 2 using IR, we use Algorithms 3.1(b) and 3.3(b), respectively. Multiplication by 2 is a variation of the addition operation for which $2a$ is computed as $a + a(\bmod p)$.

TABLE 3
Cost (in cycles) of modular operations without conditional branches (w/o CB) against operations using conditional branches (with CB); $p = 2^{256} - 189$.

Modular Operation	Atom N450			Core 2 Duo E6750			Opteron 252		
	w/o CB	With CB	Cost reduction (%)	w/o CB	With CB	Cost reduction (%)	w/o CB	With CB	Cost reduction (%)
Subtraction	34	37	8%	21	37	43%	16	23	30%
Addition with IR	31	35	11%	20	37	46%	13	21	38%
Addition	45	43	−4.4%	25	39	36%	20	23	13%
Multiplication by 2 with IR	27	34	21%	19	38	50%	10	19	47%
Multiplication by 2	40	42	5%	24	38	37%	17	20	15%
Division by 2 with IR	57	66	14%	20	36	44%	11	18	39%
Division by 2	61	70	13%	25	39	36%	18	27	33%

As shown in Table 3, the cost reductions obtained by eliminating CBs can be as high as 50%. Remarkably, the greatest performance gains are obtained in the cases of operations exploiting IR. For instance, on Core 2 Duo, an addition using IR reduces its cost in 46% when CBs have been eliminated in comparison to only the 36% reduction obtained by an addition with complete reduction. Thus, elimination of CBs favors more strongly modular arithmetic using IR. This is due to the fact that modular operations exploiting IR allow very compact implementations that are even easier to schedule efficiently when branches are removed. It is also interesting to note that, when comparing Core 2 Duo's and Opteron's performances, gains are higher for the former processor, which has more stages in its pipeline. Roughly speaking, the gain obtained by eliminating (poorly predictable) CBs on these architectures grows proportionally with the number of stages in the pipeline. In contrast, the gains on Intel Atom are significantly smaller since the pipeline execution and ILP on this in-order processor are much less efficient and, hence, the relative cost of misprediction penalty reduces, as can be deduced from eq. (1).

Following the conclusions above, we have implemented ECC point formulas such that the gain obtained by combining IR and the elimination of CBs is maximal. The reader is referred to Appendix A for details about the cost of point formulas in terms of field operations.

Next, we evaluate the cost of point doubling and doubling-addition (using Jacobian coordinates) when their "small" field operations are implemented with complete or incomplete reduction and with or without conditional branches. For the analysis, we use the revised doubling formula (1), Section 4.2, introduced in [35] and the doubling-addition formula (3.5), Section 3.2, introduced in [31]. The results are shown in Table 4.

TABLE 4
Cost (in cycles) of point operations with Jacobian coordinates when using incomplete reduction (IR) or complete reduction (CR) and with or without conditional branches (CB); $p = 2^{256} - 189$.

Point operation	Atom N450			Core 2 Duo E6750			Opteron 252		
	CR and CBs	CR and no CBs	IR and no CBs	CR and CBs	CR and no CBs	IR and no CBs	CR and CBs	CR and no CBs	IR and no CBs
Doubling	3480	3430	3381	1184	1094	1051	910	824	803
Relative reduction (%)	-	1%	3%	-	8%	11%	-	9%	12%
Doubling-addition	8828	8697	8663	2656	2468	2443	2037	1851	1849
Relative reduction (%)	-	1%	2%	-	7%	8%	-	9%	9%
Estimated relative reduction for 256-bit point multiplication (%)	-	1%	3%	-	8%	10%	-	9%	11%

As can be seen in Table 4, the computing costs of point doubling and doubling-addition on the AMD processor reduce in 12% and 9%, respectively, by combining the elimination of conditional branches with the use of incomplete reduction. Without taking into account precomputation and the final inversion to convert to affine, these reductions represent about 11% of the computing cost of point multiplication. A similar figure is observed for Intel Core 2 Duo in which doubling and doubling-addition are reduced by approx. 11% and 8%, respectively. These savings represent a reduction of about 10% in the cost of point multiplication (again, without considering precomputation and the final inversion). In contrast, following previous observations (see Table 3) the techniques are less effective on architectures such as Intel Atom, where the ILP is less powerful and branch misprediction penalty is relatively less expensive. In this case, the cost reduction of point multiplication is only about 3%.

A similar analysis to the one provided in this section can be performed on other platforms to determine whether conditional branches should be removed. In such case, it would be necessary to test both the use of look-up tables and branch predication to determine which one is the most efficient replacement for branches. Also, some testing would help to determine the performance improvement obtained by these approaches in combination with incomplete reduction.

4 Minimizing the Effect of Data Dependencies

In this section, we analyze (true) data dependencies between "close" field operations and propose *three* techniques to minimize their effect in the point multiplication performance.

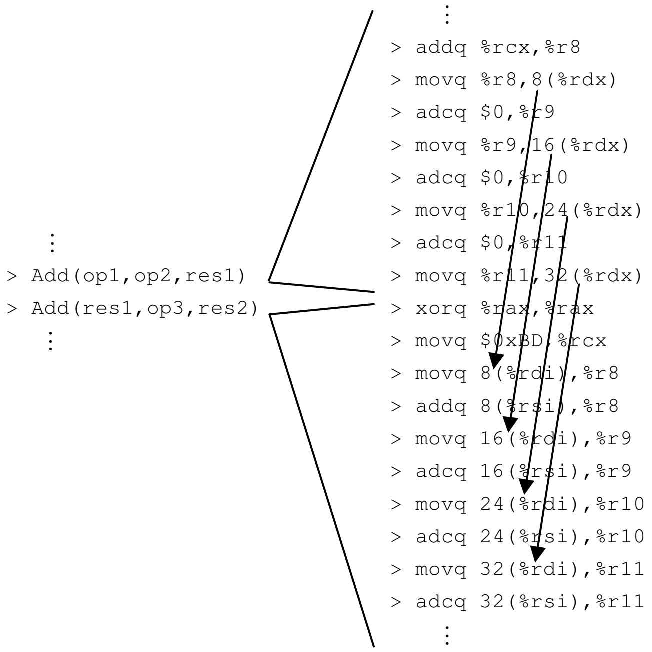
Corollary 4.1. Let I_i and I_j be write and read instructions, respectively, holding data dependence, i.e., $W(I_i) \cap R(I_j) \neq \varnothing$, where $i < j$ and I_i and I_j are scheduled to be executed at the i^{th} and j^{th} cycle, respectively, in a non-superscalar pipelined architecture. Then, if $\rho = j - i < \delta_{write}$ the pipeline is to be stalled for at least $(\delta_{write} - \rho)$ cycles, where δ_{write} specifies the number of cycles required by the write instruction I_i to complete its pipeline latency after instruction fetching.

Although Corollary 4.1 considers an *ideal* non-superscalar pipeline, it allows us to simplify the analysis on more complex processors. In particular, the value δ_{write}, which strongly depends on the particular characteristics of a given architecture, can be considered for practical purposes roughly equal to the pipeline size. Note, however, that there are hardware techniques such as data forwarding that allow a significant reduction in the value δ_{write} by sending back the result of an operation into the decode stage so that this result is immediately available to a coming instruction before the current instruction commit/store the output. Unfortunately, in our application most modular operations are not able to efficiently exploit forwarding in case the result is required by the following operation because several consecutive writings to memory are involved in the process. To illustrate this problem let us consider the execution of two consecutive field additions in Figure 4.1. For the remainder, given a field operation "$*$", the operation `res`$\leftarrow$`op1`$*$`op2` is denoted by `operation(op1,op2,res)`.

As can be seen in Figure 4.1, results stored in memory in the last stage of the first addition are read in the first stage of the second addition. In this example, *four* consecutive writings to memory and then *four* consecutive readings need to be performed because operands are 256-bit long distributed over *four* 64-bit registers. In this case, if $\delta_{write} > \rho_x$ for at least one of the dependences x indicated by arrows then the pipeline is expected to stall for at least $(\delta_{write} - \rho_x)$ cycles. Then, for the writing/reading sequence in Figure 4.1, the pipeline is roughly stalled by $\max(\delta_{write} - \rho_x)$ for $0 \leq x < 4$.

Definition 4.1. Two field operations $OP_i(op_m, op_n, res_p)$ and $OP_j(op_r, op_s, res_t)$ are said to be *data dependent at the field arithmetic level* if $i < j$ and $res_p = op_r$ or $res_p = op_s$, where OP_i and OP_j denote the field operations performed at positions i^{th} and j^{th} during a

FIGURE 4.1
Field additions with RAW dependencies on an x86-64 CPU ($p = 2^{256} - 189$). High-level field operations are in the left column and low-level assembly instructions corresponding to each field operation are to the right. In this example, destination x`(%rdx)` (first field addition) = source x`(%rdi)` (second field addition). Dependencies are indicated by arrows.



```
   ⋮
> Add(op1,op2,res1)
> Add(res1,op3,res2)
   ⋮
```

```
   ⋮
> addq %rcx,%r8
> movq %r8,8(%rdx)
> adcq $0,%r9
> movq %r9,16(%rdx)
> adcq $0,%r10
> movq %r10,24(%rdx)
> adcq $0,%r11
> movq %r11,32(%rdx)
> xorq %rax,%rax
> movq $0xBD,%rcx
> movq 8(%rdi),%r8
> addq 8(%rsi),%r8
> movq 16(%rdi),%r9
> adcq 16(%rsi),%r9
> movq 24(%rdi),%r10
> adcq 24(%rsi),%r10
> movq 32(%rdi),%r11
> adcq 32(%rsi),%r11
   ⋮
```

program execution, and op and res are registers holding the inputs and result, respectively. Then, this is called a *contiguous* data dependence in the field arithmetic if $j - i = 1$, i.e., OP_i and OP_j are consecutive in the executing sequence.

For the applications targeted in this work all field operations follow a similar writing/reading pattern to that one shown in Figure 4.1, and hence, two contiguous, data dependent field operations hold several data dependencies *x* between their internal write/read instructions. Following Definition 4.1 and Corollary 4.1, contiguous data dependencies pose a problem when $\delta_{write} > \rho_x$ in a given implementation or processor architecture, in which case the pipeline is stalled by roughly $\max(\delta_{write} - \rho_x)$ cycles for all dependencies *x*. Note that at fewer dependent write/read instruction pairs (i.e., at smaller field sizes) the expression $\max(\delta_{write} - \rho_x)$ grows as well as the number of potential stalled cycles. Similarly, at larger basic wordlengths w $\max(\delta_{write} - \rho_x)$ is expected to increase, worsening the effect of contiguous data dependencies.

Next, we propose *three* techniques that help to reduce the number of contiguous data dependencies in the field arithmetic and study several practical scenarios in which this would allow us to improve the execution performance of point multiplication.

4.1 Field Arithmetic Scheduling

A straightforward solution to eliminate contiguous data dependencies is to perform a careful scheduling of the field operations inside point formulas in such a way that data-dependent field operations are not contiguous. For all practical purposes, we can consider that any field operation has an executing latency δ_{ins} longer than the latency of a write instruction, i.e., $\delta_{ins} > \delta_{write}$. Hence, by inserting any "independent" field operation between two consecutive operations holding contiguous data dependence we guarantee that the new relative positions $\rho_{new,x}$ of the data-dependent instructions accomplishes $\rho_{new,x} = \rho_x + \delta_{ins} > \delta_{write}$ for all data dependencies *x*, where ρ_x denotes the original relative positions between data-dependent write/read instructions.

We have tested several field operation "arrangements" to observe the latter behavior on different processors. We detail here a few of our experiments with field multiplication on an Intel Core 2 Duo. For example, let us consider the field multiplication sequences given in Table 5. As can be seen, Sequence 1 involves a series of "ideal" data-independent field multiplications, where the output of a given operation is not an input to the immediately following operation. In this case, the execution reaches its maximal performance with an average of 110 cycles per multiplication because for any pair of data-dependent multiplications we have $\rho_x >> \delta_{write}$. Contrarily, the second sequence is highly dependent because each output is required as input in the following operation. In this case, $\delta_{write} > \rho_x$ for at least one dependence *x*. This is the worst-case scenario with an average of 128 cycles per multiplication, which is about 14% less efficient than the "ideal" case. We have also studied other possible arrangements such as Sequence 3, in which operands of Sequence 2 have been reordered. This slightly amortizes the impact of contiguous data dependencies because ρ_x is increased, improving the performance to 125 cycles/multiplication.

TABLE 5
Various sequences of field operations with different levels of contiguous data dependence.

Sequence 1	Sequence 2	Sequence 3
`> Mult(op1,op2,res1)`	`> Mult(op1,op2,res1)`	`> Mult(op2,op1,res1)`
`> Mult(op3,op4,res2)`	`> Mult(res1,op3,res2)`	`> Mult(op3,res1,res2)`
`> Mult(res1,op5,res3)`	`> Mult(res2,op4,res3)`	`> Mult(op4,res2,res3)`
`> Mult(res2,op6,res4)`	`> Mult(res3,op5,res4)`	`> Mult(op5,res3,res4)`

Similarly, we have also tested the effect of contiguous data dependencies on other field operations. In Table 6, we summarize the most representative field operation "arrangements" and their costs. As can be seen, the reductions in cost obtained by switching from an execution with strong contiguous data dependence (worst-case scenario with Sequence 2) to an execution with no contiguous data dependencies (best-case scenario with Sequence 1) range from approximately 9% to up to 33% on an Intel Core 2 Duo. Similar results were observed for the targeted AMD Opteron and Intel Xeon processors, where the high performance of their architectures significantly reduce relative positions ρ_x between their data-dependent write/read instructions, increasing the value $\max(\delta_{write} - \rho_x)$. Thus, minimizing contiguous data dependencies is expected to improve the execution of point multiplication on all these processors. In contrast, Sequences 1 and 2 perform similarly on processors such as Intel Atom, in which the much less powerful architecture tends to increase values ρ_x such that $\delta_{write} < \rho_x$ for all dependencies *x*.

TABLE 6
Average cost (in cycles) of modular operations using best-case (no contiguous data dependencies, Sequence 1) and worst-case (strong contiguous data dependence, Sequence 2) “arrangements” ($p = 2^{256} - 189$, on a 2.66GHz Intel Core 2 Duo E6750).

Modular Operation	Core 2 Duo E6750		
	Sequence 1	Sequence 2	Cost reduction (%)
Subtraction	21	23	9%
Addition with IR	20	24	17%
Multiplication by 2 with IR	19	23	17%
Multiplication by 3 with IR	28	34	18%
Division by 2 with IR	20	30	33%
Squaring	101	113	11%
Multiplication	110	128	14%

4.2 Merging Point Operations

This technique complements and increases the gain obtained by scheduling field operations. As expected, in some cases it is not possible to eliminate all contiguous data dependencies in a point formula. A clever way to increase the chances of eliminating more of these dependencies is by “merging” successive point operations into unified functions.

For example, let us consider the following sequence of field operations for computing a point doubling using Jacobian coordinates: $2(X_1, Y_1, Z_1) \rightarrow (X_1, Y_1, Z_1)$

```
> Sqr(Z1,t3)          | > Mult(X1,t2,t4)  | > Sqr(t1,t2)
> Sqr(Y1,t2)          | > Mult(t1,t0,t3)  | > DblSub(t2,t4,X1)[1] ●
> Add(X1,t3,t1)       | > Sqr(t2,t0)      | > Sub(t4,X1,t2)       ●
> Sub(X1,t3,t3)       | > Div2(t3,t1)     | > Mult(t1,t2,t4)      ●
> Mult3(t3,t0)      ● | > Mult(Y1,Z1,Z1)  | > Sub(t4,t0,Y1)       ●
```

In total, there are *five* contiguous data dependencies between field operations (denoted by "•") in the sequence above. Note that the last stage accounts for most dependencies, which are very difficult to eliminate. However, if another point doubling follows, one could merge

[1] `DblSub(b,c,a)` represents the operation $a \leftarrow b - 2c \pmod p$. See Section 4.3.

both successive operations and be able to reduce the number of contiguous data-dependent operations. Consider, for example, the following arrangement of *two* consecutive doublings

```
> Sqr(Z1,t3)         | > Mult(t1,t0,t3)  | > DblSub(t2,t4,X1)   | > Mult3(t3,t1)
> Sqr(Y1,t2)         | > Sqr(t2,t0)      | > Sub(t4,X1,t2)  ●   | > Sqr(Y1,t2)
> Add(X1,t3,t1)      | > Div2(t3,t1)     | > Add(X1,t3,t5)      | > Mult(t1,t5,t3)
> Sub(X1,t3,t3)      | > Mult(Y1,Z1,Z1)  | > Mult(t1,t2,t4)     | > Mult(t2,X1,t4)
> Mult3(t3,t0)    ●  | > Sqr(t1,t2)      | > Sub(X1,t3,t3)      | > Div2(t3,t1)
> Mult(X1,t2,t4)     | > Sqr(Z1,t3)      | > Sub(t4,t0,Y1)      | > …
```

As can be seen, the sequence above (instructions from the second doubling are in **bold**) allows us to further reduce the number of dependencies from *five* to only *two*.

In ECC implementations, it appears natural to merge successive doubling operations or a doubling and an addition. Efficient elliptic curve point multiplications $[k]P$ use the non-adjacent form (NAF) in combination with some windowing strategy to recode the scalar *k*. For instance, width-*w* NAF (*w*NAF) guarantees at least w successive doublings between point additions. Also, one can exploit the efficient doubling-addition operation by [31] for Jacobian coordinates or the combined (dedicated) doubling-(dedicated) addition by [26] for Twisted Edwards coordinates (see Table 1). Hence, an efficient solution for these systems is to merge $(w-1)$ consecutive doublings (for an optimal choice of *w*) in a separate function and merge each addition with the precedent doubling in another function. On the other hand, if an efficient doubling-addition formula is not available for certain setting, then it is suggested to merge w consecutive doublings in one function and have the addition in a separate function. Note that for different coordinate systems/curve forms/point multiplication methods the optimal merging strategy may vary or include different operations. Remarkably, a side-effect of this technique is that the number of function calls to point formulas is also reduced dramatically.

4.3 Merging Field Operations

This technique consists in merging various field operations with common operands to implement them in a joint function. There are *two* scenarios where this approach becomes attractive:

- The result of a field operation is required as input by a following operation: merging reduces the number of memory reads/writes and eliminates directly potential contiguous data dependencies.
- Operands are required by more than one field operation: merging reduces the number of memory reads/writes.

We remark that the feasibility of merging certain field operations depends strictly on the chosen platform and the number of general purpose registers available to the programmer/ compiler. Also, before deciding on a merging option implementers should analyze and test the increase in the code size and how this affects the performance of the cache for example. Accordingly, in the setting of ECC over prime fields, multiplication and squaring are not recommended to be merged with other operations if multiple functions containing these operations are necessary. The code increase could potentially affect the cache performance.

Taking into account the considerations above, we suggest the following merged field operations on x86-64 based processors using Jacobian and Twisted Edwards coordinates: $a-2b(\bmod p)$, $a+a+a(\bmod p)$, and the merging of $a-b(\bmod p)$ and $(a-b)-2c(\bmod p)$. We remark that this list is not exhaustive. Different platforms with more registers may enable a much wider range of merging options. Also, other possibilities for merging could be available for different coordinate systems and/or underlying fields (for instance, see Section 5.2 for the merging options suggested for ECC implementations over quadratic extension fields).

To illustrate the impact of scheduling field operations, merging point operations and merging field operations, we show in Table 7 the cost of point doubling using Jacobian coordinates when using these techniques in comparison with a naïve implementation with a high number of dependencies.

TABLE 7

Cost (in cycles) of point doubling using Jacobian coordinates with different number of contiguous data dependencies and the corresponding reduction in the cost of point multiplication. "Unscheduled" refers to implementations with a high number of dependencies (in this case, 10 per doubling and 13 per doubling-addition). Implementations that apply scheduling of field operations, merging of point operations and merging of field operations are listed under "Scheduled and merged" (in this case, 1.25 depend. per doubling and 3 per doubling-addition); $p = 2^{256} - 189$

Point operation	Atom N450		Core 2 Duo E6750		Opteron 252	
	"Unscheduled"	"Scheduled and merged"	"Unscheduled"	"Scheduled and merged"	"Unscheduled"	"Scheduled and merged"
Doubling	3390	3332	1115	979	786	726
Relative reduction (%)	-	2%	-	12%	-	8%
Estimated reduction for 256-bit point multiplication (%)	-	1%	-	9%	-	5%

As can be seen in Table 7, by reducing the number of dependencies from *ten* to about *one* per doubling, minimizing function calls and reducing the number of memory reads/writes, we are able to reduce the cost of a doubling by 12% and 8% on Intel Core 2 Duo and AMD Opteron processors, respectively. It is also important to note that on a processor such as AMD Opteron, which has a smaller pipeline and consequently less lost due to contiguous data dependencies (smaller δ_{write} with roughly the same values ρ_x as Core 2 Duo), the estimated gain obtained with these techniques in the point multiplication is lower (5%) in comparison with the Intel processor (9%). Finally, following our analysis in previous sections, Intel Atom only obtains a very small improvement in this case because contiguous data dependencies do not affect the execution performance (see Section 4.1).

The reader is referred to Appendix A for details about the suggested field arithmetic scheduling, merging of point operations and merging of field operations for point formulas using Jacobian and (extended) Twisted Edwards coordinates.

5 OPTIMIZATIONS FOR THE $\mathbb{F}_{p^2}$ FIELD ARITHMETIC

The techniques and optimizations described so far are not exclusive to the popular $\mathbb{F}_p$ field arithmetic. In fact, the scheduling of field operations, merging of field operations and merging of point operations are generic and can be extended to different finite fields with similar benefits and results. In this section, we analyze how the aforementioned techniques can be applied to the arithmetic over a quadratic extension field $\mathbb{F}_{p^2}$. This application has gained sudden importance thanks to the recently proposed GLS method [19], which exploits an efficiently computable homomorphism to speed up the execution of point multiplication over $\mathbb{F}_{p^2}$.

For our study, we consider the highly-optimized assembly module of the field arithmetic over $\mathbb{F}_{p^2}$ written by M. Scott [42]. This module exploits the "nice" Mersenne prime $p = 2^{127} - 1$, which allows a very simple reduction step with no conditional branches. Although IR can also be applied to this scenario, in practice we observe that the gain is negligible on the platforms under study. Future work may consider the analysis of this technique on different platforms.

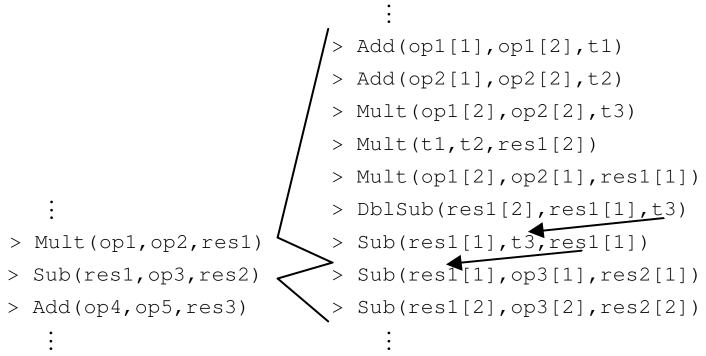
5.1 Scheduling of field operations

As described in Section 2.2, each $\mathbb{F}_{p^2}$ operation consists of a few field operations over $\mathbb{F}_p$. Thus, the analysis of data dependencies and scheduling of operations should be performed taking into account this underlying layer. For instance, let us consider the execution of a $\mathbb{F}_{p^2}$ multiplication followed by a subtraction shown in Figure 5.1. Note that multiplication is implemented using Karatsuba with 3 $\mathbb{F}_p$ multiplications and 5 $\mathbb{F}_p$ additions/subtractions.

As can be seen in Figure 5.1, the scheduling of the internal $\mathbb{F}_p$ operations of the $\mathbb{F}_{p^2}$ multiplication has been performed in such a way that contiguous data dependencies are minimal between $\mathbb{F}_p$ operations (there is only *one* dependence between `DblSub` and `Sub` in the last stage of multiplication). A similar analysis can be performed between contiguous higher-layer $\mathbb{F}_{p^2}$ operations. In Figure 5.1, the last $\mathbb{F}_p$ operation of the multiplication and the first $\mathbb{F}_p$ operation of the subtraction hold contiguous data dependence. There are different solutions to eliminate this problem. For example, it can be eliminated by rescheduling the $\mathbb{F}_{p^2}$ subtraction and addition, as shown in Figure 5.2(a). Note that addition does not hold any dependence with the multiplication or subtraction, as required. Alternatively, if internal $\mathbb{F}_p$ field operations of the subtraction are rescheduled, as shown in Figure 5.2(b), the contiguous data dependence is also eliminated. These strategies can be

FIGURE 5.1

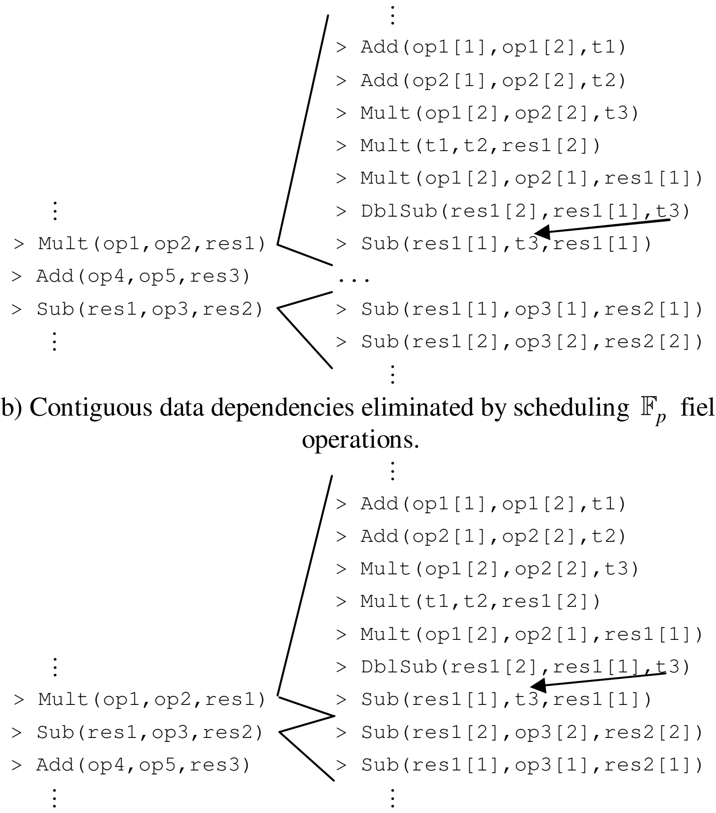
$\mathbb{F}_{p^2}$ operations with contiguous data dependencies. High-level $\mathbb{F}_{p^2}$ operations are in the left column and their corresponding low-level $\mathbb{F}_p$ operations are in the right column. $\mathbb{F}_{p^2}$ elements $(a+bi)$ are represented as (`op[1],op[2]`). Dependencies are indicated by arrows.



```
                                   ⋮
                              > Add(op1[1],op1[2],t1)
                              > Add(op2[1],op2[2],t2)
                              > Mult(op1[2],op2[2],t3)
                              > Mult(t1,t2,res1[2])
                              > Mult(op1[2],op2[1],res1[1])
    ⋮                         > DblSub(res1[2],res1[1],t3)
 > Mult(op1,op2,res1)         > Sub(res1[1],t3,res1[1])
 > Sub(res1,op3,res2)         > Sub(res1[1],op3[1],res2[1])
 > Add(op4,op5,res3)          > Sub(res1[2],op3[2],res2[2])
    ⋮                              ⋮
```

FIGURE 5.2

(a) Contiguous data dependencies eliminated by scheduling $\mathbb{F}_{p^2}$ field operations.



```
                                   ⋮
                              > Add(op1[1],op1[2],t1)
                              > Add(op2[1],op2[2],t2)
                              > Mult(op1[2],op2[2],t3)
                              > Mult(t1,t2,res1[2])
                              > Mult(op1[2],op2[1],res1[1])
    ⋮                         > DblSub(res1[2],res1[1],t3)
 > Mult(op1,op2,res1)         > Sub(res1[1],t3,res1[1])
 > Add(op4,op5,res3)          ...
 > Sub(res1,op3,res2)         > Sub(res1[1],op3[1],res2[1])
    ⋮                         > Sub(res1[2],op3[2],res2[2])
                                   ⋮
```

(b) Contiguous data dependencies eliminated by scheduling $\mathbb{F}_p$ field operations.

```
                                   ⋮
                              > Add(op1[1],op1[2],t1)
                              > Add(op2[1],op2[2],t2)
                              > Mult(op1[2],op2[2],t3)
                              > Mult(t1,t2,res1[2])
                              > Mult(op1[2],op2[1],res1[1])
    ⋮                         > DblSub(res1[2],res1[1],t3)
 > Mult(op1,op2,res1)         > Sub(res1[1],t3,res1[1])
 > Sub(res1,op3,res2)         > Sub(res1[2],op3[2],res2[2])
 > Add(op4,op5,res3)          > Sub(res1[1],op3[1],res2[1])
    ⋮                              ⋮
```

applied to point formulas to minimize the appearance of such dependencies.

The reader is referred to Appendix B for details about the scheduling of $\mathbb{F}_{p^2}$ operations suggested for point formulas using Jacobian and (extended) Twisted Edwards coordinates.

5.2 Merging of point and field operations

In the case of the GLS method, merging of point doublings is not as advantageous as in the traditional scenario of ECC over $\mathbb{F}_p$ because most contiguous data dependencies can be eliminated by simply rescheduling field operations inside point formulas using the techniques from Section 5.1 (see Appendix B). Moreover, GLS employs point multiplication techniques such as interleaving (see Section 6.2), which does not guarantee a long series of consecutive doublings between additions. Nevertheless, it is still advantageous the use of the merged doubling-addition operation, which is a recurrent operation in interleaving.

On the other hand, merging field operations is more advantageous in this scenario than over $\mathbb{F}_p$. There two reasons for this to happen. First, arithmetic over $\mathbb{F}_{p^2}$ works on top of the arithmetic over $\mathbb{F}_p$, which opens new possibilities to merge more $\mathbb{F}_p$ operations. Second, operations are on fields of half size, which means that fewer registers are required for representing field elements and more registers are available for holding intermediate operands.

For implementations using Jacobian and (extended) Twisted Edwards coordinates we suggest the following merged field operations on x86-64 based processors: $a-2b(\bmod p)$, $(a+a+a)/2\ (\bmod p)$, $a+b-c(\bmod p)$, the merging of $a+b(\bmod p)$ and $a-b(\bmod p)$, the merging of $a-b(\bmod p)$ and $c-d(\bmod p)$, and the merging of $a+a(\bmod p)$ and $a+a+a(\bmod p)$. Again, we remark that this list is not intended to be exhaustive and different merging options could be more advantageous or be available on different platforms with different coordinate systems or underlying fields. Please, refer to Appendix B for details about the merged $\mathbb{F}_{p^2}$ operations suggested for the GLS method with $\mathcal{J}$ and $\mathcal{E}/\mathcal{E}^e$ coordinates.

6 Performance Evaluation

In this section, we combine and demonstrate the efficiency of the techniques described in Sections 3-5 to accelerate the computation of a full point multiplication using $\mathcal{J}$ and $\mathcal{E}/\mathcal{E}^e$ coord. For our implementations, we use the well-known MIRACL library by M. Scott [42], which contains an extensive set of cryptographic functions that simplified the development/ optimization process of our crypto routines. Comparisons focus on implementations of variable-scalar-variable-point elliptic curve point multiplication with approximately 128 bits of security.

6.1 Details of the "Traditional" Implementations

Field Arithmetic:

As previously described, the field arithmetic over $\mathbb{F}_p$ was written using x86-64 compatible assembly language and optimized by exploiting incomplete reduction and elimination of conditional branches for modular addition, subtraction and multiplication/division by constants (see Section 3). For the case of modular multiplication and squaring, there are *two* methods that are commonly preferred in the literature for implementation on GPPs: schoolbook (or operand scanning method) and Comba [9] (or product scanning method) (see Section 5.3 of [14] or Section 2.2.2 of [24]). Both methods require n^2 w-bit multiplications when multiplying two n-digit numbers. However, we choose to implement Comba's method since it requires approx. $3n^2$ w-bit additions, whereas schoolbook requires $4n^2$. Our code was aggressively optimized by careful scheduling instructions to exploit the instruction-level parallelism.

Point Arithmetic:

For our implementations, we chose Jacobian and extended Twisted Edwards coord. (see Section 2.2) and used the formulas for doubling, addition and doubling-addition optimized by the authors (see Sections 4.1-4.2 of [35]). Thus, we use the execution patterns based on doublings and doubling-additions proposed by [31] and [26] for Jacobian and Twisted Edwards coordinates, respectively. The costs in terms of multiplications and squarings can be found in Table 1. Note that we use general additions (or general doubling-additions) because inversion is relatively expensive and its inclusion during precomputation cancels any gain using addition with mixed coordinates during the evaluation stage.

This arithmetic layer was optimized through the use of the techniques described in Section 4, namely field arithmetic scheduling, merging of point operations and merging of field operations. Because the maximal performance was found with a window of size 5 for the scalar recoding using wNAF (see next subsection), we merged *four* consecutive doublings into a joint function and every addition with the precedent doubling into another function. Please refer to Appendix A for complete details about these functions exhibiting minimal number of field operations, different merged field operations and reduced number of contiguous data dependencies.

Point Multiplication and Precomputation:

For scalar recoding, we use width-w Non-Adjacent Form (wNAF), which offers minimal nonzero density among signed binary representations for a given window width (i.e., for certain number of precomputed points) [2]. In particular, we use Alg. 3.35 of [24] for conversion from integer to wNAF representation. Although left-to-right conversion algorithms exist [2], which save memory and allow on-the-fly computation of point multiplication, they are not advantageous on the targeted CPUs. In fact, our tests show that converting the scalar to wNAF and then executing the point multiplication achieves higher performance than interleaving conversion and point multiplication. That is because the latter approach "interrupts" the otherwise smooth flow of point multiplication by calling the conversion function at every iteration of the double-and-add algorithm. Our choice is also justified because there are no stringent constraints in terms of memory in the targeted platforms.

For precomputation on $\mathcal{J}$ coordinates, we chose a variant of the LM scheme [36] that does not require inversions (see Section 7.1 of [33]). This method achieves the lowest cost for precomputing points, given by $(5L+2)\mathrm{M}+(2L+4)\mathrm{S}$, where L represents the number of non-trivial points (note that we avoid here the S-M trading in the first doubling). On $\mathcal{E}/\mathcal{E}^e$ coordinates, we precompute points in the traditional way using the sequence $P+2P+2P+\ldots+2P$, adding $2P$ with general additions. Because precomputed points are left in projective form no inversion is required and the cost is given by $(8L+4)\mathrm{M}+2\mathrm{S}$. This involves computing $2P$ as $2\mathcal{A}\rightarrow\mathcal{E}^e$, which costs $5\mathrm{M}+2\mathrm{S}$ (*one* squaring is saved because $Z_P=1$; *one* extra multiplication is required to compute T coordinate of $2P$), *one* mixed addition to compute $P+2P$ as $\mathcal{A}+\mathcal{E}^e\rightarrow\mathcal{E}^e$ that costs 7M and $(L-1)$ general additions $\mathcal{E}^e+\mathcal{E}^e\rightarrow\mathcal{E}^e$ that cost 8M each. For both coordinate systems, we chose a window with size $w=5$ (i.e., precomputing $\{P,[3]P,\ldots,[15]P\}$, $L=7$), which is optimal and slightly better than fractional windows using $L=6$ or 8.

6.2 Details of the GLS-based Implementations

As mentioned previously, for this case we make use of the optimized assembly module of the field arithmetic over $\mathbb{F}_{p^2}$ written by M. Scott [42], which exploits the Mersenne prime $p=2^{127}-1$ allowing the use of a very simple reduction step with no conditional branches.

For the point arithmetic, we slightly modify formulas for the "traditional" implementations since in this case these require a few extra multiplications with the twisted curve parameter μ (see Section 2.2). For example, the (dedicated) addition using extended Twisted Edwards coordinates with cost 8M (pp. 332 of [26]) cannot be used in this case and has to be replaced by a formula that costs 9M (also discussed in pp. 332 of [26] as "9M+1D"), which is *one* multiplication more expensive ("1D" is avoided because parameter a is still set to -1). Accordingly (and also following our discussions in Sections 4.1 and 5.1), the scheduling of the field arithmetic slightly differs. Moreover, different merging options for the field and point arithmetic are exploited (see Section 5.2). The reader is referred to Appendix B for complete details about the revised formulas exhibiting minimal number of field operations, different merged operations and reduced number of contiguous data dependencies.

For the point multiplication, each of the two scalars k_0 and k_1 in the multiple point multiplication $[k_0]P+[k_1](\lambda P)$ is converted using fractional wNAF [39], and then the evaluation stage is executed using interleaving (see Alg. 3.51 of [24]). Similarly to our experiments with the "traditional" implementations, we remark that the separation of the conversion and evaluation stages yields better performance in our case.

For precomputation on $\mathcal{J}$, we use the LM scheme (see Section 4 of [36]) that has minimal cost among methods using only *one* inversion, i.e., $1\mathrm{I}+(9L+1)\mathrm{M}+(2L+5)\mathrm{S}$, where L represents the number of non-trivial points (we avoid here the S-M trading in the first doubling). A fractional window with $L=6$ achieves the optimal performance in our case.

Again, on $\mathcal{E}/\mathcal{E}^e$ coordinates we precompute points using general additions in the sequence $P+2P+\ldots+2P$. Precomputed points are better left in projective coordinates, in which case the cost is given by $(9L+4)\text{M}+2\text{S}$. This cost involves the computation of $2P$ as $2\mathcal{A}\rightarrow\mathcal{E}^e$, which costs $5\text{M}+2\text{S}$ (*one* squaring is saved because $Z_P=1$; *one* extra multiplication is required to compute T coordinate of $2P$), *one* mixed addition to compute $P+2P$ as $\mathcal{A}+\mathcal{E}^e\rightarrow\mathcal{E}^e$ that costs 8M and $(L-1)$ general additions $\mathcal{E}^e+\mathcal{E}^e\rightarrow\mathcal{E}^e$ that cost 9M each. In this case, an integral window of size $w=5$ (i.e., $L=7$) achieves optimal performance. As pointed out by [19], precomputing $\{P,[3]\psi(P),[5]\psi(P),\ldots,[2L+1]\psi(P)\}$ can be done on-the-fly at low cost.

6.3 Timings

Here we summarize the timings obtained by our "traditional" implementations using $\mathcal{E}/\mathcal{E}^e$ and $\mathcal{J}$ coordinates (called *ted256189* and *jac256189*, respect.) and our implementations using GLS with $\mathcal{E}/\mathcal{E}^e$ and $\mathcal{J}$ coordinates (called *ted1271gls* and *jac1271gls*, respect.), when running them on a single core of Intel and AMD processors based on the x86-64 ISA. The curves used for these implementations are described in detail in Appendix C. For verification of each implementation, the results of 10^4 point multiplications with "random" scalars were all validated using MIRACL. Several "random" point multiplications were also verified with Magma.

All the tested programs were compiled with GCC v4.4.1 on the Intel Core 2 Duo E6750 and Intel Atom N450 and with GCC v4.3.4 on the Intel Xeon E5440 and AMD Opteron 252 processors. For measuring computing time, we follow [23] and use a method based on cycle counts. To obtain our timings, we ran each implementation 10^5 times with randomly generated scalars, averaged and approximated the results to the nearest 1000 cycles. Table 8 summarizes our results, labeled as *ted1271gls*, *jac1271gls*, *ted256189* and *jac256189*. All costs include scalar conversion, the point multiplication computation (precomputation and evaluation stages) and the final normalization step to affine. For comparison purposes, Table 8 also includes the cycle counts that we obtained when running the implementations by M. Scott (displayed as *gls1271-ref4* and *gls1271-ref3* [42]) on exactly the same platforms. Finally, the last 5 rows of the table detail cycle counts of several state-of-the-art implementations as reported in the literature. However, these referenced results are used only to provide an approximate comparison since the processor platforms are not identical (though they use very similar processors).

As can be seen in Table 8, our fastest implementation on the targeted platforms is *ted1271gls*, using $\mathcal{E}/\mathcal{E}^e$ with the GLS method. This implementation is about 22% faster than the previous record set by *gls1271-ref4* [18] on a slightly different processor (1.66GHz Intel Core 2 Duo). A more precise comparison, however, would be between measurements on identical processor platforms. In this case, *ted1271gls* is approx. 20%, 22%, 22% and 28% faster than *gls1271-ref4* [42] on Atom N450, Core 2 Duo E6750, Xeon E5440 and Opteron 252, respectively. Although [42] uses inverted Twisted Edwards coordinates ($\mathcal{E}^{inv}$), the improvement with the change of coordinates only explains a small fraction of the speed-up. Similarly, in the case of $\mathcal{J}$ combined with GLS, *jac1271gls* is about 23% faster than the record set by *gls1271-ref3* [19] on a 1.66GHZ Intel Core 2 Duo. When comparing cycle

TABLE 8
Cost (in cycles) of point multiplication.

Implementation	Coordinates	Field arithmetic	Atom N450	Core 2 Duo E6750	Xeon E5440	Opteron 252
ted1271gls	$\mathcal{E} / \mathcal{E}^e$	$\mathbb{F}_{p^2}$, 127-bit	**588000**	**229000**	**230000**	**211000**
jac1271gls	$\mathcal{J}$	$\mathbb{F}_{p^2}$, 127-bit	644000	252000	255000	238000
ted256189	$\mathcal{E} / \mathcal{E}^e$	$\mathbb{F}_p$, 256-bit	982000	281000	289000	232000
jac256189	$\mathcal{J}$	$\mathbb{F}_p$, 256-bit	1168000	337000	343000	274000
gls1271-ref4 [42]	$\mathcal{E}^{inv}$	$\mathbb{F}_{p^2}$, 127-bit	732000	295000	296000	295000
gls1271-ref3 [42]	$\mathcal{J}$	$\mathbb{F}_{p^2}$, 127-bit	832000	332000	332000	341000
gls1271-ref4 [18]	$\mathcal{E}^{inv}$	$\mathbb{F}_{p^2}$, 127-bit	-	293000 [1]	-	-
gls1271-ref3 [19]	$\mathcal{J}$	$\mathbb{F}_{p^2}$, 127-bit	-	326000 [1]	-	-
curve25519 [23]	Montgomery	$\mathbb{F}_p$, 255-bit	-	386000 [2]	-	307000 [4]
Hisil et al. [27]	$\mathcal{E} / \mathcal{E}^e$	$\mathbb{F}_p$, 256-bit	-	362000 [3]	-	-
Hisil et al. [27]	$\mathcal{J}$	$\mathbb{F}_p$, 256-bit	-	468000 [3]	-	-

(1) On a 1.66GHZ Intel Core 2 Duo. (2) On a 2.66GHZ Intel Core 2 Duo E6700. (3) On a 2.66GHZ Intel Core 2 Duo E6550. (4) On a 2.4GHZ AMD Celeron 250.

counts on identical processor platforms, *jac1271gls* is 23%, 24%, 23% and 30% faster than *gls1271-ref3* [42] on Atom N450, Core 2 Duo E6750, Xeon E5440 and Opteron 252, respect. Our implementations are also significantly faster than the implementation of Bernstein's *curve25519* by Gaudry and Thomé [23]. For instance, *ted1271gls* is 41% faster than *curve25519* [23] on a 2.66GHz Intel Core 2 Duo.

If the GLS method is not considered, the fastest implementations using $\mathcal{E}/\mathcal{E}^e$ and $\mathcal{J}$ coordinates are *ted256189* and *jac256189*, respectively. In this case, *ted256189* and *jac256189* are 22% and 28% faster than the previous best cycle counts due to Hisil et al. [27] using also $\mathcal{E}/\mathcal{E}^e$ and $\mathcal{J}$ coordinates, respectively, on a 2.66GHz Intel Core 2 Duo.

It is also interesting to note that the performance boost given by the GLS method strongly depends on the characteristics of a given platform. For instance, *ted1271gls* and *jac1271gls* are about 40% and 45% faster than their "counterparts" over $\mathbb{F}_p$, namely *ted256189* and *jac256189*, respectively, on an Intel Atom N450. On an Intel Core 2 Duo E6750, the differences reduce to 19% and 25% (respect.). And on an AMD Opteron processor, the differences reduce even further to only 9% and 13% (respect.). Thus, it seems to exist a correlation between an architecture's "aggressiveness" for scheduling operations/exploiting ILP and the gap between the costs of $\mathbb{F}_p$ and $\mathbb{F}_{p^2}$ operations on x86-64 based processors. In general, the greater such "aggressiveness" the smaller the $\mathbb{F}_p - \mathbb{F}_{p^2}$ gap. And since working on the quadratic extension involves a considerable increase in the number of multiplications and additions, GLS loses its attractiveness if such gap is not large enough on certain platform. For the record, *ted1271gls* achieves the best cycle count on an AMD Opteron processor with an advantage of about 31% over the best previous result in the literature due to Gaudry and Thomé (i.e., *curve25519* [23]).

7 Conclusions

In this paper, we have combined efficiently techniques such as incomplete reduction and the elimination of conditional branches to implement highly efficient field arithmetic over prime fields. Moreover, we have studied the impact of data dependencies between field operations and proposed *three* techniques that reduce significantly the appearance of pipeline stalls on x86-64 based processors. Our methods also reduce the number of function calls and memory reads/writes and can be easily extended to different underlying fields. We have finally shown that, by combining efficiently all these techniques with state-of-the-art algorithms and formulas for ECC point multiplication, significant gains in performance are achieved. Our high-speed implementations are up to 31% faster than the best previous results in the literature. Although our implementations (in their current form) only compute $[k]P$ where k and P vary, several of the optimizations discussed in this work are generic and can be easily adapted to speed up other implementations using a fixed point P, digital signatures and different coordinate systems/curve forms/underlying fields.

Acknowledgments. This work was made possible by the facilities of the Shared Hierarchical Academic Research Computing Network (SHARCNET) and Compute/Calcul Canada. We would like to thank the Natural Sciences and Engineering Research Council of Canada (NSERC) and the Ontario Centres of Excellence (OCE) for partially supporting this work.

References

[1] Advanced Micro Devices, "AMD64 Architecture Programmer's Manual, Volume 1: Application Programming," 2009. Available at `http://developer.amd.com/DOCUMENTATION/GUIDES/Pages/default.aspx`

[2] R. Avanzi, "A Note on the Signed Sliding Window Integer Recoding and its Left-to-Right Analogue," in *Workshop on Selected Areas in Cryptography (SAC 2004)*, LNCS Vol. 3357, pp. 130–143, Springer, Heidelberg, 2005.

[3] D. Bernstein, P. Birkner, M. Joye, T. Lange and C. Peters, "Twisted Edwards Curves," in *Advances of Cryptology - Africacrypt 2008*, LNCS Vol. 5023, pp. 389–405, Springer, Heidelberg, 2008.

[4] D. Bernstein, "Curve25519: New Diffie-Hellman Speed Records," in *Public Key Cryptography (PKC'06)*, LNCS Vol. 3958, pp. 229-240, Springer, Heidelberg, 2006.

[5] M. Brown, D. Hankerson, J. Lopez and A. Menezes, "Software Implementation of the NIST Elliptic Curves over Prime Fields," in *Progress in Cryptology CT-RSA 2001*, LNCS Vol. 2020, pp. 250-265, Springer, Heidelberg, 2001.

[6] O. Billet and M. Joye, "The Jacobi Model of an Elliptic Curve and Side-Channel Analysis," in *AAECC 2003*, LNCS Vol. 2643, pp. 34–42, Springer, Heidelberg, 2003.

[7] D. Chudnovsky and G. Chudnovsky, "Sequences of Numbers Generated by Addition in Formal Groups and New Primality and Factorization Tests," in *Advances in Applied Mathematics*, Vol. 7, No 4, pp. 385-434, 1986.

[8] H. Cohen, A. Miyaji and T. Ono, "Efficient Elliptic Curve Exponentiation using Mixed Coordinates," in *Advances in Cryptology – Asiacrypt'98*, LNCS Vol. 1514, pp. 51–65, Springer, Heidelberg, 1998.

[9] P. G. Comba, "Exponentiation Cryptosystems on the IBM PC," in *IBM Systems Journal*, Vol. 29(4), pp. 526–538, 1990.

[10] C. Neil and P. Schwabe, "Fast Elliptic-Curve Cryptography on the Cell Broadband Engine," in *Progress in Cryptology – Africacrypt 2009*, LNCS Vol. 5580, pp. 368-385, Springer, Heidelberg, 2009.

[11] V. Dimitrov, L. Imbert and P.K. Mishra, "Efficient and Secure Elliptic Curve Point Multiplication using Double-Base Chains," in *Advances in Cryptology – Asiacrypt 2005*, LNCS Vol. 3788, pp. 59–78, Springer, Heidelberg, 2005.

[12] H. Edwards, "A Normal Form for Elliptic Curves," in *Bulletin of the American Mathematical Society*, Vol. 44, pp. 393–422, 2007.

[13] D. Bernstein and T. Lange, "Explicit-Formula Database (EFD)," building on work by many authors, 2007. Available at http://www.hyperelliptic.org/EFD/oldefd/

[14] S.S. Erdem, T. Yanik and Ç.K. Koç, "Fast Finite Field Multiplication," in Ç.K. Koç (ed.) Cryptographic Engineering, Chapter 5, Springer, 2009.

[15] A. Fog, "Instruction Tables: Lists of Instruction Latencies, Throughputs and Micro-operation Breakdowns for Intel, AMD and VIA CPUs," 2009. Available at http://www.agner.org/optimize/#manuals, accessed on Jan. 2010.

[16] A. Fog, "The Microarchitecture of Intel, AMD and VIA CPUs," 2009. Available at http://www.agner.org/optimize/#manuals, accessed on January 2010.

[17] J. Großschädl, R. Avanzi, E. Savas and S. Tillich, "Energy-Efficient Software Implementation of Long Integer Modular Arithmetic," in *Workshop on Cryptographic Hardware and Embedded Systems (CHES'05)*, LNCS, Vol. 3659, pp. 75-90, Springer, Heidelberg, 2005.

[18] S. Galbraith, X. Lin and M. Scott, "Endomorphisms for Faster Elliptic Curve Cryptography on a Large Class of Curves," in *Cryptology ePrint Archive*, Report 2008/194, 2008.

[19] S. Galbraith, X. Lin and M. Scott, "Endomorphisms for Faster Elliptic Curve Cryptography on a Large Class of Curves," in *Advances of Cryptology - Eurocrypt 2009*, LNCS Vol. 5479, pp. 518–535, Springer, Heidelberg, 2009.

[20] R. Gallant, R. Lambert and S. Vanstone, "Faster Point Multiplication on Elliptic Curves with Efficient Endomorphisms," in *Advances of Cryptology - CRYPTO 2001*, LNCS Vol. 2139, pp. 190–200, Springer, Heidelberg, 2001.

[21] N. Gura, A. Patel, A. Wander, H. Eberle and S.C. Shantz, "Comparing Elliptic Curve Cryptography and RSA on 8-bit CPUs," in *Workshop on Cryptographic Hardware and Embedded Systems (CHES'04)*, LNCS, Vol. 3156, pp. 119-132, Springer, Heidelberg, 2004.

[22] P. Gaudry and E. Thomé, "mpFq – A Finite Field Library," 2007–2009. Available at http://mpfq.gforge.inria.fr/mpfq-1.0-rc2.tar.gz

[23] P. Gaudry and E. Thomé, "The mpFq Library and Implementing Curve-Based Key Exchanges," in *SPEED 2007*, pp. 49–64, 2007.

[24] D. Hankerson, A. Menezes and S. Vanstone, "Guide to Elliptic Curve Cryptography," Springer-Verlag, 2004.

[25] H. Hisil, K. Wong, G. Carter, E. Dawson, "Faster Group Operations on Elliptic Curves," in *Cryptology ePrint Archive*, Report 2007/441, 2007.

[26] H. Hisil, K. Wong, G. Carter and E. Dawson, "Twisted Edwards Curves Revisited," in *Advances of Cryptology - Asiacrypt 2008*, LNCS Vol. 5350, pp. 326–343, Springer, Heidelberg, 2008.

[27] H. Hisil, K. Wong, G. Carter and E. Dawson, "Jacobi Quartic Curves Revisited," *Cryptology ePrint Archive*, Report 2009/312, 2009.

[28] J. Hennessy and D. Patterson, "Computer Architecture: A Quantitative Approach," Morgan Kaufman, 2006.

[29] A. A. Karatsuba and Y. P. Ofman, "Multiplication of Multidigit Numbers on Automata," in *Doklady Akademii Nauk SSSR*, Vol. 145(2), pp. 293–294, 1962.

[30] N. Koblitz, "Elliptic Curve Cryptosystems," in *Mathematics of Computation*, Vol. 48, pp. 203–209, 1987.

[31] P. Longa, "Accelerating the Scalar Multiplication on Elliptic Curve Cryptosystems over Prime Fields," *Master's Thesis*, University of Ottawa, 2007. Available at http://patricklonga.bravehost.com/publications.html#thesis

[32] P. Longa, "ECC Point Arithmetic Formulae (EPAF)," 2008. Available at http://patricklonga.bravehost.com/jacobian.html

[33] P. Longa and C. Gebotys, "Setting Speed Records with the (Fractional) Multibase Non-Adjacent Form Method for Efficient Elliptic Curve Scalar Multiplication," *CACR technical report*, CACR 2008-06, 2008.

[34] P. Longa and C. Gebotys, "Novel Precomputation Schemes for Elliptic Curve Cryptosystems," in *International Conference on Applied Cryptography and Network Security (ACNS 2009)*, LNCS Vol. 5536, pp. 71-88, Springer, Heidelberg, 2009.

[35] P. Longa and C. Gebotys, “Efficient Techniques for High-Speed Elliptic Curve Cryptography,” in *Workshop on Cryptographic Hardware and Embedded Systems (CHES 2010)*, 2010 (to appear). Available at `http://eprint.iacr.org/2010/315`

[36] P. Longa and A. Miri, “New Composite Operations and Precomputation Scheme for Elliptic Curve Cryptosystems over Prime Fields,” in *International Conference on Practice and Theory in Public Key Cryptography (PKC 2008)*, LNCS Vol. 4939, pp. 229–247, Springer, Heidelberg, 2008.

[37] P. Longa and A. Miri, “Fast and Flexible Elliptic Curve Point Arithmetic over Prime Fields,” in *IEEE Transactions on Computers*, Vol. 57, No 3, pp. 289–302, 2008.

[38] V. Miller, “Use of Elliptic Curves in Cryptography,” in *Advances in Cryptology - CRYPTO’85*, LNCS Vol. 218, pp. 417–426, Springer, 1986.

[39] B. Möller, “Improved Techniques for Fast Exponentiation,” in *International Conference of Information Security and Cryptology (ICISC 2002)*, pp. 298-312, 2002.

[40] P. L. Montgomery, “Modular Multiplication without Trial Division,” in *Mathematics of Computation*, Vol. 44(170), pp. 519–521, 1985.

[41] G.W. Reitweisner, “Binary Arithmetic,” in *Adv. Comput.*, Vol. 1, pp. 232–308, 1960.

[42] M. Scott, “MIRACL – Multiprecision Integer and Rational Arithmetic C/C++ Library,” 1988–2007. Available at `ftp://ftp.computing.dcu.ie/pub/crypto/miracl.zip`

[43] R. Szerwinski and T. Güneysu, “Exploiting the Power of GPUs for Asymmetric Cryptography,” in *Workshop on Cryptographic Hardware and Embedded Systems (CHES’08)*, LNCS Vol. 5154, pp. 79–99, 2008.

[44] J.A. Solinas, “Efficient Arithmetic on Koblitz Curves,” in *Design, Codes and Cryptography*, Vol. 19, pp. 195–249, 2000.

[45] O. Ugus, D. Westhoff, R. Laue, A. Shoufan and S.A. Huss, “Optimized Implementation of Elliptic Curve based Additive Homomorphic Encryption for Wireless Sensor Networks,” in *Workshop on Embedded Systems Security (WESS 2007)*, pp. 11–16, 2007.

[46] S.B. Xu and L. Batina, “Efficient Implementation of Elliptic Curve Cryptosystems on an ARM7 with Hardware Accelerator,” in *International Conference on Information and Communications Security (ICICS’01)*, LNCS Vol. 2200, pp. 266-279, Springer, Heidelberg, 2001.

[47] T. Yanik, E. Savaş and C.K. Koç, “Incomplete Reduction in Modular Arithmetic,” in *IEE Proceedings of Computers and Digital Techniques,* Vol. 149(2), pp. 46–52, 2002.

A Point Operations Using $\mathcal{J}$ and $\mathcal{E}/\mathcal{E}^e$ Coordinates

The following Maple scripts verify formulas used for the “traditional” implementations discussed in this work. Note that point and field operations have been carefully merged and scheduled to reduce the number of function calls, memory reads/writes and potential pipeline stalls. Temporary registers are denoted by t_i and M=multiplication, S=squaring, Add=addition, Sub=subtraction, Mul*x*=multiplication by *x*, Div*x* = division by *x*, Neg=negation. DblSub represents the computation $a-2b \pmod p$ and SubDblSub represents the merging of $a-b \pmod p$ and $(a-b)-2c \pmod p$. Underlined field operations are merged and *operation*$_{\text{IR}}$ represents a field operation using incomplete reduction. In practice, input registers are reused to store the result of an operation.

```
# Weierstrass curve (for verification):
x1:=X1/Z1^2; y1:=Y1/Z1^3; x2:=X2/Z2^2; y2:=Y2/Z2^3; ZZ2:=Z2^2; ZZZ2:=Z2^3; a:=-3;
x3:=((3*x1^2+a)/(2*y1))^2-2*x1; y3:=((3*x1^2+a)/(2*y1))*(x1-x3)-y1;
x4:=((y1-y2)/(x1-x2))^2-x2-x1; y4:=((y1-y2)/(x1-x2))*(x2-x4)-y2;
x5:=((y1-y4)/(x1-x4))^2-x4-x1; y5:=((y1-y4)/(x1-x4))*(x4-x5)-y4;
```

DBL, $2\mathcal{J} \rightarrow \mathcal{J}$: $2(X_1,Y_1,Z_1) \rightarrow (X_{out},Y_{out},Z_{out})$. Cost = 4M+4S+3Sub+1DblSub+1Add$_{\text{IR}}$+

1Mul3$_{IR}$+1Div2$_{IR}$; 5 contiguous data dependencies

```
# In practice, Xout,Yout,Zout reuse the registers X1,Y1,Z1 for all cases below.
t4:=Z1^2; t3:=Y1^2; t1:=X1+t4; t4:=X1-t4; t0:=3*t4; t5:=X1*t3; t4:=t1*t0; t0:=t3^2;
t1:=t4/2; t3:=t1^2; Zout:=Y1*Z1; Xout:=t3-2*t5; t3:=t5-Xout; t5:=t1*t3; Yout:=t5-t0;
simplify([x3-Xout/Zout^2]), simplify([y3-Yout/Zout^3]); # Check
```

4DBL, $8\mathcal{J} \rightarrow \mathcal{J}$: $8(X_1,Y_1,Z_1) \rightarrow (X_{out},Y_{out},Z_{out})$. Cost = 4*(4M+4S+3Sub+1DblSub+1Add$_{IR}$+1Mul3$_{IR}$+1Div2$_{IR}$); 1.25 contiguous data dependencies/doubling

```
t4:=Z1^2; t3:=Y1^2; t1:=X1+t4; t4:=X1-t4; t2:=3*t4; t5:=X1*t3; t4:=t1*t2; t0:=t3^2;
t1:=t4/2; Zout:=Y1*Z1; t3:=t1^2; t4:=Z1^2; Xout:=t3-2*t5; t3:=t5-Xout; t2:=Xout+t4;
t5:=t1*t3; t4:=Xout-t4; Yout:=t5-t0; t1:=3*t4; t3:=Yout^2; t4:=t1*t2; t5:=Xout*t3;
t1:=t4/2; t0:=t3^2; t3:=t1^2; Zout:=Yout*Zout; Xout:=t3-2*t5; t4:=Zout^2; t3:=t5-
Xout; t2:=Xout+t4; t5:=t1*t3; t4:=Xout-t4; Yout:=t5-t0; t1:=3*t4; t3:=Yout^2;
t4:=t1*t2; t5:=Xout*t3; t1:=t4/2; t0:=t3^2; t3:=t1^2; Zout:=Yout*Zout; Xout:=t3-2*t5;
t4:=Zout^2; t3:=t5-Xout; t2:=Xout+t4; t5:=t1*t3; t4:=Xout-t4; Yout:=t5-t0; t1:=3*t4;
t3:=Yout^2; t4:=t1*t2; t5:=Xout*t3; t1:=t4/2; t0:=t3^2; t3:=t1^2; Zout:=Yout*Zout;
Xout:=t3-2*t5; t3:=t5-Xout; t5:=t1*t3; Yout:=t5-t0;
```

mDBLADD, $2\mathcal{J} + \mathcal{A} \rightarrow \mathcal{J}$: $2(X_1,Y_1,Z_1) + (x_2,y_2) \rightarrow (X_{out},Y_{out},Z_{out})$. Cost = 13M+5S+7Sub+2DblSub+1Add$_{IR}$+1Mul2$_{IR}$; 5 contiguous data dependencies

```
t5:=Z1^2; t6:=Z1*t5; t4:=x2*t5; t5:=y2*t6; t1:=t4-X1; t2:=t5-Y1; t4:=t2^2; t6:=t1^2;
t5:=t6*X1; t0:=t1*t6; t3:=t4-2*t5; t4:=Z1*t1; t3:=t3-t5; t6:=t0*Y1; t3:=t3-t0;
t1:=2*t6; Zout:=t4*t3; t4:=t2*t3; t0:=t3^2; t1:=t1+t4; t4:=t0*t5; t7:=t1^2;
t5:=t0*t3; Xout:=t7-2*t4; Xout:=Xout-t5; t3:=Xout-t4; t0:=t5*t6; t4:=t1*t3; Yout:=t4-
t0;
simplify([x5-Xout/Zout^2]), simplify([y5-Yout/Zout^3]); # Check
```

DBLADD, $2\mathcal{J} + \mathcal{J} \rightarrow \mathcal{J}$: $2(X_1,Y_1,Z_1) + (X_2,Y_2,Z_2,Z_2^2,Z_2^3) \rightarrow (X_{out},Y_{out},Z_{out})$. Cost = 16M+5S+7Sub+2DblSub+1Add$_{IR}$+1Mul2$_{IR}$; 3 contiguous data dependencies

```
t0:=X1*ZZ2; t5:=Z1^2; t7:=Y1*ZZZ2; t4:=X2*t5; t6:=t5*Z1; t1:=t4-t0; t5:=Y2*t6;
t6:=t1^2; t2:=t5-t7; t4:=t2^2; t5:=t6*t0; t0:=t1*t6; t3:=t4-2*t5; t6:=Z1*t1; t3:=t3-
t5; t4:=Z2*t6; t3:=t3-t0; t6:=t7*t0; Zout:=t4*t3; t4:=t2*t3; t1:=2*t6; t0:=t3^2;
t1:=t1+t4; t4:=t0*t5; t7:=t1^2; t5:=t0*t3; Xout:=t7-2*t4; Xout:=Xout-t5; t3:=Xout-t4;
t0:=t5*t6; t4:=t1*t3; Yout:=t4-t0;
simplify([x5-Xout/Zout^2]), simplify([y5-Yout/Zout^3]); # Check
```

```
# Twisted Edwards curve (for verification):
x1:=X1/Z1; y1:=Y1/Z1; x2:=X2/Z2; y2:=Y2/Z2; T2:=X2*Y2/Z2; a:=-1;
x3:=(2*x1*y1)/(y1^2+a*x1^2); y3:=(y1^2-a*x1^2)/(2-y1^2-a*x1^2);
x4:=(x3*y3+x2*y2)/(y3*y2+a*x3*x2); y4:=(x3*y3-x2*y2)/(x3*y2-y3*x2);
```

DBL, $2\mathcal{E} \rightarrow \mathcal{E}$: $2(X_1,Y_1,Z_1) \rightarrow (X_{out},Y_{out},Z_{out})$. Cost = 4M+3S+1SubDblSub+1Add$_{IR}$+

1Mul2$_{IR}$+1Neg; no contiguous data dependencies

```
t1:=2*X1;  t2:=X1^2;  t4:=Y1^2;  t3:=Z1^2;  Xout:=t2+t4;  t4:=t4-t2;  t3:=t4-2*t3;
t2:=t1*Y1; Yout:=-t4; Zout:=t4*t3; Yout:=Yout*Xout; Xout:=t3*t2;
simplify([x3-Xout/Zout]), simplify([y3-Yout/Zout]); # Check
# Iterate this code n times to implement nDBL with cost
n(4M+3S+1SubDblSub+1AddIR+1Mul2IR+1Neg)
```

Merged DBL–ADD, $(2\mathcal{E})^e + \mathcal{E}^e \rightarrow \mathcal{E}$: $2(X_1,Y_1,Z_1)+((X_2+Y_2),(X_2-Y_2),2Z_2,2T_2)\rightarrow (X_{out},Y_{out},Z_{out})$. Cost = 12M+3S+3Sub+1SubDblSub+4Add$_{IR}$+1Mul2$_{IR}$; no contiguous data dependencies

```
# If Z2=1 (Merged DBL-mADD), t5:=(2*Z2)*t6 is replaced by t5:=2*t6 and the number of
multiplies reduces to 11M at the expense of one extra Mul2
t1:=2*X1;  t5:=X1^2;  t7:=Y1^2;  t6:=Z1^2;  Xout:=t5+t7;  t7:=t7-t5;  t6:=t7-2*t6;
t5:=t1*Y1; t8:=t7*Xout; t0:=t7*t6; t7:=t6*t5; t6:=Xout*t5; Xout:=t7+t8; t1:=t7-t8;
t7:=(2*T2)*t0; t5:=(2*Z2)*t6; t0:=(X2-Y2)*t1; t1:=t5+t7; t6:=(X2+Y2)*Xout; Xout:=t5-
t7; t7:=t0-t6; t0:=t0+t6; Xout:=Xout*t7; Yout:=t1*t0; Zout:=t0*t7;
simplify([x4-Xout/Zout]), simplify([y4-Yout/Zout]); # Check
```

B Point Operations Using $\mathcal{J}$ And $\mathcal{E}/\mathcal{E}^e$ For the GLS Method

The following Maple scripts verify formulas used for the GLS-based implementations discussed in this work. In the remainder, DblSub represents $a-2b \pmod p$ or $a-b-c \pmod p$, Mul3Div2 represents $(a+a+a)/2 \pmod p$, AddSub represents the merging of $a+b \pmod p$ and $a-b \pmod p$, AddSub2 represents $a+b-c \pmod p$, SubSub represents the merging of $a-b \pmod p$ and $c-d \pmod p$, and Mul2Mul3 represents the merging of $a+a \pmod p$ and $a+a+a \pmod p$.

```
# Weierstrass curve (for verification):
x1:=X1/Z1^2; y1:=Y1/Z1^3; a:=-3;
x3:=((3*x1^2+u^2*a)/(2*y1))^2-2*x1; y3:=((3*x1^2+u^2*a)/(2*y1))*(x1-x3)-y1;
x4:=((y1-y2)/(x1-x2))^2-x2-x1; y4:=((y1-y2)/(x1-x2))*(x2-x4)-y2;
x5:=((y1-y4)/(x1-x4))^2-x4-x1; y5:=((y1-y4)/(x1-x4))*(x4-x5)-y4;
```

DBL, $2\mathcal{J} \rightarrow \mathcal{J}$: $2(X_1,Y_1,Z_1)\rightarrow (X_{out},Y_{out},Z_{out})$. Cost = 4M+4S+2Sub+1DblSub+1Mul3Div2+1AddSub+1Mulμ; no contiguous data dependencies

```
# In practice, Xout,Yout,Zout reuse the registers X1,Y1,Z1 for all cases below.
t2:=Z1^2; t3:=Y1^2; t1:=u*t2; t2:=X1+t1; t1:=X1-t1; t1:=3*t1/2; t4:=t3*X1; t1:=t2*t1;
t3:=t3^2; Xout:=t1^2; Zout:=Y1*Z1; Xout:=Xout-2*t4; t2:=t4-Xout; t1:=t1*t2; Yout:=t1-
t3;
simplify([x3-Xout/Zout^2]), simplify([y3-Yout/Zout^3]); # Check
```

mADD, $\mathcal{J}+\mathcal{A}\rightarrow\mathcal{J}$: $(X_1,Y_1,Z_1)+(x_2,y_2)\rightarrow (X_{out},Y_{out},Z_{out})$. Cost = 8M+3S+5Sub+1DblSub; no contiguous data dependencies

```
t2:=Z1^2; t1:=Z1*t2; t2:=t2*x2; t1:=t1*y2; t2:=t2-X1; t1:=t1-Y1; t3:=t2^2; t4:=t1^2;
Zout:=Z1*t2; t2:=t2*t3; t3:=t3*X1; Xout:=t4-t2; Xout:=Xout-2*t3; t3:=t3-Xout;
t1:=t1*t3; Yout:=t2*Y1; Yout:=t1-Yout;
simplify([x4-Xout/Zout^2]), simplify([y4-Yout/Zout^3]); # Check
```

mDBLADD, $2\mathcal{J}+\mathcal{A}\rightarrow\mathcal{J}$: $2(X_1,Y_1,Z_1)+(x_2,y_2)\rightarrow(X_{out},Y_{out},Z_{out})$. Cost = 13M+5S+2Sub+2DblSub+1SubSub+1Add+1Mul2+1Mul2Mul3+1Div2; no contiguous data depend.

```
t2:=Z1^2; t1:=Z1*t2; t3:=x2*t2; t1:=y2*t1; t2:=t3-X1; t1:=t1-Y1; t3:=t2^2; t5:=t1^2;
t4:=X1*t3; t3:=t2*t3; Xout:=2*t4; t4:=3*t4; Zout:=Z1*t2; t5:=t5-t3-t4; Yout:=t3*Y1;
t1:=t1*t5; t2:=2*Yout; t3:=t5^2; t1:=t1+t2; t2:=Xout*t3; Xout:=t1^2; t3:=t5*t3;
Xout:=Xout-t2-t3; t2:=t2/2; Zout:=Zout*t5; Yout:=Yout*t3; t2:=Xout-t2; t1:=t1*t2;
Yout:=t1-Yout;
simplify([x5-Xout/Zout^2]), simplify([y5-Yout/Zout^3]); # Check
```

```
# Twisted Edwards curve (for verification):
x1:=X1/Z1; y1:=Y1/Z1; a:=-1;
x2:=X2/Z2; y2:=Y2/Z2; T2:=X2*Y2/Z2; x5:=X5/Z5; y5:=Y5/Z5; T5:=X5*Y5/Z5;
x3:=(2*x1*y1)/(y1^2+u*a*x1^2); y3:=(y1^2-u*a*x1^2)/(2-y1^2-u*a*x1^2);
x4:=(x3*y3+x2*y2)/(y3*y2+u*a*x3*x2); y4:=(x3*y3-x2*y2)/(x3*y2-y3*x2);
x6:=(x4*y4+x5*y5)/(y4*y5+u*a*x4*x5); y6:=(x4*y4-x5*y5)/(x4*y5-y4*x5);
```

DBL, $2\mathcal{E}\rightarrow\mathcal{E}$: $2(X_1,Y_1,Z_1)\rightarrow(X_{out},Y_{out},Z_{out})$. Cost = 4M+3S+1Sub+1AddSub+2Mul2+1Mulμ; no contiguous data dependencies

```
Zout:=Z1^2; t1:=2*X1; t2:=X1^2; t1:=t1*Y1; Xout:=u*t2; Yout:=Y1^2; Zout:=2*Zout;
t2:=Yout-Xout; Yout:=Yout+Xout; Zout:=Zout-t2; Yout:=t2*Yout; Xout:=t1*Zout;
Zout:=t2*Zout;
simplify([x3-Xout/Zout]), simplify([y3-Yout/Zout]); # Check
```

Merged DBL–ADD, $(2\mathcal{E})^e+\mathcal{E}^e\rightarrow\mathcal{E}$: $2(X_1,Y_1,Z_1)+(X_2,Y_2,Z_2,T_2)\rightarrow(X_{out},Y_{out},Z_{out})$. Cost = 13M+3S+3Sub+1Add+2AddSub+1AddSub2+2Mul2+2Mulμ; no contiguous data dependencies

```
# If Z2=1 (Merged DBL-mADD), T1:=T1*Z2 is not needed and the number of multiplies
reduces to 12M
Zout:=Z1^2; t1:=2*X1; t2:=X1^2; t1:=t1*Y1; Xout:=u*t2; Yout:=Y1^2; Zout:=2*Zout;
t2:=Yout-Xout; Yout:=Xout+Yout; Zout:=Zout-t2; T1:=t1*Yout; Yout:=t2*Yout;
Xout:=t1*Zout; Zout:=t2*Zout; t1:=Xout*X2; T1:=T1*Z2; Zout:=Zout*T2; t2:=u*t1;
t3:=T1+Zout; Zout:=T1-Zout; T1:=Yout*Y2; Xout:=Xout-Yout; Yout:=X2+Y2; t2:=T1-t2;
Xout:=Xout*Yout; Yout:=Zout*t2; t1:=Xout+T1-t1; Zout:=t1*t2; Xout:=t1*t3;
simplify([x4-Xout/Zout]), simplify([y4-Yout/Zout]); # Check
```

Merged DBL–ADDADD, $(2\mathcal{E})^e+\mathcal{E}^e+\mathcal{E}^e\rightarrow\mathcal{E}$: $2(X_1,Y_1,Z_1)+(X_2,Y_2,Z_2,T_2)+(X_3,Y_3,Z_3,T_3)\rightarrow(X_{out},Y_{out},Z_{out})$. Cost = 22M+3S+5Sub+2Add+3AddSub+2AddSub2+2Mul2+3Mulμ; no contiguous data dependencies

```
# If Z2=1, T1:=T1*Z2 is not needed and the number of multiplies reduces in 1M
# If Z5=1, T1:=T1*Z5 is not needed and the number of multiplies reduces in 1M
Zout:=Z1^2; t1:=2*X1; t2:=X1^2; t1:=t1*Y1; Xout:=u*t2; Yout:=Y1^2; Zout:=2*Zout;
```

```
t2:=Yout-Xout;   Yout:=Xout+Yout;   Zout:=Zout-t2;   T1:=t1*Yout;   Yout:=t2*Yout;
Xout:=t1*Zout;  Zout:=t2*Zout;  t1:=Xout*X2;  T1:=T1*Z2;  Zout:=Zout*T2;  t2:=u*t1;
t3:=T1+Zout;  Zout:=T1-Zout; T1:=Yout*Y2; Xout:=Xout-Yout; Yout:=X2+Y2; t2:=T1-t2;
Xout:=Xout*Yout;   Yout:=Zout*t2;   Xout:=Xout+T1-t1;   T1:=Zout*t3;   Zout:=Xout*t2;
Xout:=Xout*t3;   t1:=Xout*X5;   T1:=T1*Z5;   Zout:=Zout*T5;   t2:=u*t1;   t3:=T1+Zout;
Zout:=T1-Zout; T1:=Yout*Y5; Xout:=Xout-Yout; Yout:=X5+Y5; t2:=T1-t2; Xout:=Xout*Yout;
Yout:=Zout*t2; Xout:=Xout+T1-t1; Zout:=Xout*t2; Xout:=Xout*t3;
simplify([x6-Xout/Zout]), simplify([y6-Yout/Zout]); # Check
```

C THE CURVES

The curves below provide approximately 128-bit level of security and were found by using a modified version of the Schoof's algorithm provided with MIRACL.

– For the implementation on short Weierstrass form over $\mathbb{F}_p$ using $\mathcal{J}$, we chose the curve $E_w: y^2 = x^3 - 3x + B$, where $p = 2^{256} - 189$, $B = 0\times$fd63c3319814da55e88e9328e96273c 483dca6cc84df53ec8d91b1b3e0237064 and $\#E_w(\mathbb{F}_p) = 10r$ where *r* is the 253-bit prime:

$$115792089237316195423570985008687907853945513728367127682874172327905003185 17 .$$

The implementation corresponding to this curve is referred to as *jac256189*.

– For Twisted Edwards over $\mathbb{F}_p$ using $\mathcal{E}/\mathcal{E}^e$, we chose the curve $E_{tedw}: -x^2 + y^2 = 1 + 358x^2y^2$, where $p = 2^{256} - 189$ and $\#E_{tedw}(\mathbb{F}_p) = 4r$ where *r* is the 255-bit prime

$$28948022309329048855892746252171976963381653644566793329716531190136815607949 .$$

The implementation corresponding to this curve is referred to as *ted256189*.

– Let $E_{w-gls}: y^2 = x^3 - 3x + 44$ be defined over $\mathbb{F}_p$, where $p = 2^{127} - 1$. For the case of the Weierstrass form using GLS, we use the quadratic twist $E'_{w-gls}: y^2 = x^3 - 3\mu x + 44\mu$ of $E_{w-gls}(\mathbb{F}_{p^2})$, where $\mu = 2 + i \in \mathbb{F}_{p^2}$ is non-square. $\#E'_{w-gls}(\mathbb{F}_{p^2})$ is the 254-bit prime

$$28948022309329048855892746252171976962649922236103390147584109517874592467701 .$$

The same curve is also used in [19]. Our implementation corresponding to this curve is referred to as *jac1271gls*.

– Let $E_{tedw-gls}: -x^2 + y^2 = 1 + 109x^2y^2$ be defined over $\mathbb{F}_p$, where $p = 2^{127} - 1$. For the case of Twisted Edwards using the GLS method, we use the quadratic twist $E'_{tedw-gls}: -\mu x^2 + y^2 = 1 + 109\mu x^2y^2$ of $E_{tedw-gls}(\mathbb{F}_{p^2})$, where $\mu = 2 + i \in \mathbb{F}_{p^2}$ is non-square. In this case, $\#E'_{tedw-gls}(\mathbb{F}_{p^2}) = 4r$ where *r* is the 252-bit prime

$$7237005577332262213973186563042994240709941236554960197665975021634500559269 .$$

The implementation corresponding to this curve is referred to as *ted1271gls*.