Elliptic Curve Cryptography on Heterogeneous Multicore Platform

Sergey Victorovich Morozov

Thesis submitted to the faculty of the Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Master of Science
In
Computer Engineering

Patrick Schaumont
Peter Athanas
Michael Hsiao

August 30
Blacksburg, VA

Keywords: Elliptic Curve, Cryptography, ARM, DSP, Multicore, Multiprocessor, Point Multiplication, Prime Field, Binary Field

Copyright 2010, Sergey Victorovich Morozov

Elliptic Curve Cryptography on Heterogeneous Multicore Platform

Sergey Victorovich Morozov

ABSTRACT

Elliptic curve cryptography (ECC) is becoming the algorithm of choice for digital signature generation and authentication in embedded context. However, performance of ECC and the underlying modular arithmetic on embedded processors remains a concern. At the same time, more complex system-on-chip platforms with multiple heterogeneous cores are commonly available in mobile phones and other embedded devices. In this work we investigate the design space for ECC on TI's OMAP 3530 platform, with a focus of utilizing the on-chip DSP core to improve the performance and efficiency of ECC point multiplication on the target platform. We examine multiple aspects of ECC and heterogeneous design such as algorithm-level choices for elliptic curve operations and the effect of interprocessor communication overhead on the design partitioning. We observe how the limitations of the platform constrict the design space of ECC. However, by closely studying the platform and efficiently partitioning the design between the general purpose ARM core and the DSP, we demonstrate a significant speed-up of the resulting ECC implementation. Our system focused approach allows us to accurately measure the performance and power profiles of the resulting implementation. We conclude that heterogeneous multiprocessor design can significantly improve the performance and power consumption of ECC operations, but that the integration cost and the overhead of interprocessor communication cannot be ignored in any actual system.

ACKNOWLEDGEMENTS

I would like to thank all the faculty members under whom I have studied during my 5 years at Virginia Tech. It was your knowledge and teaching that have allowed me to complete this thesis and the Computer Engineering program at Virginia Tech. I would like to single out Kathleen Meehan who strongly encouraged and aided my application to the graduate program at Virginia Tech.

A special thank you goes to my committee members, Michael Hsiao and Peter Athanas, for reviewing this work as quickly as they did.

A separate thank you goes to Patrick Schaumont, my committee chair and advisor at Virginia Tech, who guided this research and my entire graduate tenure at Virginia Tech. I have very much enjoyed studying under you in Secure Embedded Systems group during the last two years. Working with the other graduate students in group was the most enjoyable and valuable part of my education.

TABLE OF CONTENTS

CHAPTER 1: INTRODUCTION **1**

CHAPTER 2: BACKGROUND **4**

2.1 EC Basics 4

2.2 ECC Parameters and Implementation Issues 9

CHAPTER 3: ECC ON HETEROGENEOUS MULTICORE PLATFORMS **13**

3.1 Heterogeneous Multicore Platforms in Embedded Devices 13

3.2 Public-key Cryptography for Embedded Devices 14

3.3 Related Work 15

3.4 TI OMAP 3530 Details 18

CHAPTER 4: ECC IMPLEMENTATION **22**

4.1 Initial Design Partitioning 22

4.2 ARM to DSP Communication 24

4.3 Rethinking Design Partitioning 27

4.4 ECC Algorithm Implementations 28

4.5 Development Environment 29

CHAPTER 5: RESULTS **31**

5.1 Field Multiplication 31

5.2 Point Multiplication 32

5.3 Power Efficiency 34

5.4 TinyECC Comparison 37

CHAPTER 6: FUTURE WORK **40**

6.1 Performance and Efficiency Improvement 40

6.2 ECC Functionality Extension 41

6.3 Extending Co-design Approach 41

CHAPTER 7: CONCLUSION **42**

BIBLIOGRAPHY **43**

APPENDIX A: FORMULAS FOR EC POINT OPERATIONS **45**

LIST OF FIGURES

FIGURE 1. ECC IMPLEMENTATION PYRAMID 5
FIGURE 2. POINTS OF $y^2=x^3+4x+240$ OVER $GF(29)$ 8
FIGURE 3. BEAGLEBOARD 17
FIGURE 4. OMAP3530 SYSTEMS DIAGRAM 18
FIGURE 5. INITIAL ECC MULTICORE PARTITIONING 23
FIGURE 6. INTERPROCESSOR COMMUNICATION ON OMAP3530 24
FIGURE 7. INTERPROCESSOR COMMUNICATION SEQUENCE 25
FIGURE 8. MODIFIED ECC MULTICORE PARTITIONING 28
FIGURE 9. EXECUTION TIME COMPARISON OF POINT MULTIPLICATION ON OMAP3530 ... 33
FIGURE 10. POWER TRACE OF POINT MULTIPLICATION WITH ARM-ONLY VERSION 34
FIGURE 11. POWER TRACE OF POINT MULTIPLICATION WITH ARM+DSP VERSION 35
FIGURE 12. ENERGY COST COMPARISON OF POINT MULTIPLICATION ON OMAP3530...... 37
FIGURE A.1. ALGORITHM FOR POINT MULTIPLICATION ON PRIME FIELD EC 45
FIGURE A.2. POINTDOUBLE() FUNCTION 46
FIGURE A.3. POINTADD() FUNCTION 47
FIGURE A.4. ALGORITHM FOR POINT MULTIPLICATION ON BINARY FIELD EC 48
FIGURE A.5. MDOUBLE() FUNCTION 49
FIGURE A.6. MADD() FUNCTION 50

List of Tables

Table I. Recommended Prime Field EC Parameters 10
Table II. Recommended Binary Field EC Parameters 10
Table III. Operation Counts for Point Multiplication in 192-bit Prime Field. ... 11
Table IV. Comparison of ARM and DSP Cores in OMAP3530 19
Table V. Round-trip Message Latency for ARM to DSP Communication 26
Table VI. Number of Function Calls for ECC Operations. 27
Table VII. Algorithms for our ECC Implementation 29
Table VIII. Field Multiplication on OMAP3530 32
Table IX. Point Multiplication on OMAP3530 33
Table X. Power Characteristics of OMAP3530 during Point Multiplication ... 36
Table XI. Energy Cost of Point Multiplication on OMAP3530 37
Table XII. Comparison of Point Multiplication on OMAP3530 and Imote2 38

CHAPTER 1

INTRODUCTION

Modern mobile devices and embedded systems are designed to provide an ever increasing number of services for the applications and the end-user. Cryptographic services in particular are in demand for mobile phones, sensor networks, and RFID systems.

Public-key cryptography has become a popular method to implement secure communication, including authentication and key-exchange. In contrast to traditional symmetric-key cryptography systems which rely on shared common secrets among two parties, in a public-key system each party possesses a public key freely visible to outsiders and a corresponding private key that is never revealed to others. The two keys share a mathematic relationship that allow data to be encoded or signed with a public key, and decrypted and verified only with the corresponding private key. In combination with other cryptographic primitives, public-key cryptography can provide authentication, confidentiality, verification and non-repudiation services. Mathematical properties of the system make computing a private key for a given public key computationally infeasible, and keep the public key cryptosystem secure.

Unfortunately computational costs associated with public key cryptography tend to be quite expensive. Public-key algorithms like RSA, DSA, and elliptic curve cryptography (ECC) all deal with operands hundreds to thousands of bits in size and use modular arithmetic for the underlying computations. Modular multiplication of large numbers up to several thousand bits in size tends to be the critical underlying part of public-key cryptography and can be prohibitively expensive on some types of embedded processors. ECC stands out among other public key algorithms due its comparatively smaller operand sizes, however ECC utilizes an additional layer of algorithms on top of the modular arithmetic operations.

While algorithmic-level improvements are continually being proposed, typically they offer only marginal improvements and tradeoffs. This means that the designer must look to hardware solutions to solve the performance problem. Typically this implies either expanding the capabilities of an existing processor with specialized instructions or designing completely new co-processors. Researchers often approach the problem by coming up with an HDL description of the new co-processor or datapath, and conclude the research with a simulation or an FPGA prototype of the design. At best, the resulting proposed design is still years away from appearing in the target platforms. At worst, the presented design may completely neglect the overhead of implementing the new hardware module in a real system and may present a greatly exaggerated performance speedup.

We approached this problem from another angle. We have observed that recent trends in mobile and embedded computing have pushed increasingly complex System-on-Chip designs into existing platforms such as TI's OMAP and DaVinci chips. These SoCs now frequently include multiple heterogeneous cores: a standard RISC core could be augmented with DSP, SIMD extensions, and a graphics processing unit. The primary question that this work addresses is what benefit can designers expect to see by tightly mapping an application to a specific multicore platform. Previous research has shown that VLIW DSP processor could efficiently implement large operand modular arithmetic and ECC [1]. Our objective was to apply these methods to an existing heterogeneous platform, and study the resulting implementation to analyze the integration costs that arose. This approach contrasts with the popular simulation-only accelerations of ECC and the flexible-implementation prototypes that utilize configurable platforms like FPGAs.

The contributions of this thesis are as follows. First we examine the design space of elliptic curve cryptography system on a heterogeneous multicore platform. We demonstrate our implementation on the TI OMAP 3530 chip, which contains ARM Cortex A8 and a C64x+ DSP cores. We also analyze the effect of core-to-core communication in our platform and discuss its relevance to multicore partitioning. Our implementation provides high speed ECC cryptography framework for a high level operating system on the ARM. We show that the multicore implementation demonstrates a significant speed-up over existing production grade ECC libraries for a general-purpose

processor. We make a strong effort to create a fair comparison between our ARM-DSP implementation and ARM-only implementation by utilizing identical ECC algorithm.

This research demonstrates an elliptic curve implementation using varying sizes of prime and binary fields, rather than examining a single small prime field as is commonly done in other related research efforts. This allows us to characterize performance improvements for ECC of varying field type and field size, rather than only extrapolating our results to other designs. Further, since we implement our designs on a production grade platform as opposed to a prototyping system, we are able measure the power efficiency of the implementations rather than rely on crude data sheet estimates. The results further support our case for a multicore design by providing up to over 4x reduction in power.

The remainder of this thesis is organized as follows: in Chapter 2 we review the basics behind elliptic curve cryptography. In Chapter 3 we describe our target platform type, qualify the usefulness of ECC on that platform type, discuss related works, and select the prototype platform for our implementation. In Chapter 4 we describe our design approach to multicore ECC and present our implementation. In Chapter 5 we provide an analysis of the results and comparison with another implementation. Chapter 6 introduces ideas for future work, and Chapter 7 presents the conclusion.

CHAPTER 2

BACKGROUND

In this chapter we review the basics behind elliptic field cryptography and the underlying field arithmetic. We also review the recommended elliptic curve parameters and implementation issues.

2.1 EC Basics

Elliptic curves may be defined over any finite field (also known as Galois field), $GF(q)$, where q is referred to as the characteristic of the field. In this work we consider only prime fields $GF(p)$, where p is a prime number, and $GF(2^m)$ where the characteristic is a primitive binary polynomial of degree m.

An elliptic curve defined over a finite field describes a finite set of points, referred to as *points on the curve*. Figure 1 shows a system level description of elliptic curve cryptography implementation in the shape of a pyramid. At the second level of the pyramid, the elliptic curve defines several *Elliptic Curve (EC) Point Operations* that allow manipulation of the *points on the curve*. The fact that the elliptic curve is defined over a finite field implies that the arithmetic used by the *EC Point Operations* is finite field arithmetic, also referred to as the underlying *Field Operations* seen at the bottom of the pyramid. The *EC Point Operations* can be used in such a way as to allow arithmetic between elliptic curve points and scalar factors (i.e. simple integers), creating *EC Scalar Operations*. Only one such operation is actually used in elliptic curve cryptography: elliptic curve point multiplication (often referred to as simply "point multiplication"). The *EC Cryptography Operations* and protocols are cryptographic algorithms that provide actual cryptographic services like encryption, authentication, and signature generation and verification. All *EC Cryptography Operations* rely on point multiplication.

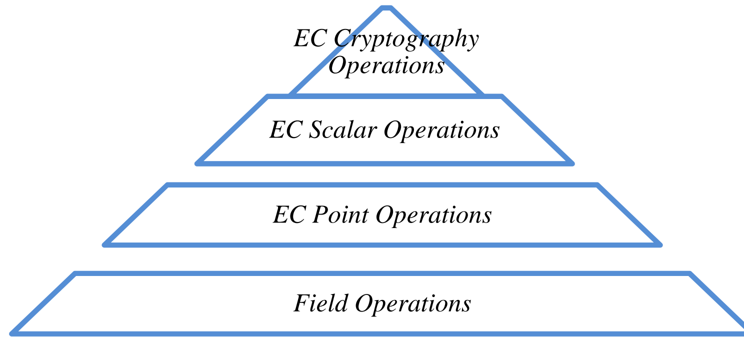

Figure 1. ECC Implementation Pyramid

The next few sections give an overview of each level of the pyramid starting with the bottom *Field Operations* and continuing up.

2.1.1 Field Operations

EC Cryptography Operations require several *Field Operations* in their formulas: addition, subtraction, multiplication, and inversion. These finite field arithmetic operations are defined for elements of the finite field, meaning that all the operations are modular arithmetic with characteristic of the field as the modulus.

For example, prime field $GF(29)$ defines the set consisting of all positive integers between the prime number 29 and 0, or $[0, 1, 2, 3, \dots, 28]$. Multiplication of elements 2 and 15 in the field would yield 1 since product is reduced 29, the characteristic of the field:

$$2 * 15 \; mod \; 29 = 30 \; mod \; 29 = 1 \tag{1}$$

Field addition and subtraction methods are likewise straight forward: plain-integer arithmetic followed by reduction. Inversion of operand *a* involves finding the inverse of a, a^{-1}, such that

$$a * a^{-1} \; mod \; 29 = 1 \tag{2}$$

As evident from the previous example the inverse of 15 in $GF(29)$ is 2. We implement the inverse operation using the well known method based on the ancient Euclidean algorithm for finding the greatest common divisor. The method is detailed in [2], [3].

In binary field $GF(2^m)$, the elements of the field are binary polynomials, i.e. polynomials whose coefficients are either 0 or 1, with the degree less than *m*. The

characteristic of the field is an irreducible *m*-degree binary polynomial. For example, binary field $GF(2^4)$ has one possible characteristic irreducible binary polynomial

$$f(x) = x^4 + x + 1 \tag{3}$$

and includes as its elements all polynomials *A(x)* of the form

$$A(x) = a_3x^3 + a_2x^2 + a_1x^1 + a_0\,, \tag{4}$$

where each coefficient $a_i \in \{0\,,1\}$. A binary polynomial can be efficiently represented as a binary number where i-th bit is the coefficient a_i of the current term:

$$x^4 + x + 1 = (0001\ 0011)_2 = \mathrm{0x13}\ . \tag{5}$$

Arithmetic in binary field is likewise modular. Addition of binary polynomials is equivalent to performing Exclusive-OR operation on their binary number representations. Multiplication requires computing partial products and then adding them together using the field addition, i.e. Exclusive-OR. This multiplication is sometimes referred to as carry-less multiplication or XOR-multiplication. The product of the multiplication must be reduced by the characteristic polynomial. Further discussion of binary field multiplication is found in [4]. Our binary field inversion is implemented nearly identically to the prime field inversion [2].

2.1.2 EC Point Operations

Points on the elliptic curve over *GF(p)* are defined by the simplified equation

$$y^2 = x^3 + ax + b \tag{6}$$

where $a, b \in GF(p)$ are elliptic curve parameters, and the ordered pair (x, y) are a point on the curve with $x, y \in GF(p)$ as well. The points defined by the elliptic curve $y^2 = x^3 + 4x + 20$ over $GF(29)$ are shown in Figure 1. For binary field the simplified equation for elliptic curve is slightly different:

$$y^2 + x\,y = x^3 + ax^2 + b. \tag{7}$$

Additional variations on these equations exist, but all elliptic curves discussed in this paper fall under the above equations.

The central operation with elliptic curve points is point addition: given two points on the curve $Q(x_1, y_1)$ and $R(x_2, y_2)$ computing their geometric sum to find $P(x_3, y_3)$. It is important to realize that point addition is not done simply by adding x_1 with x_2 and y_1 with y_2. Rather the formula for point addition reduces down to

$$P(x_3, y_3) = Q(x_1, y_1) + R(x_2, y_2) \text{ in } GF(p) \quad (8)$$

$$x_3 = \left(\frac{y_2 - y_1}{x_2 - x_1}\right) - x_2 - x_1 \quad (9)$$

$$y_3 = \left(\frac{y_2 - y_1}{x_2 - x_1}\right)(x_1 - x_3) - y_1 \quad (10)$$

over prime field. One must realize that all arithmetic operations in the above formulas are field operations. That is, $y_2 - y_1$ is actually $y_2 - y_1$ mod p and so on.

Notice that above formulas will not work if we try to add a point to itself: the $x_2 - x_1$ term in the denominator of Equations 9 and 10 would evaluate to 0. Adding a point to itself is known as point doubling operation, and is a special case of point addition. The formula for point doubling is

$$P(x_3, y_3) = 2 \cdot Q(x_1, y_1) \quad (11)$$

$$x_3 = \left(\frac{3x_1^2 + a}{2y_1}\right) - 2x_1 \quad (12)$$

$$y_3 = \left(\frac{3x_1^2 + a}{2y_1}\right)(x_1 - x_3) - y_1 \quad (13)$$

over prime field. For binary field the formulas change to

$$P(x_3, y_3) = Q(x_1, y_1) + R(x_2, y_2) \text{ in } GF(2^m) \quad (14)$$

$$x_3 = \left(\frac{y_2 + y_1}{x_2 + x_1}\right)^2 + \left(\frac{y_2 + y_1}{x_2 + x_1}\right) + x_1 + x_2 + a \quad (15)$$

$$y_3 = \left(\frac{y_2 + y_1}{x_2 + x_1}\right)(x_1 + x_3) + x_3 + y_1 \quad (16)$$

for point addition and to

$$P(x_3, y_3) = 2 \cdot Q(x_1, y_1) \text{ in } GF(2^m) \quad (17)$$

$$x_3 = x_1^2 + \frac{b}{x_1^2} \quad (18)$$

$$y_3 = x_1^2 + \left(\frac{x_1 + y_1}{x_1}\right)x_3 + x_3 \quad (19)$$

for point doubling.

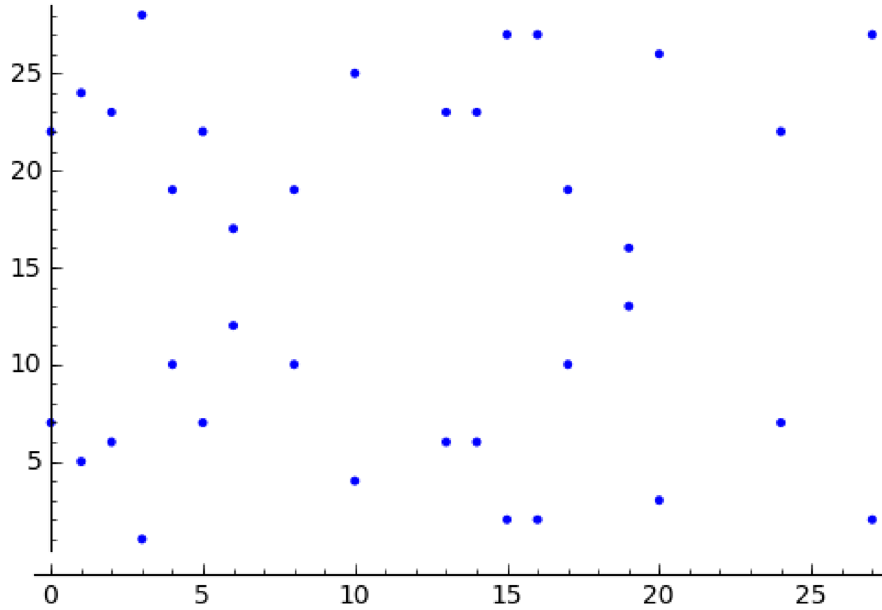

Figure 2. Points of *y^2=x^3+4x+20* over *GF(29)*

2.1.3 EC Scalar Operations

EC Point Operations can be repeated a defined number of times to create a scalar operation. The central operation in ECC is the point multiplication, which multiplies a point P on a curve with a scalar number k. Mathematically this can be expressed as adding a point P k-times to itself to compute a new point Q:

$$Q = k \cdot P = P + \ldots + P \quad \text{(addition repeated } k\text{-times total).} \tag{20}$$

The point multiplication can be rewritten to use fewer operations if we use point doubling when appropriate. For instance if the $k = 7$ point Q can be computed as

$$Q = (2 \cdot (2 \cdot P + P)) + P \tag{21}$$

which requires only 4 *EC Point Operations* (two doublings and two additions), rather than 7 additions with the naïve approach.

Point addition over an elliptic curve forms a cyclic group: adding point P to itself n-times, where n is the number of points on the curve, allows us to traverse every point on the curve and end up back at point P. In terms of point multiplication this means that on every prime field or binary field elliptic curve with n points

$$Q = n \cdot P = P. \tag{22}$$

This property leads to the use of EC point multiplication in cryptography.

2.1.4 EC Cryptography Operations

In public-key cryptography every entity possesses a freely displayed public-key and a secret corresponding private key that is kept hidden and safe, thus forming a public-private key pair. EC public-key algorithms (*EC Cryptography Operations)* then allow for entities to engage in cryptographic protocols to provide encryption, authentication and digital signature schemes. For ECC, the point multiplication is central to all *EC Cryptography Operations*. These operations and protocols are not discussed in this thesis.

We limit our discussion to explaining that in ECC, the point multiplication $Q = k \cdot P$ forms the foundation of public-key crypto system: the public-private key pair. Some predefined point P serves as a base point for the curve and is considered to be a parameter of the curve; a randomly selected scalar value k acts as the secret key, and the point Q computed through point multiplication acts as the corresponding private key. Therefore the process of key pair generation for ECC is dependent entirely on point multiplication. All other *EC Cryptography Operations* also utilize point multiplication in some manner.

The security of ECC lies in the irreversibility of the point multiplication. Given a point P, and the resulting point multiplication product $Q = k \cdot P$, it is computationally infeasible to determine k given only knowledge of Q and P. This problem is known as elliptic curve discrete logarithm problem (ECDLP). The best known method for solving ECDLP is the Pollard's rho algorithm, which involves randomly traversing the points of an EC curve in order to find a collision. Pollard's rho algorithm has an expected running time of $\sqrt{\pi n}/2$ where n is the number of points on the curve. The number of points on the prime field based EC curve is approximately p, the size of the prime. Thus, selecting a large enough characteristic makes the Pollard rho algorithm impractical.

2.2 ECC Parameters and Implementation Issues

As discussed previously, the values of a and b in the equation of the elliptic curve, as well as the base point used for key generation in ECC are all parameters for a given elliptic curve. There are additional parameters used in *EC Cryptography Operations*, which are not discussed here. The issue that must be addressed by any ECC implementation is the selection or generation of these parameters.

While it is possible to randomly generate ECC parameters at runtime, this is seldom done for in ECC implementations, especially in constrained or embedded platforms. Instead an ECC implementation typically supports a set of predefined parameters. Possible sources of these predefined parameters are the standards published by National Institute of Standards (NIST) [5] or by Standards for Efficient Cryptography Group (SECG) [6]. Table I shows the names of the prime field elliptic curves from SECG and the underlying prime number p used in this project, while Table II shows the equivalent for binary field elliptic curves. The complete definition of parameters is listed in the actual standard [6].

Table I. Recommended Prime Field EC Parameters

EC Name	Parameter p (Prime Number)
secp160r1	$2^{160} - 2^{31} - 1$
secp224r1	$2^{224} - 2^{96} + 1$

Table II. Recommended Binary Field EC Parameters

EC Name	Parameter *f(x)* (Irreducible Polynomial)
sect163r1	$x^{163} + x^7 + x^6 + x^3 + 1$
sect409r1	$x^{283} + x^{12} + x^7 + x^5 + 1$
sect409r1	$x^{409} + x^{87} + 1$

Another advantage of using these predefined parameters is the opportunity to incorporate known and documented improvements for the various levels of ECC algorithms. For instance, there are known fast reduction techniques for the primes [1] in Table I and polynomials [4] in Table II. There are also known optimizations for point addition and point doubling for prime field elliptic curves with parameter $a = -3$ or for binary field curves with $a = 1$ [3].

While this work does not focus on the mathematical optimization of ECC algorithms, some optimizations have been shown to be essentially required to achieve good performance with ECC. The formulas for computing point doubling and point addition presented in Section 2.1.2 all contain a field division operation. This operation

can be implemented using field multiplication and inversion; however the inversion operation tends to be at least an order of magnitude more expensive than other operations in computational terms. To avoid frequent use of the inversion operation the designer should make use of projective coordinate point representation. With projective coordinates, an elliptic curve point is represented by an ordered triple variables (X, Y, Z) as opposed to the standard affine coordinates where a point represented as (x, y). In our implementations the prime field curves use the Jacobian projective coordinates where the relationship between projective and affine coordinates is

$$(X, Y, Z) \rightarrow \left(\frac{X}{Z^2}, \frac{Y}{Z^2}\right),$$

and the binary field curves use López-Dahab projective coordinates where

$$(X, Y, Z) \rightarrow \left(\frac{X}{Z}, \frac{Y}{Z^2}\right).$$

Projective coordinates increase the number of field multiplications in point addition and point doubling, but eliminate modular inversion operations during point addition and point doubling. With projective coordinates, field inversion is required only at the end of point multiplication in order to convert the result back to the standard affine representation.

Table III. Operation Counts for Point Multiplication in 192-bit Prime Field.

Coordinate System	Point Operation		Field Operation	
	Addition	Doubling	Multiplication	Inversion
Affine	95	191	977	286
Projective	95	191	2420	1

Table III estimates the expected number of operations for a single point multiplication in 192-bit prime field [3]. The cost of field additions is nearly negligible compared to multiplications and is thus not shown. Given the approximation that one inversion has a similar execution time as 80 field multiplications, it is clearly evident that the projective coordinate are the preferred method. In fact, by their estimates the affine based implementation would be nearly a magnitude slower [3]. Though the operation counts change for binary field, and the order of magnitude difference is reduced, the

relationship holds true: projective coordinate point representation is required for optimal performance of ECC.

CHAPTER 3

ECC ON HETEROGENEOUS MULTICORE PLATFORMS

One of the goals in this research was to target an existing, actively used embedded platform, rather than attempting to design or prototype the future version of the said platform. We believed that by closely examining the capabilities of such a platform, we could come up with a much more optimal implementation of ECC. Because of this, we considered the microprocessor based SoCs found in common embedded devices, rather than an FPGA based platform or custom ASIC designs used to prototype future systems. This section describes the usage of heterogeneous multicore platforms in embedded devices, justifies the usefulness of ECC on such devices, and discusses related work to improving ECC on such devices, and identifies our target platform.

3.1 Heterogeneous Multicore Platforms in Embedded Devices

Multiple heterogeneous processors have been commonplace in embedded systems for years, though often only on the same circuit board rather than in the same chip. For example, cell phones typically contain a general microprocessor for controlling the device and a separate DSP to handle the actual signal processing. Similarly, many wireless nodes like the Imote2 node for TinyOS [7] contain a general purpose processor and a wireless transceiver with its own internal signal processing, which acts similar to the phone's DSP.

TI OMAP series is an example of embedded platform's evolution into complex heterogeneous system. Currently in its third generation, OMAP is popular among cell phone manufactures such as Nokia and Samsung, and has also found its way into internet tablets and smartphones like Palm Pre and Motorola Droid [8]. Though many variations of the device exist, most versions combine an ARM general purpose CPU with a DSP,

packaged together with a myriad of other peripherals, all under one chip. The ARM on the OMAP style chips is designed to run some operating system, rather than "C on hardware" approach seen on lower end embedded systems. Various Linux distributions, Android, and Windows CE have all been ported and used in OMAP based systems.

The OMAP has also found its way into other embedded devices, including robotics projects for computer vision processing [9]. Other commercial applications for OMAP like chips could include handheld RFID readers/scanners. While significantly more powerful than some lower end wireless sensor nodes, overall the OMAP series is only slightly ahead of TinyOS Imote2 wireless sensor platform [7]. This means an OMAP like multicore chip could potentially appear in higher end wireless sensors, secure wireless routers, and similar devices.

3.2 Public-key Cryptography for Embedded Devices

The availability of ECC or other public-key cryptography services on these mobile devices could prove very useful to the end user. A defining characteristic of ECC compared to other public-key cryptography methods such as RSA is the comparatively smaller operand sizes. For example to achieve the security level approximately equivalent to 3072 bit operand RSA system, the elliptic curve system requires approximately 256 bit parameters and operands. This is due to the fact that the ECDLP problem is computationally more difficult than RSA's underlying integer factorization problem [3]. Another popular public-key system, DSA relies on integer discrete logarithm problem (DL) and is roughly similar to RSA's factorization in terms of computational difficulty.

In [10] Gupta specifically compares RSA and ECC in the context of secure handshake protocols commonly used for secure web based transactions. The performance is measured in terms of latency for a single handshake, and in the number of handshakes the server can establish per second. The results show that 193-bit ECC implementation is significantly faster than equivalent 2048-bit RSA system for all types of client-server handshakes. The authors note that the performance advantage of ECC over RSA is expected to increase for higher key sizes.

This makes ECC uniquely suited for resource constrained applications. In the case of wireless sensor networks or RFID technologies ECC could actually improve the security and performance of the system. Moreover, while public-key based encryption has not yet made its way into the communication protocols of the cell phone universe, performance is likely one of the factors for keeping it out. Elbaz and Anuar have previously described schemes that would utilize public key cryptography for mobile messaging [11, 12]. Elbaz lists secure browsing from cell phones, digital signatures and authentication for mobile payments, and digital rights management as other possible consumers of public key infrastructure on mobile phones [11]. Tillich studies the implementation of public-key point multiplication on Java Virtual Machine enabled cell phones, like the Nokia 6610 and Sony Erickson P900, citing e-commerce as a possible application [13]. The study finds that the performance of ECC on JVM is unacceptable on some of these devices.

The above section presents the justification for our type of platform and for applications of ECC in this type of platform. Heterogeneous multicore embedded platforms are already here, sitting in every cell phone, and in multiple other embedded devices. While the use of ECC on these embedded devices still falls into “niche” category, rather than “common”, improved performance of ECC is one of the keys that will lead wider adoption in the future.

3.3 Related Work

Because of the potential usability of ECC on embedded processor and resource constrained platforms considerable research has been done on improving the performance of ECC in this domain. Frequently the researchers choose to design custom hardware to support more efficient *Field Operations* or *EC Point Operations* in the form of co-processors or custom-instruction additions.

Alrimeih discusses performance-security tradeoffs for a prime field ECC coprocessor [14]. While they carefully describe the performance of their co-processor down to each clock cycle, the performance overhead between the co-processor and the main processor is left unmentioned. Moreover, the researchers do not offer a comparison

with a baseline system, making it difficult to determine the resulting benefit of the co-processor.

Guo proposes an SoC implementation of ECC system composed of a RISC core and a specialized coprocessor on an FPGA platform [15]. This time the researchers do pay close attention to the bus integration of the system. Working with an FPGA allows the designer nearly complete customization of the underlying platform and the corresponding mapping field arithmetic and ECC algorithms, leading to a much larger design field. Further, FPGA co-processor design allows for a very closely coupled processor-bus-coprocessor interface. By contrast, our work emphasizes designing on a fixed-core SoC platform with a more complex interprocessor communication mechanism.

Another popular approach for ECC implementations is to look at algorithm-only optimizations for some set of embedded platforms. The best example of this is the development of TinyECC library for embedded sensor nodes [7]. The target platforms for this library are any TinyOS based wireless sensor platforms, such as the MICAz, TelosB, and Imote2 platforms. TinyECC is designed to provide ECC crypto using services for the wireless communication channels used by these sensor nodes, and is designed exclusively with secp160r1 curve mentioned in previous chapter. The authors of the library implement several optional optimizations that allow for trade-off between code-size, runtime memory usage, and performance. The results indicate that the use of projective coordinates is absolutely critical, regardless of processor implementation. The next most important factor for the performance is the optimization of field multiplication. While the lower end TinyOS platforms are not in the same category as the platforms targeted by us, the higher end Imote2 has a 32-bit ARM processor running at up to 416 MHz and is compared with our own implementation and platform in the results section. The primary difference between this research and TinyECC, is that our work focuses on optimizing the design of ECC on a fixed heterogeneous multicore platform, while the latter focuses writing a generic library without closely tailoring the design to specific hardware of one platform.

Yan designs a prime field-ECC system for a TMS320C6416 DSP from Texas Instruments [1]. The underlying context for that project was to demonstrate the viability of ECC for DSP powered sensor nodes, such as those found in underwater acoustic

sensor networks. Parts of the code for prime field ECC implementation examined in this work are taken from that project with permission of the authors. Our work differs and expands on that project because we consider a multicore heterogeneous system-on-chip platform, where we provide ECC functionality as a service to an operating system and demonstrate the usefulness of the DSP as coprocessor/accelerator for ECC computations. By contrast, the original work was targeting a platform composed just of a DSP processor. We examine how the cost of interprocessor communication affects both the design field the resulting performance of ECC in such a multicore platform, and we provide a fair comparison to single-core-only implementation on the same platform.

Beyond the embedded world ECC, has appeared in several open source libraries. OpenSSL [16] library is among the most developed and has compared favorably with other open source libraries in a comparative study [17]. OpenSSL includes routines for EC point multiplication in nearly all recommended binary and prime fields. Though targeted at desktops and servers with Unix-like operating systems, OpenSSL has been compiled for embedded Linux-based devices, including our target platform. Because it implements all the elliptic curves that we do, and because it is available on our platform

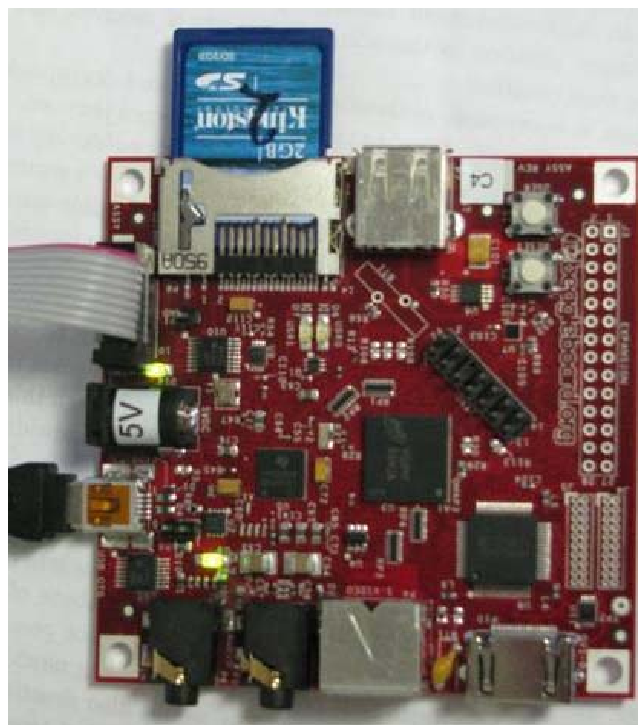

Figure 3. BeagleBoard

allowing for an ideal comparison, we use OpenSSL as the primary benchmark against own our implementation.

3.4 TI OMAP 3530 Details

As mentioned, the TI OMAP line is a prime example of the type of multicore platform we are targeting in this research. Specifically, we chose to use the TI's BeagleBoard with OMAP3530 device as our target platform. The platform is pictured in Figure 3. This platform is low-cost and relevant to existing products, and it has garnered significant attention in the amateur and professional embedded device development community [18]. Moreover, we were aware of existing research of improving the performance of binary field operations on this board [4].

Figure 4 shows the organization of the relevant subsystems of OMAP3530. The central part of OMAP3530 is the microprocessor with the ARM Cortex-A8 core. This subsystem itself includes multiple submodules: processor's level 1 and level 2 caches, interrupt controller, and bridge modules to the main bus referred to as L3 interconnect.

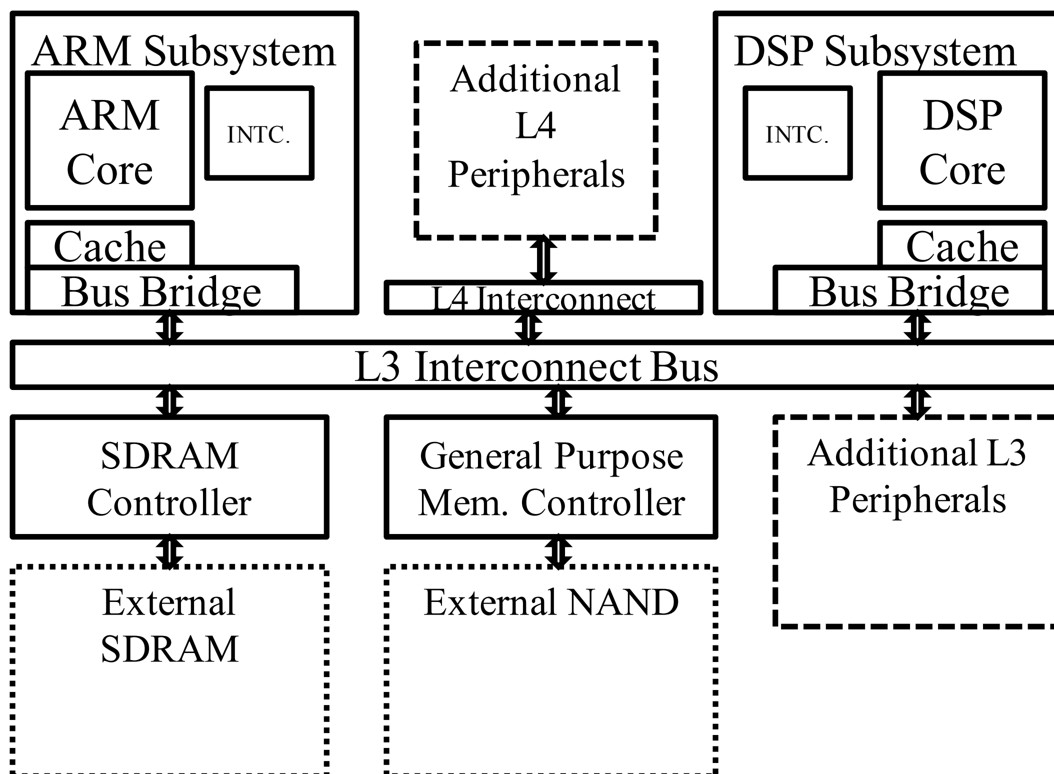

Figure 4. OMAP3530 Systems Diagram

The C64x+ DSP is located in a separate subsystem, with its own caches, interrupt controller, and bus bridge/memory management unit. The L3 Interconnect bus is implemented using Open-core protocol (OCP), and connects the ARM, DSP, SDRAM controller and other subsystems together. There is also an additional bus referred to as the L4 Interconnect where multiple other peripherals are connected. In the case of the BeagleBoard, there is 256 MB of RAM and 256 MB of NAND memory located on the OMAP3530 chip in a package-on-package configuration.

Table IV. Comparison of ARM and DSP Cores in OMAP3530

	ARM Core	DSP Core
Operating Frequency	500 MHz	360 MHz
Data Caches	32K L1 64K L2 Additional 80K RAM	16K L1 256KB L2 (outside the module)
Main Memory	256 MB total, split configured at boot	
Connection to L3 Interconnect	64-bit	64-bit
Hardware Multipliers	Yes	Yes (2)
Hardware Carry-less Multiplayer	No	Yes (2)
Operating System	Linux, Android, None	DSPBIOS
Primary Compiler	ARM gcc	C6x Code Gen. Tools

Table IV summarizes some of the key features that will be important in the implementation of ECC. The ARM and the DSP run at different clock rates: nominally 500MHz for the ARM and 360 MHz for the DSP. The higher clock rate is expected to give the ARM subsystem faster access to main memory. The slower-clocked DSP is still likely to achieve better arithmetic throughput due to multiple instruction execution units in the datapath. Looking at the cache sizes, given that the operand sizes even for the larger fields will be under 64 bytes and that expected variable count for ECC is fairly small, we expect both the DSP and ARM to be able to contain all temporary variables in L1 caches. The 256 MB of “on package” memory located on top of OMAP3530 can be

memory mapped between the processors at boot. Again, the memory configuration will not be critical, since EC point multiplication by itself is not at all memory intensive. We also observe that both the ARM and the DSP have hardware multipliers, which will prove very useful for prime field multiplication. On the other hand, the ARM lacks the support for hardware carry-less multiplication needed for binary field, while the DSP supports XOR-Multiply instruction with the maximum operand size of 32 and 9 bits. This difference is likely to improve the performance of binary field operations on the DSP.

In a typical embedded device, the ARM processor would be running an operating system. Since one of our goals is to study the performance of elliptic curve cryptography under realistic conditions, the ARM will be running a Linux distribution. Utilizing the DSP requires running TI's DSPBIOS operating system. The purpose of DSPBIOS is to take care of interrupt handling, memory mapping and bus access, as well as task scheduling. It also important to realize that the compilation environments for both processors are completely separate. The ARM side of the design can be compiled with an ARM port of the gcc compiler, while any DSP program must be compiled with TI's C6x Code Generation Tools.

To the best of our knowledge, no previous attempts to utilize the OMAP's DSP for cryptography related applications have been done. Rather, the OMAP3530 and specifically BeagleBoard are targeted at multimedia applications. Multiple additional peripherals in the chip and on the board provide additional multimedia related functionality; there are microcontrollers for video output, a PowerVR Graphics Accelerator submodule, peripherals to control audio channels and flash SD cards all located in the OMAP3530 chip, and additional power regulator and audio output chips located on the BeagleBoard adjacently to the OMAP3530. The typical use of the C64x+ DSP on the BeagleBoard has been limited to acceleration of multimedia codecs. The main product provided by TI for the OMAP DSPs is in fact the Codec Engine software stack. Codec Engine includes algorithms for image and video encoding/decoding, voice processing, and also is designed to allow the end-user to define his or her own algorithms for the DSP.

Though our design will not explicitly use Codec Engine framework, it could be integrated with Codec Engine at a later point, or perhaps even lead to the development of Crypto Engine.

CHAPTER 4

ECC IMPLEMENTATION

Implementing an ECC system requires building up the ECC pyramid shown in Figure 1. Moreover, since our goal is to utilize the DSP to improve the performance we must also consider how the ECC implementation will be partitioned across the cores.

Field Operations level requires a working field arithmetic library for the target platform. For *Field Operations* on the ARM, we would be able to utilize routines from the OpenSSL library if needed. Field arithmetic on TI's DSPs has previously been studied in [1] and [4]. Our implementation of field arithmetic is based on their work, including the use of the original source code with permissions of the authors.

Binary field arithmetic was developed by authors in [4]. The target platform for that work was also TI's OMAP3530 BeagleBoard. The work described the methodology for writing compiler friendly C code for field arithmetic. Binary field multiplication on the DSP was shown to be significantly faster relative to the ARM, due to the use of the DSP's carry-less multiplication instruction (XORMPY) and high utilization of DSP's datapath.

Prime field operations are based on the research described in [1]. The target platform there was a stand-alone TMS320C6416 DSP. The underlying field operations were written in TI's DSP assembly language to maximize performance. The C6416 DSP is structurally very similar to the C64x+ DSP, and when profiled our port of that work achieved nearly identical performance.

4.1 Initial Design Partitioning

With functional field arithmetic implementation, the primary design question became how to partition the design across the ARM and the DSP. The results showed that field multiplication operations were significantly faster on the DSP. However, due to the availability of the ARM and its higher clock rate, it was desirable to come up with a

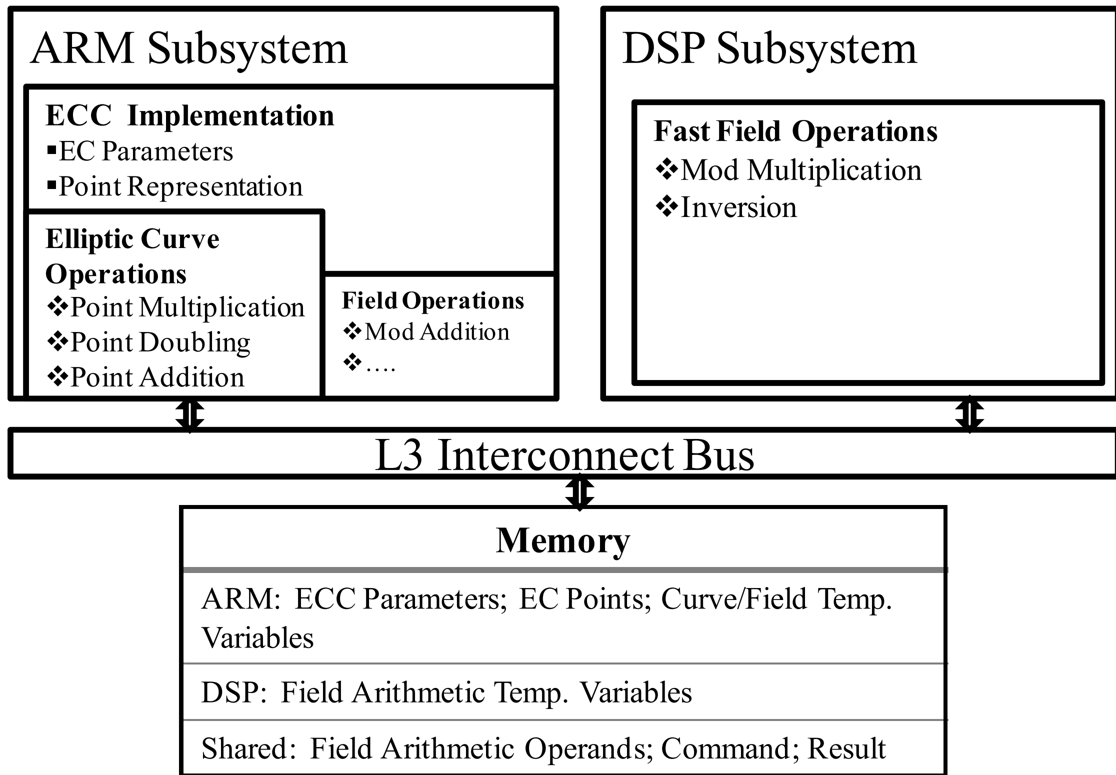

Figure 5. Initial ECC Multicore Partitioning

co-design implementation that would offload field multiplication to the DSP and execute other operations on the ARM.

The original design partitioning is shown in Figure 5. The partitioning calls for implementing only the slower field operations on the DSP. In order to perform an ECC point multiplication, the ARM would select the desired elliptic curve, compute or recall the parameters for that specific curve, determine the appropriate point representation, and begin carrying out point addition and point doubling operations. Whenever a field multiplication or inversion needed to be performed, the ARM would send the computation off to the DSP, and if the algorithm allowed it, continue performing some related arithmetic while waiting for the answer from the DSP.

Under this design, the majority of memory usage would be on the ARM memory section. The ARM memory section would contain elliptic curve parameters and points, temporary variables produced during EC Curve Operations and temporary variables from field addition. The shared memory would be used for communication: storing the operands for field multiplication or inversion on the DSP, storing the command for the

DSP to specific which instruction to carry out, and lastly storing the computed result. The DSP memory section would only store some temporary variables needed for field multiplication and inversion computations.

The viability of this partitioning depended largely on the overhead of ARM to DSP communication.

4.2 ARM to DSP Communication

In order to determine if the design presented in Figure 5 was viable, we investigated and profiled the interprocessor communication on the OMAP3530 platform.

To facilitate interprocessor communication the OMAP3530 chip includes an interprocessor communication (IPC) module attached to the L4 Interconnect as shown in Figure 6. The IPC consists of a memory-mapped mailbox which contains small hardware FIFOs and connections to the interrupt controllers of both ARM and DSP subsystems. These FIFOs store 4 messages of 32-bits in size, and are typically used to exchange memory addresses. The actual sharing of data is done through a shared memory (some predefined section of the RAM on the L3 bus). At boot, sections of memory are assigned

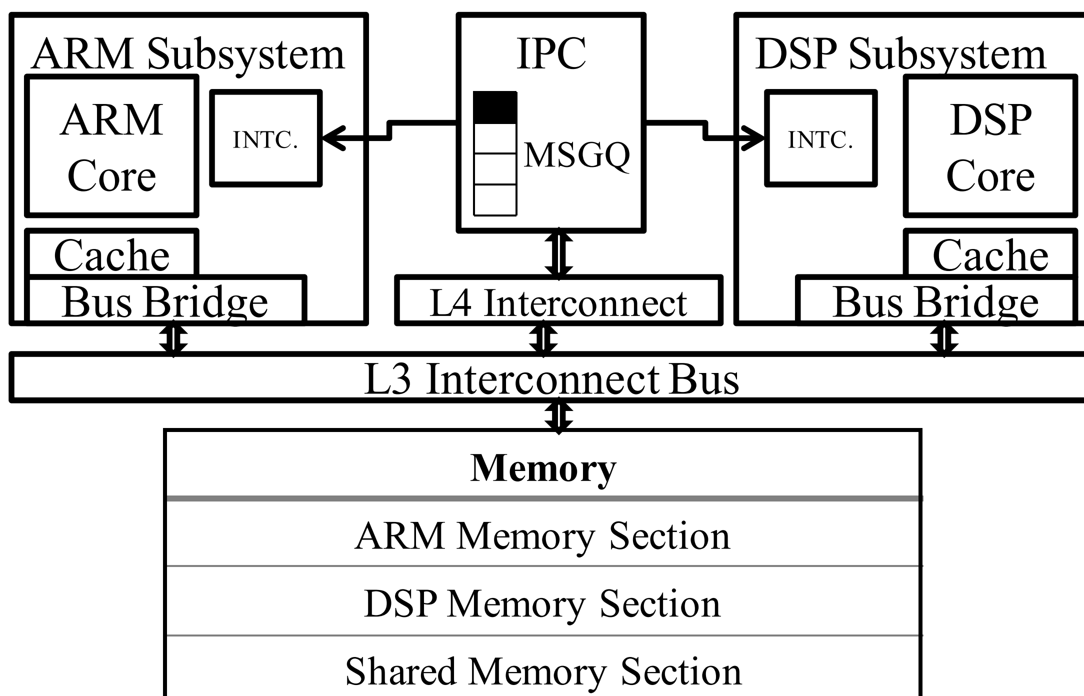

Figure 6. Interprocessor Communication on OMAP3530

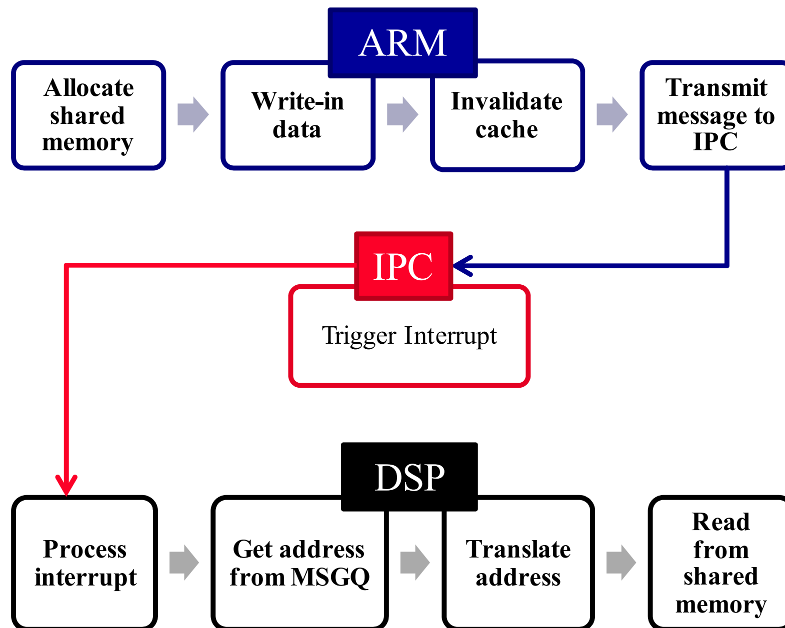

Figure 7. Interprocessor Communication Sequence

to each processor or reserved for interprocessor communication.

Typical sequence of operations to transmit data from one core to the other is shown in Figure 7. The procedure requires that the sending processor prepares the shared memory region, addresses cache coherency issues, transmits a message to the IPC queue, and instructs the IPC to trigger an interrupt for the reading processor. The reading processor processes the interrupt, translates the transmitted addresses to its own memory mapping, and reads the data as necessary. The sequence must be repeated in reverse direction to transmit a reply. It is possible to instantiate multiple communications channels between the ARM and the DSP, though this does not affect the overall communication bandwidth.

Even from this brief description it should be apparent that interprocessor communication between ARM and DSP is quite complicated. Moreover, we have completely omitted the fact that this entire process must be handled through the operating system: the Linux kernel on the ARM and DSPBIOS on the DSP. Handling cache coherency from the operating system requires additional functionality in the kernel, typically provided by a board support library package. Virtual addressing for each processor's possibly unique memory map must also be handled differently based on the

operating system. Essentially, the clear message regarding interprocessor communication on the OMAP system is that if starting from scratch the work becomes a separate project.

Luckily, TI provides a DSPLink library that provides the needed functionality [19]. DSPLink is built for a given target platform and operating system. On the ARM side in Linux the library creates a kernel module that must be loaded at boot or using `insmod` command prior to the execution of DSP-using application. On the DSP side, DSPLink closely integrates with the DSPBIOS and only requires that the DSP-compiled code be linked against the DSP-side of DSPLink library.

We profiled the DSPLink library, and were initially surprised by the seemingly poor performance. A single round trip message of 1 KB in size using our first build of DSPLink was taking over 300 μs. Given the clock speed of the processors, the communication was taking up tens of thousands of clock cycles. We observed that this latency could vary depending on toolchain version and compilation options of the libraries, and message sizes. However, even for the smallest message sizes, and with all debug related compilation options disabled, we never observed the round trip delay time significantly under ~150 μs. We also investigated another older library with similar functionality DSPBridge [20], and looked at precompiled versions of the libraries provided by TI. All the results that we observed were fairly consistent: only with advanced debug functionalities, such as debug outputs to the kernel or traced execution, did we observe a significant increase in communication overhead. Otherwise, no configuration providing any drastic improvement in communication time was found.

Table V. Round-trip Message Latency for ARM to DSP Communication

Library	Message Size	Round-Trip Delay (μs)
DSPLink	128 bytes	153
	1024 bytes	300
DSPBridge	128 bytes	168
	1024 bytes	254

After this analysis we concluded that the ~150 μs latency even for small messages is expected in the OMAP3530 system. The profiling results under final compilation configuration settings for each library are shown in Table V. In our design we ultimately

used the newer DSPLink library due its ease of use, and the hope that our research might be more relevant to the current BeagleBoard development community.

4.3 Rethinking Design Partitioning

The results from interprocessor communication profiling necessitated a change in our multicore partitioning. The results regarding the performance of field multiplication implied that even the largest field multiplication on the DSP would execute in well under 20 μs [1, 4]. Attaching 150 μs overhead to each field multiplication would make the entire multicore design completely impractical and useless.

Table VI. Number of Function Calls for ECC Operations.

	EC Crypto Operations		**EC Point Multiplication**		**EC Point Double/Add**	
	Min	Max	Min	Max	Min	Max
Number of Function Calls	1	2	$1.4n$	$2n$	$15n$	$22n$

Table VI presents the approximate number of function calls at each stage of the ECC pyramid, where n is the size of the underlying field characteristic in bits. The numbers are presented as ranges since the field type and the particular algorithm choice determines the exact number of underlying calls. For instance, an EC Crypto Operation like public/private key pair generation would require just 1 call to a point multiplication function. The point multiplication would require $1.4n$ to $2n$ calls to point addition and point doubling (so up to 320 function calls for a 160-bit field). For field operations, we counted only the number of field multiplications, neglecting the field additions. In choosing the new partitioning strategy we had to determine where in this table could we afford to insert the 150 μs overhead.

In the end, the choice was essentially made for us. Given the large overhead, the only acceptable place to impose the 150 μs overhead and hope to achieve a performance gain would be inside the EC Crypto Operations, or in other words at the calls to point multiplication. Figure 8 presents the modified multicore partitioning design based on this new observation. Under this design the ECC implementation on the ARM would be responsible for EC parameters and the finite field, and would then issue the point

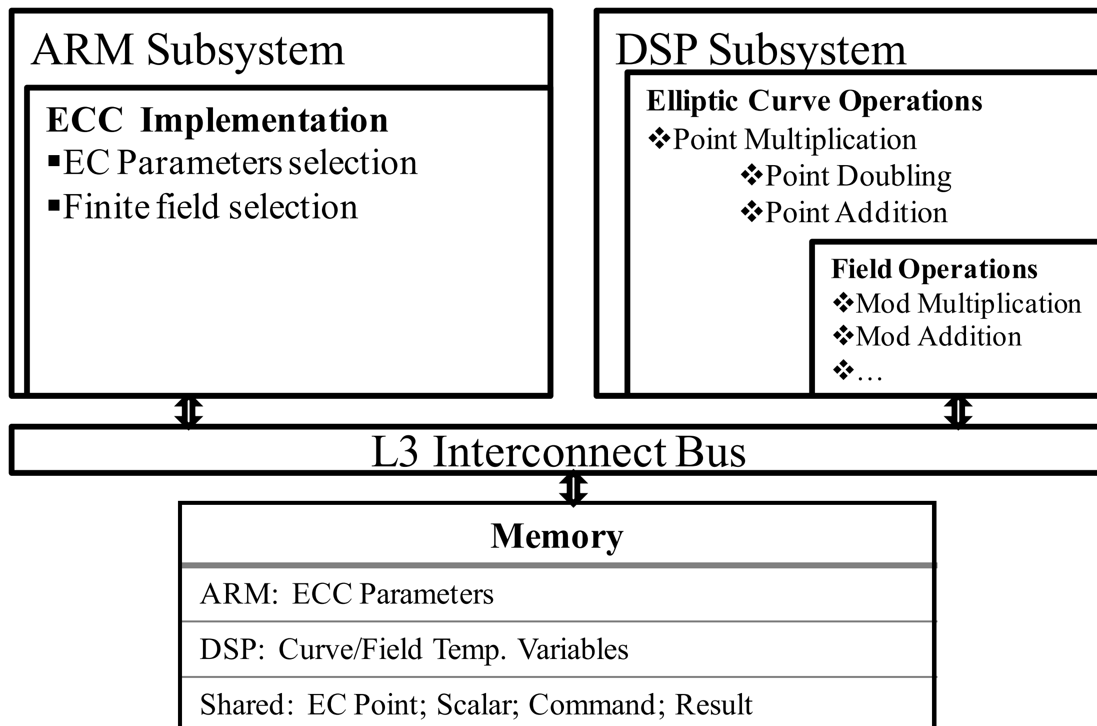

Figure 8. Modified ECC Multicore Partitioning

multiplication call to the DSP by transmitting the starting point P, scalar value k, command parameters (field type and size) to the DSP. The DSP would execute the functions to compute all the elliptic curve operations and all the required field operations. After computing $Q = k \cdot P$, the DSP would transmit the Q back to the ARM.

4.4 ECC Algorithm Implementations

One of the objectives of this work was to demonstrate the benefits of multicore ECC design over conventional approaches. Prior to this research, if an OMAP3530 developer needed ECC functionality he would likely look for a compatible library that would satisfy his requirements. As mentioned in Chapter 2, that developer would likely select OpenSSL, due to its performance and compatibility with the ARM architecture. To create a fair comparison between our multicore implementation and OpenSSL ARM-only implementation, we studied the EC algorithms used by OpenSSL. We were able to identify the exact algorithms being used by OpenSSL by comparing them to published algorithms in ECC related literature. In some cases we made modifications to the

OpenSSL source code to implement identical algorithms in both implementations, even if it involved removing some optimization (such as precomputation for large sized curves). We observe that the underlying field operations in OpenSSL were left unchanged and implemented with all available performance optimization options. Our rationale for this method of implementation was that as long as the underlying field arithmetic was executing optimally on the given processor, the comparison between identical point operation algorithms would provide us with the most useful result. Any EC algorithm-level optimizations based on field size for instance, could be applicable to both types of implementations.

The exact algorithms for point multiplication, doubling, and addition on prime and binary fields are presented in Appendix A. Table VII lists the final configurations of our prime and binary field implementation and cites the sources of the algorithms.

Table VII. Algorithms for our ECC Implementation

Design Parameter	Implementation	
	Prime Field EC	**Binary Field EC**
Elliptic Curves and Parameters	sect163r1, sect283r1, sect409r1	secp160r1, secp224r1
Coordinate System	Lopez-Dahab projective	Jacobian projective
Point Multiplication Algorithm	Algorithm 2P [21]	Algorithm 3.31 in [3]
Point Doubling Algorithm	Mdouble Algorithm [21]	Algorithm 3.21 in [3]
Point Addition Algorithm	Madd Algorithm [21]	Algorithm 3.22 in [3]

4.5 Development Environment

The development environment for the OMAP3530 is very complex, to the point where you could describe it as one of the main difficulties for this kind of multicore development. As we observed while profiling DSPLink, everything from the compilation options of the library itself to the version of the Linux kernel can affect the final results.

Developers wishing to reproduce the results presented in this work should attempt to match the build environment closely.

The most active and used development environment for the BeagleBoard is the OpenEmbedded framework. OpenEmbedded uses a large set of compilation utilities to build everything from the gcc compiler to the Linux kernel itself for a target platform. We used OpenEmbedded to compile the gcc compiler, the Angstrom Linux kernel, and DSPBIOS and C6x compilers for the DSP. Though OpenEmbedded can build a version of DSPLink, we used a newer version (1.61.04) directly from TI and compiled it with OpenEmbedded generated toolchain.

When compiling for the ARM on the BeagleBoard we compiled the libraries with the highest optimization options (-O3 and others as appropriate), and provided the appropriate target platform for the compiler. On the DSP we likewise used -O3 for TI's compiler, and relied on provided makefiles for externally obtained code.

We note that debugging of DSP applications in particular is very challenging from the ARM side. Standard debuggers are unavailable, and even printf debugging functionality has to be improvised using DSPLink. The alternative used for this project was to test the functionality of the code on software emulated DSP core provided in Code Composer Studio development environment.

CHAPTER 5

RESULTS

In this section we present the performance evaluation of our multicore ARM+DSP point multiplication implementation. We compare our implementation with the optimally compiled OpenSLL library for the ARM core on the same platform. We also discuss the power efficiency of the two implementations. We identify TinyECC on Imote2 as the most appropriate comparison target from outside research, and compare our results to it as well.

5.1 Field Multiplication

As mentioned, the performance of field multiplication is critical to the performance of EC point multiplication. Given the fact that we are using identical EC algorithms in our ARM+DSP and OpenSSL ARM-only implementations, whatever speed-up we develop must originate at field operations.

Table VIII shows the execution time of a single field multiplication, accounting for the slower DSP clock frequency relative to the ARM. These results are measured by timing 10000 random field multiplications, including the reduction part. The results are used as operands in the following iteration of the loop. This also emulates field multiplication usage during ECC operations. The communication overhead between DSP and ARM is subtracted out to obtain these results. Given the fact that field multiplication is the absolutely dominant operation, we can use this resulting improvement as our "ideal" speed-up, and compare it to the results that we observe for point multiplication.

We used the implementation from [4] for binary field multiplication and [1] for prime field multiplication and ported it to our ECC project. Our results were mostly comparable to original reported values when platform differences are accounted for. The $GF(2^{409})$ result for field multiplication was significantly faster than expected from results in [4]. This development also contradicted the claim that the relative speed-up decreases

with larger field size. We suspect that either the original result was misreported, or that a newer version of DSP compiler produced more optimized implementation.

Table VIII. Field Multiplication on OMAP3530

Finite Field	500MHz ARM	360MHz TI C64x+ DSP	Speedup
	Time (μs)		
$GF(2^{163})$	7.32	0.97	7.55
$GF(2^{283})$	17.4	2.35	7.4
$GF(2^{409})$	30.4	3.29	9.24
GF(p160)	3.34	0.83	4.02
GF(p224)	7.81	1.57	4.97

The results show that DSP's datapath is significantly more efficient than the ARM at modular multiplication. The observed speed-up for prime field multiplication is noticeably lower than binary field. This occurs because the multiplier on the ARM is able to execute regular multiplication reasonably well, though not as well as the DSP. When it comes to binary field, the lack of hardware carry-less multiplication on the ARM allows the DSP to take a larger lead.

Overall the performance of field multiplication shows that there is at the least potential improvement to be gained by utilizing the DSP.

5.2 Point Multiplication

Table IX presents the execution times of a single random point multiplication using OpenSSL ARM-Only implementation and our ARM+DSP multicore implementation. The scalar k for point multiplication, while randomly selected, is constant between the OpenSSL based implementation and the ARM+DSP implementation for a given curve. The results are presented graphically in Figure 9.

This measurement does include all the overhead associated with point multiplication for the given design. In the case of the ARM+DSP design, we measure the time from just prior to sending the operands to DSP with DSPLink, to right after the

ARM receives the reply containing the computed point while waiting on the reply via DSPLink. In the case of OpenSSL ARM implementation, we measure the time for the point multiplication function to return. This choice was motivated by the desire to show expected speed-up in a real application, one that would have to account for ARM to DSP communication.

Table IX. Point Multiplication on OMAP3530

ECC Curve	ARM Only	ARM+DSP	Speed-up
	Time (μs)		
sect163r1	9674	2106	4.59
sect283r1	35034	7965	4.4
sect409r1	82611	16083	5.14
secp160r1	8700	2929	2.97
secp224r1	15609	6043	2.58

Because we used the alternate partitioning and use only a single round-trip message in the design, communication is not a large factor in the final execution times. In fact, the 150 μs overhead is essentially negligible in the larger field designs. However,

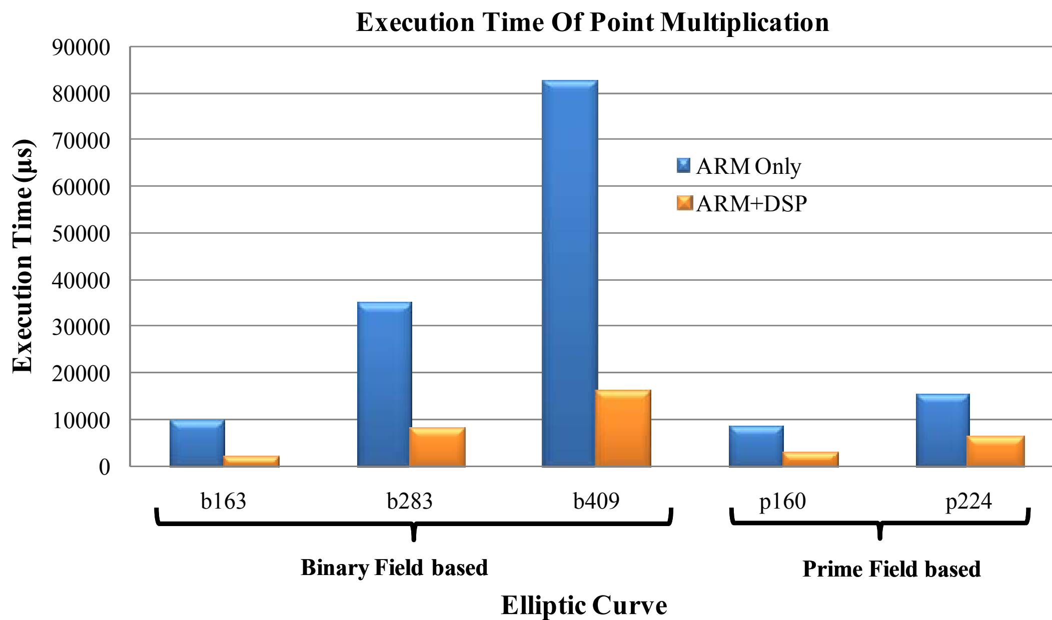

Figure 9. Execution Time Comparison of Point Multiplication on OMAP3530

the speed-up demonstrated with point multiplication shows a considerable drop when compared to the speed-up seen in the field multiplication by itself. We lose between a factor of 1.35 to 1.93 when comparing point multiplication speed-up to field multiplication. We hypothesize that the performance loss occurs because the DSP cannot achieve high datapath utilization during “utility” tasks, like fetching operands from memory, and creating and zeroing out temporary variables, and thus loses some lead over the 500 MHz ARM.

We observe that as expected from field multiplication results, the speed-up for prime field ECC is lower than binary field. However, even considering the least improved result, we compute a point multiplication with our design more than 2.5 times faster than with OpenSSL’s implementation.

5.3 Power Efficiency

BeagleBoard’s PCB design includes a test point to measure to allow for power measurement. Two exposed pins allow us to measure the voltage drop across a 0.3 ohm resistor. The resistor is located right between the board’s power supply and the primary

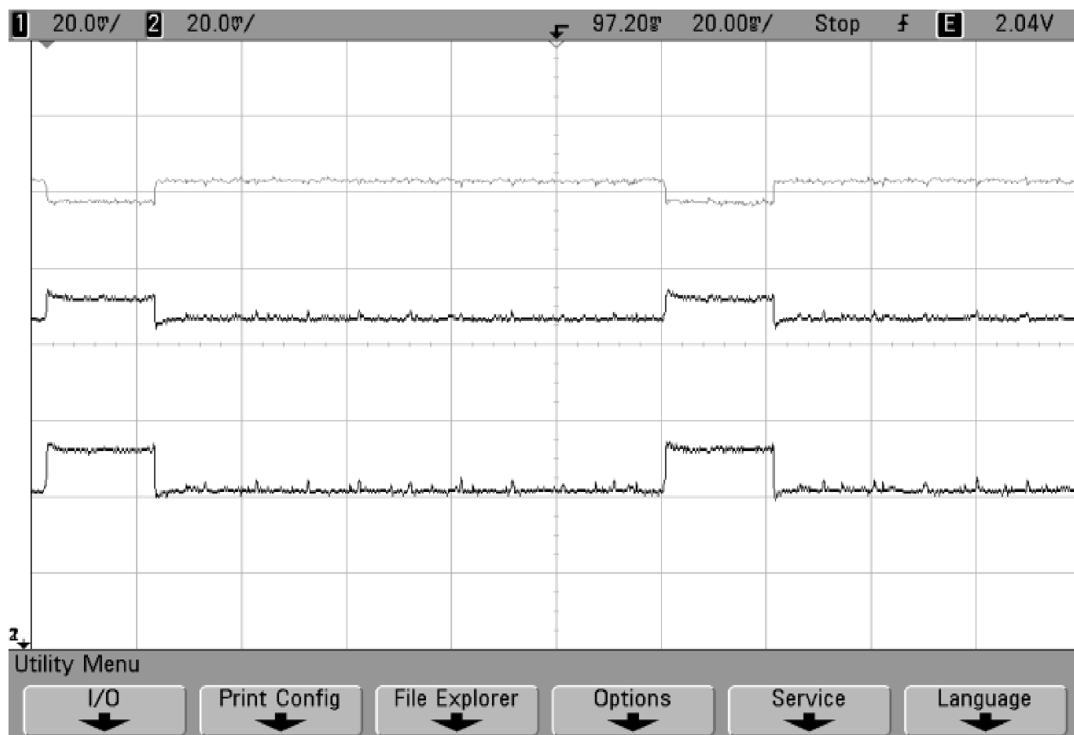

Figure 10. Power Trace of Point Multiplication with ARM-Only Version

power regulator chip TPS65950, with the OMAP3530 chip located on the next step in the power chain.

To measure the power we developed the capability to send a trigger signal to one of the open GPIO pins on the beagle board. We measured the power by using this trigger to capture the voltage drop across the resistor with an oscilloscope. For power measurement purposes we execute 10 identical point multiplications in a row and zero out the results between each. We then sleep() the processor for some time in order to help us identify start and stop points in the power traces. Figure 10 shows the voltage trace as captured by the oscilloscope for 10 point multiplications in sect163r1 using OpenSSL ARM-Only implementation with a single division representing 20 millivolts vertically and 20 milliseconds horizontally. Figure 11 shows the same 10 sect163r1 point multiplications for ARM+DSP implementation with a single division representing 20 millivolts vertically and 10 milliseconds horizontally. On both figures, the top light-shaded line represent the voltage drop across the resistor, while 2nd and 3rd lines are the voltages at the two end points of the resistor.

Figure 11. Power Trace of Point Multiplication with ARM+DSP Version

By computing the current through the resistor and multiplying that current by the power supply voltage we were able to get a precise estimate of the power draw by the entire board. We motivate the use of power numbers for the entire board by the fact that the OMAP chip by itself does not represent a complete system. By analyzing the two traces we can determine the power during the idle section of our program (with ARM executing a `sleep()` instruction), during execution of point multiplication with ARM-Only implementation, and during execution of ARM+DSP implementation. The results for sect163r1 traces are presented in Table X.

Table X. Power Characteristics of OMAP3530 during Point Multiplication

Mode	Current (Amps)	Power (Watts)	Power Relative to Idle
ARM Idle	0.132	0.676	1
ARM-Only	0.15	0.771	1.141
ARM+DSP	0.155	0.795	1.177

We observe a 14% increase in the power draw of the device during ARM-Only implementation execution relative to idle. With ARM+DSP implementation, with the ARM waiting for DSP to reply, the power draw is increased by 17% relative to idle. We note that we did not attempt to use any kind of power saving methods on the ARM while it is waiting for the reply from the DSP. For instance, if we were to dynamically reduce ARM's clock frequency during the waiting period it may be possible to achieve lower overall power draw without affecting performance.

By multiplying the power consumption during execution with the execution time, we are able to compare the energy cost associated with doing a single point multiplication on the BeagleBoard using OpenSSL ARM-Only implementation and our ARM+DSP implementation. The results are presented in Table I and pictured in Figure 12.

Even with the increased power due to DSP utilization, we save considerable energy to per point multiplication with the ARM+DSP scheme once we take the execution time into account. While this method for measuring the power does not let us

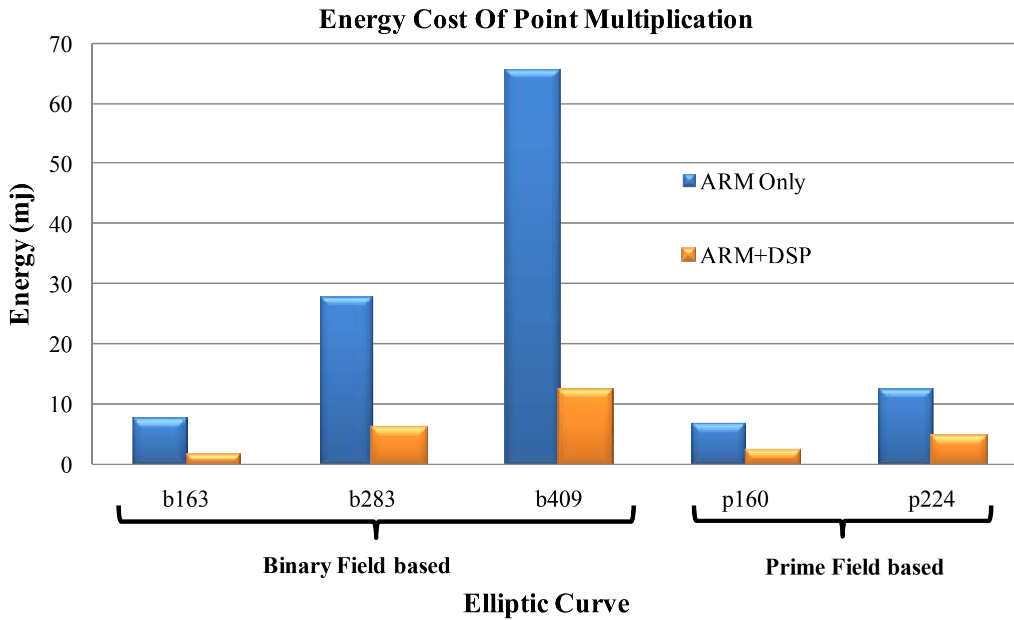

Figure 12. Energy Cost Comparison of Point Multiplication on OMAP3530

isolate the power used by OMAP3530 chip or a particular subsystem, the platform level is appropriate given our original goals.

Table XI. Energy Cost of Point Multiplication on OMAP3530

ECC Curve	ARM Only	ARM+DSP	Energy Savings Factor
	Energy (millijoules)		
sect163r1	7.69	1.62	4.75
sect283r1	27.87	6.14	4.54
sect409r1	65.71	12.4	5.3
secp160r1	6.92	2.26	3.06
secp224r1	12.42	4.66	2.67

5.4 TinyECC Comparison

The unique nature of our implementation made finding outside data for comparison challenging. FPGA based co-processor implementations were not appropriate given the differences in the platforms. Microcontroller based implementations often targeted 8-bit platforms running at 10 MHz or less, which were

likewise not comparable to our 500/360 MHz OMAP 3530. Additionally we designed our implementation to run as a service for the operating system, while many ECC implementations tend to ignore that overhead. The best candidate for comparison that we found was the TinyECC library running on the 32-bit Imote2 TinyOS based platform [7].

TinyECC implemented only the support for prime field secp160r1 curve, however we chose to include the binary field sect163r1 in our comparison as well. The reason for this was that we observed noticeably better performance in our binary field curves, and replacing a prime field curve with similar sized binary field curve would not affect the security of an application. Table XII presents the comparison for execution time and energy cost of a single point multiplication.

Table XII. Comparison of Point Multiplication on OMAP3530 and Imote2

	Implementation	CPU (MHz)	Exec. Time (ms)	Energy Cost (mJ)
Ours	ARM-Only sect163r1	500	9.67	6.14
	ARM+DSP sect163r1	360	2.11	1.62
	ARM-Only secp160r1	500	8.7	6.92
	ARM+DSP secp160r1	360	2.93	2.26
[7]	TinyECC secp160r1	104	43.62	2.73
		416	11.25	-

We note that the Imote2 platform could be configured to run at 104 or 416 MHz. At 416 MHz it was the highest performing TinyECC platform relative to other TinyOS based platforms, while at 104 MHz it was the most power efficient. The power consumption in 416 MHz configuration was not reported, but was implied to be higher. We also note that the energy consumption reported for Imote2 TinyECC implementation is based on datasheet estimates, rather than power traces [7].

At 416 MHz the execution time of Imote2 implementation is comparable to the execution of OpenSSL based implementation on our platform, especially when accounting for the difference in clock speed. When we compare it to our ARM+DSP design we notice a clear performance gain. The performance gain is large enough to

allow the ARM+DSP design to save energy on per-multiplication basis, even compared with the most power efficient setting of Imote2 at 104 MHz. Overall our ARM+DSP implementation, especially the binary field version, compares favorably to TinyECC's Imote2 in all aspects.

CHAPTER 6

FUTURE WORK

As with any research, this work has multiple possible avenues for advancement and improvement. In this case, future research could focus on performance or efficiency improvement, functionality expansion of our ECC framework, or extending the presented co-design based approach other platform.

6.1 Performance and Efficiency Improvement

We have shown that speed-up in the underlying field operations translates into point multiplication in our ECC framework. Though a factor of that speed-up is lost when we build up the entire ECC framework, performance of field multiplication in particular is critical to the final speed of point multiplication. Tergino identified Karatsuba multiplication method as a possible way to improve the performance of modular multiplication in large binary finite fields [4], and it is likely that this would improve the performance of ECC as well.

As noted this work largely neglected the methods for optimizations of EC point operations, for the purpose of creating a fair comparison between ARM-Only and ARM-DSP implementation. These optimizations, particularly those involving precomputation of points, are known and described in [3]. Our hypothesis is that these optimizations should be applicable to ARM-Only and ARM+DSP designs more or less equally, but the optimal point multiplication algorithms for OMAP3530 are still out there.

As with Imote2 platform, the OMAP3530's clock speed for DSP and ARM are configurable. This work did not attempt to find the optimal clock frequency for the energy efficiency of point multiplication. Other possibilities for better energy efficiency include investigating additional power saving modes for the ARM core while the execution is offloaded to the DSP.

6.2 ECC Functionality Extension

This worked has focused on improving the performance of point multiplication for ECC, and has skipped the implementation of the actual cryptographic services that make use of it. Our implementation is presently limited to being able to generate public and private key pairs for ECC. Algorithms like Elliptic Curve Digital Signature Algorithm (ECDSA) or the key-agreement algorithms like ECMQV are still need to perform the actual cryptographic services.

One possibility would be to continue building on top of our ARM-DSP project to implement such algorithms. Another more attractive possibility would be to integrate our point multiplication scheme into OpenSSL library directly, as a compile time optimization for the library. On the appropriate platform OpenSSL's point multiplication call would go out to the DSP in order to free up the ARM and improve the efficiency of the application.

6.3 Extending Co-design Approach

It is interesting to note that while our original design partitioning proved impractical due to high communication latency, had the communication been 10 or even 50 times faster, it would not open up the design space of multicore ECC. An overhead of 15 or 3 μs on each field arithmetic operation would still be just as impractical.

Instead our approach has opened up another co-design possibility: the ARM core is currently idling during the DSP-based point multiplication computation. Presumably an actual ECC using system would be able to find other tasks to schedule on the ARM during this idle time. A prototype of such functionality would likely show much higher performance improvement than we demonstrated with our ARM+DSP point multiplication.

CHAPTER 7

CONCLUSION

Our objective in this research was to study heterogeneous multicore ECC design on a current-generation SoC platform. We implemented the key component of ECC on TI's OMAP3530 heterogeneous multicore platform. Our work was based techniques and implementations on related non-multicore designs, which improved performance by closely mapping the underlying algorithms to a datapath. Our results have shown that there are clear and tangible performance and power efficiency benefits to extending such work to multicore platforms. Though platform constraints can limit the design space for a true co-design implementation, using appropriate partitioning it is still possible to achieve considerable speed-up over non-multicore implementations.

While custom hardware coprocessors have sometimes promised larger potential improvements, our research shows that existing SoC systems already have the acceleration tools embedded inside them, ready to be used for performance intensive applications.

We note that the design process for this type of multicore embedded system is still daunting. Achieving underlying speed-up required using platform specific code for the DSP. Multicore partitioning required profiling the interprocessor communication, where large communication latency limited our design space. The large number of ECC design parameters and algorithms prevented us from exhaustively testing all possible implementations to find the optimal version. Moreover, the complex development environment extended the duration of this research far beyond the original timelines.

The performance and energy efficiency improvement from this kind of co-design is certainly attainable, but aspiring co-designers should expect to invest a thesis-worth of effort.

BIBLIOGRAPHY

[1] H. Yan, *et al.*, "Efficient Implementation of Elliptic Curve Cryptography on DSP for Underwater Sensor Networks " in *Workshop on Optimizations for DSP and Embedded Systems (ODES-7)*, 2009, pp. 10-15.

[2] S. C. Shantz, "From Euclid's GCD to Montgomery Multiplication to the Great Divide," Sun Microsystems, Inc., Tech Rep. TR-2001-95. June 2001.

[3] D. Hankerson, *et al.*, *Guide to Elliptic Curve Cryptography*. New York: Springer, 2004.

[4] C. S. Tergino, "Efficient Binary Field Multiplication on a VLIW DSP," M.S. thesis, Dept. Elect. and Comp. Eng., Virginia Polytechnic Inst. and State Univ., Blacksburg, VA, 2009.

[5] National Institute of Standards and Technology, "Recommended Elliptic Curves for Federal Government Use," 1999.

[6] Standards for Efficient Cryptography Group, "SEC 2: Recommended Elliptic Curve Domain Parameters," Certicom Corp., 2000.

[7] A. Liu and P. Ning, "TinyECC: A Configurable Library for Elliptic Curve Cryptography in Wireless Sensor Networks," in *7th Int. Conf. Information Processing in Sensor Networks (IPSN 2008)*, 2008, pp. 245-256.

[8] S. Madhavapeddy and B. Carlson, "OMAP™ 3 architecture from Texas Instruments opens new horizons for mobile Internet devices," white paper, Texas Instruments, Aug. 2008.

[9] *PIXHAWK: Open Source Micro Air Vehicle Computer Vision at ETH Zurich*. [Online]. Available: http://pixhawk.ethz.ch/start

[10] V. Gupta, *et al.*, "Performance analysis of elliptic curve cryptography for SSL," in *Proc. 1st ACM Workshop on Wireless Security*, 2002, pp. 87-94.

[11] L. Elbaz, "Using Public Key Cryptography in Mobile Phones," white paper, Discretix Technologies Ltd., 2002.

[12] N. B. Anuar, *et al.*, "Mobile messaging using public key infrastructure: m-PKI," in *Proc. 12th WSEAS Int. Conf. on Computers*, Heraklion, Greece, 2008.

[13] S. Tillich and J. Großschädl, "A Survey of Public-Key Cryptography on J2ME-Enabled Mobile Devices," in *Computer and Information Sciences - ISCIS 2004*.

vol. 3280, C. Aykanat*, et al.*, Eds., ed: Springer Berlin / Heidelberg, 2004, pp. 935-944.

[14] H. Alrimeih and D. Rakhmatov, "Security-Performance Trade-offs in Embedded Systems Using Flexible ECC Hardware," *IEEE Des. Test,* vol. 24, pp. 556-569, 2007.

[15] G. Xu and P. Schaumont, "Optimizing the HW/SW boundary of an ECC SoC design using control hierarchy and distributed storage," in *Design, Automation & Test in Europe Conference & Exhibition, 2009. (DATE '09.)*, 2009, pp. 454-459.

[16] *OpenSSL: The Open Source tookit for SSL/TLS*. [Online]. Available: http://www.openssl.org/

[17] G. K. Abusharekh A., "Comparative Analysis of Software Libraries for Public Key Cryptography," *ECRYPT Workshop on Software Performance Enhancement for Encryption and Decryption,* 2007, pp. 1-16.

[18] *Beagleboard.org - brief*. [Online]. Available: http://beagleboard.org/brief

[19] "DSP/BIOS Link Release Notes," Tech. Note, Texas Instruments, LNK 111 REL, Nov. 13, 2009.

[20] *DSP Bridge - OMAPpedia*. [Online]. Available: http://omappedia.org/wiki/DSPBridge_Project

[21] J. López and R. Dahab, "Fast Multiplication on Elliptic Curves Over GF (2m) without precomputation," in *Proc. 1st Int. Workshop on Cryptographic Hardware and Embedded Systems* , 1999, pp. 724-724.

Appendix A

FORMULAS FOR EC POINT OPERATIONS

The following figures present the algorithms used for prime field elliptic curve point multiplication. The algorithm for point multiplication in a prime field EC curve is shown in Figure A.1. The algorithm requires that the scalar value *k* be converted to non-adjacent form (NAF). NAF is a representation of an integer using elements from $[0, \pm 1]$ and contains no two consecutive nonzero digits. Formula for obtaining NAF is discussed in [3].

Prime Field EC Point Multiplication

```
Input: integer k>0, EC point P =(x1,y1)
Output: Q = kP
   1. Compute NAF(k)
   2. k:=(k_{l-1}…k_1k_0)_2
   3. X3,Y3,Z3 :=(0,0,1)
   4. For i from l-1 downto 0 do
            PointDouble(X3,Y3,Z3)
            If ki== 1 then
                PointAdd(X3,Y3,Z3, x1, y1)
            If ki==-1 then
                PointAdd(X3,Y3,Z3, x1,-y1)
   5. Return Q:=Convert(X3,Y3,Z3)
```

Figure A.1. Algorithm for Point Multiplication on Prime Field EC

The point doubling algorithm for elliptic curves over prime fields shown in Figure A.2 is a slight reordering of Algorithm 3.21 in [3]. The inputs and outputs are in Jacobian projective coordinates.

PointDouble() Function for Prime Field EC Point Multiplication

```
Input: X1,Y1,Z1 coordinates of point P1
Output: X1,Y1,Z1 coordinates of point 2*P1
    1.  If X1==0 and Y1==0 return(0,0,1)
    2.  T1:= Z1^2
    3.  T2:= X1-T1
    4.  T1:= X1+T1
    5.  T2:= T2*T1
    6.  T2:= 3T2
    7.  Y1:= 2*Y1
    8.  Z1:= Y3*Z1
    9.  Y1:= Y1^2
    10.  T3:= Y1*X1
    11.  Y1:= Y1^2
    12.  Y1:= Y1/2
    13.  X1:= T3^2
    14.  T1:= 2*T3
    15.  X1:= X1-T1
    16.  T1:= T3-X3
    17.  T1:= T1*T2
    18.  Y1:= T1-Y1
```

Figure A.2. PointDouble() Function

The point addition algorithm for elliptic curves over prime fields shown in Figure A.3 is a slight variation of Algorithm 3.22 in [3]. One of the input points is in Jacobian projective coordinates while the other is in affine coordinates. This is known as mixed addition.

PointAdd() Function for Prime Field EC Point Multiplication

Input: X_1, Y_1, Z_1 coordinates of point P; x_2, y_2 coordinates of point Q;
Output: X_1, Y_1, Z_1 coordinates of point P+Q

1. If $x_2 == 0$ and $y_2 == 0$ return (X_1, Y_1, Z_1)
2. If $X_1 == 0$ and $Y_1 == 0$ return $(x_2, y_2, 1)$
3. If $X_1 == x_2$ and $Y_1 == y_2$ return PointDouble(X_1, Y_1, Z_1)
4. $T_1 := Z_1^2$
5. $T_2 := T_1 * Z_1$
6. $T_1 := T_1 * x_2$
7. $T_2 := T_2 * y_2$
8. $T_1 := T_1 - X_1$
9. $T_2 := T_2 - Y_1$
10. $Z_1 := Z_1 * T_1$
11. $T_3 := T_1^2$
12. $T_4 := T_3 * T_1$
13. $T_3 := T_3 * X_1$
14. $T_1 := 2 * T_3$
15. $X_1 := T_2^2$
16. $X_1 := X_1 - T_1$
17. $X_1 := X_1 - T_4$
18. $T_3 := T_3 - X_3$
19. $T_3 := T_3 * T_2$
20. $T_4 := T_4 * Y_1$
21. $Y_1 := T_3 - T_4$

Figure A.3. PointAdd() Function

The point multiplication algorithm for elliptic curves over binary fields shown in Figure A.4 is based on the algorithm presented in [21]. The algorithm uses Lopez-Dahab projective coordinates for point representation. The algorithm ignores the *Y* coordinate during the intermediate steps of point multiplication, and instead keeps track of two related elliptic curve points. The *Y* coordinate of point *Q* is calculated from the two points at the last step.

Binary Field EC Point Multiplication

```
Input: integer k>0, EC point P =(x,y)
Output: Q = kP
   1. if k == 0 or x == 0 return Q :=(0,0)
   2. k:=(k_{l-1}…k_1k_0)_2
   3. X1:=x; Z1:=1; X2:=x^4+b; Z2:=x^2
   4. For i from l-2 downto 0 do
          if k_i == 1 then
               Madd(X1, Z1, X2, Z2)
               Mdouble(X2, Z2)
          else
               Madd(X2, Z2, X1, Z1)
               Mdouble(X1, Z1)
   5. Return Q:= Convert(X1, Z1, X2, Z2))
```

Figure A.4. Algorithm for Point Multiplication on Binary Field EC

The algorithm for point doubling in a binary field is shown in Figure A.5 and is a slight reordering of the *Mdouble()* algorithm presented in [21] . As noted, due to the choice of point multiplication algorithm the *Y* coordinate can be ignored during the calculation.

Mdouble() Function for Binary Field EC Point Multiplication

Input: X_1, Z_1 coordinates of point P, EC parameter b
Output: X_1, Z_1 coordinates of 2*P

1. $T_1 := X_1^2$
2. $T_2 := Z_1^2$
3. $Z_1 := T_1 * T_2$
4. $X_1 := T_1^2$
5. $T_1 := T_2^2$
6. $T_2 := b * T_1$
7. $X_1 := X_1 + T_1$

Figure A.5. Mdouble() Function

The algorithm for point addition in binary field is shown in Figure A.6 is a based on *Madd()* algorithm in [21]. Again observe that the algorithm ignores the *Y* coordinate. The algorithm requires executing a field addition with the *x* coordinate of the original base point of point multiplication.

Madd() Function for Binary Field EC Point Multiplication

Input: X_1, Z_1 coordinates of point P_1; X_2, Z_2 coordinates of point Q_2; x coordinate of original point P

Output: X_1, Z_1 coordinates of point P_1+Q_2

1. $T_1 := X_1 * Z_2$
2. $T_2 := X_2 * Z_1$
3. $X_1 := T_2 * T_1$
4. $T_2 := T_1 + T_2$
5. $Z_1 := T_2^2$
6. $T_1 = x * Z_1$
7. $X_1 := X_1 + T_1$

Figure A.6. Madd() Function