Combining Algorithm Exploration with Instruction Set Design: A Case Study in Elliptic Curve Cryptography*

Johann Großschädl [1], Paolo Ienne [2], Laura Pozzi [2], Stefan Tillich [1], and Ajay K. Verma [2]

[1] Graz University of Technology
Institute for Applied Information Processing
Inffeldgasse 16a, A–8010 Graz, Austria
{johann.groszschaedl,stefan.tillich}@iaik.at

[2] Ecole Polytechnique Fédérale de Lausanne
School of Computer and Communication Sciences
CH–1015 Lausanne, Switzerland
{paolo.ienne,laura.pozzi,ajay.verma}@epfl.ch

Abstract

In recent years, processor customization has matured to become a trusted way of achieving high performance with limited cost/energy in embedded applications. In particular, Instruction Set Extensions (ISEs) have been proven very effective in many cases. A large body of work exists today on creating tools that can select efficient ISEs given an application source code: ISE automation is crucial for increasing the productivity of design teams. In this paper we show that an additional motivation for automating the ISE process is to facilitate algorithm exploration: the availability of ISE can have a dramatic impact on the performance of different algorithmic choices to implement identical or equivalent functionality. System designers need fast feedbacks on the ISE-ability of various algorithmic flavors. We use a case study in elliptic curve (EC) cryptography to exemplify the following contributions: (1) ISE can reverse the relative performance of different algorithms for one and the same operation, and (2) automatic ISE, even without predicting speed-ups as precisely as detailed simulation can, is able to show exactly the trends that the designer should follow.

1. Introduction

One of the most successful ways to use processors for complex programmable *System-on-Chips (SoCs)* is to take a basic processor architecture and modify it to better suit the application-domain at hand. This opportunity for customization ranks among the most interesting differences between general-purpose computing—where performance is the all-dominating parameter—and embedded computing—where such a one-fits-all strategy is not optimal. Design goals for embedded SoC are generally more heterogenous and may express a certain level of performance as a design constraint and energy consumption or silicon area as the parameters to minimize. While the design of custom processors and tool chains from scratch has never been very practical, several customizable and extensible processors have appeared on the market with some success [9, 19]. Roughly speaking, these products provide an architectural template (including supporting tool chain), which accepts Instruction Set Extensions (ISEs) in the form of application-specific functional units as shown in Figure 1.

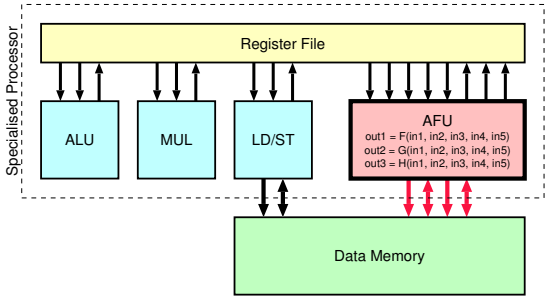

Figure 1. Extensible processor with a five-input three-output application-specific functional unit.

While ISE has been seen so far primarily as an efficient paradigm for embedded application acceleration, and automatic ISE as a useful tool for fast and effective exploration of architectural choices, we claim and show in this paper that an additional motivation to automate the ISE process is to *help algorithm exploration*. In many application domains there exist different algorithmic choices to realize identical or equivalent functionality, e.g., the use of a prime field or a binary extension field for the implementation of an elliptic curve cryptosystem [3]. The effectiveness of these options is typically evaluated on conventional microprocessors, while we claim that the potentiality of ISE should always be borne in mind during algorithm exploration. In fact, we show in this paper that the availability of ISEs can have a dramatic

*The research of Johann Großschädl and Stefan Tillich is supported by the Austrian Science Fund (FWF) under grant number P16952–N04.

3-9810801-0-6/DATE06 © 2006 EDAA

impact on the effectiveness of the different algorithms at choice, and that it may completely redirect the algorithm selection. We also show that state of the art techniques for automatic ISE generation are able to tell exactly the correct trends that the system designer should follow.

The rest of this paper is organized as follows: in Section 2 and 3 we overview elliptic curve (EC) cryptography and its underlying arithmetic operations. In Section 4 we study a number of algorithmic flavors for implementing EC cryptography and demonstrate that ISEs can reverse the relative performance of different algorithms for one and the same arithmetic operation, e.g., multiplication in a finite field. We finally show in Section 5 that an automatic ISE tool can predict correctly the same trends. Section 6 summarizes the main results and concludes the paper.

2. Elliptic Curve Cryptography

Public-key cryptography is an integral part of modern security protocols like SSL or IPSec/IKE [15]. The recent years have seen a growing interest in *elliptic curve (EC) cryptography*, a special variant of public-key cryptography characterized by a good balance between security and performance. Compared to their traditional counterparts like RSA, EC systems can use much shorter keys (in the range of 160–200 bits instead of 1024–2048 bits) to guarantee a reasonable level of security [3].

From a mathematical point of view, EC systems operate in the group of points on an elliptic curve defined over a *finite field* [3]. Several standard bodies recommend to use a *prime field* GF(p) or a *binary extension field* GF(2^m) for the implementation of EC cryptography. The performance of an EC cryptosystem is primarily determined by the efficiency of the arithmetic operations in the underlying finite field, in particular the field multiplication. However, these arithmetic operations are very computation-intensive since the operands have a length of ≥ 160 bits. Moreover, certain arithmetic operations, such as multiplication in binary extension fields, are not very well supported by general-purpose processors. This motivated a number of micro-processor vendors to extend their instruction set architectures with special instructions to facilitate the efficient implementation of field arithmetic; two familiar examples are SmartMIPS [17] and the ARM SecurCore architecture [1].

3. Arithmetic Algorithms

Formally, a *finite field* or *Galois field* can be described as a finite set of elements on which two operations—addition and multiplication—are defined such that the field axioms hold [14]. The elements of a prime field GF(p) are simply the integers from 0 to $p-1$, while a binary field GF(2^m) consists of binary polynomials of degree up to $m-1$.

3.1. Multiplication in Prime Fields

The arithmetic in a prime field GF(p) is the conventional modular arithmetic, i.e., addition and multiplication of field elements (integers) modulo the prime P. However, since the operand length (≥ 160 bits) exceeds the wordsize of the processor, we are forced to represent the operands by arrays of single-precision words, e.g., arrays of 32-bit words.

We will use the following *notation* in this paper: Uppercase letters represent field elements (long integers), while lowercase letters, usually indexed, refer to the individual words of a field element. We denote the bitlength of field elements by n and the processor's wordsize by w. Following this notation, we can write an n-bit integer A as

$$A = \sum_{i=0}^{s-1} a_i \cdot 2^{i \cdot w} = a_{s-1} \cdot 2^{(s-1) \cdot w} + \cdots + a_1 \cdot 2^w + a_0, \quad (1)$$

whereby $s = \lceil n/w \rceil$ is the number of words, and all words a_i are in the range of $0 \leq a_i \leq 2^w - 1$. For example, a 192-bit integer A can be represented by an array of six words on a 32-bit processor, i.e., $A = (a_5, a_4, a_3, a_2, a_1, a_0)$.

The multiplication of elements of a prime field GF(p) is performed in two steps: multiplication of two s-word integers A and B, yielding a $2s$-word product, and reduction of this product modulo the prime P. Most elliptic curve cryptosystems use special primes for which the reduction operation can be accomplished very efficiently [12]. Therefore, the overall execution time of a multiplication in GF(p) is dominated by the long integer multiplication.

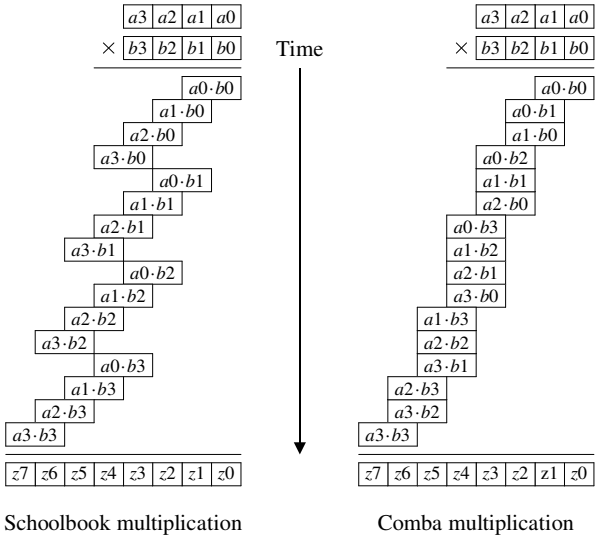

Figure 2. Schoolbook and Comba multiplication.

There exist two different algorithms for implementing a long integer multiplication: the *schoolbook multiplication* [15] and the *Comba multiplication* [7] (see Figure 2). Both

algorithms require to carry out exactly s^2 single-precision multiplications for s-word operands, but they differ in the order in which they process the partial products and in the number of load/store operations. Since both algorithms are well documented in the literature, e.g., in [12], we only summarize their main characteristics in this paper.

The schoolbook multiplication can be implemented according to Algorithm 2.9 in [12]. From an algorithmic point of view, the schoolbook multiplication has a nested loop structure with a relatively simple inner loop that does the bulk of computation. The individual words z of the result $Z = A \cdot B$ are produced in a row-by-row fashion as shown in Figure 2. In any iteration of the inner loop, an operation of the form $z + a \cdot b + u$ is carried out, whereby a, b, z, and u are all w-bit words. The result of $z + a \cdot b + u$ is always a $2w$-bit (i.e., double-precision) quantity, and thus it fits into two w-bit registers.

The inner loop is iterated exactly s^2 times, and hence the number of $(w \times w)$-bit (i.e., single-precision) multiplications is also s^2. Besides the single-precision multiplication and additions, two load operations (for b and z) and one store operation (for z) take place in the inner loop. The word a is loaded in the outer loop and can be kept in a register during the iterations of the inner loop (see [12] for details).

Comba multiplication [7], depicted on the right of Figure 2, forms the product $Z = A \cdot B$ by computing each word z of Z at a time, starting with the least significant word z_0. A formal description of Comba's multiplication method is given in [12, Algorithm 2.10]. The algorithm has a nested loop structure, whereby the inner loop is iterated s^2 times when the operands have a length of s words. However, the main differences to the schoolbook method are the order in which the partial products are generated (see Figure 2), and that the store operations are performed solely in the outer loop. A word z of the result Z is written to memory only after it has been completely calculated. Therefore, Comba's method requires fewer memory accesses than the schoolbook method.

The operation carried out in the inner loop of Comba's method is a "classical" multiply-accumulate operation: two single-precision words a, b are multiplied and the $2w$-bit product $a \cdot b$ is added to a cumulative sum. This cumulative sum is generally larger than $2w$ bits, and therefore it must be held in three w-bit registers. In summary, the inner loop of Comba's method is iterated s^2 times, and each iteration loads two words from memory and performs a MAC operation. The store operations are done in the outer loop.

3.2. Multiplication in Binary Fields

The elements of a binary extension field GF(2^m) are binary polynomials (i.e., polynomials whose coefficients are 0 or 1) of degree up to $m-1$. The addition of two elements of GF(2^m) is simply accomplished by adding the coefficients of the corresponding binary polynomials modulo 2, which is nothing else than a logical XOR operation. On the other hand, the multiplication in GF(2^m) is performed modulo an *irreducible polynomial* $p(t)$ of degree m [14]. In this paper we use indexed Greek letters to denote the coefficients of a binary polynomial, i.e., we write

$$a(t) = \sum_{i=0}^{m-1} \alpha_i \cdot t^i = \alpha_{m-1} \cdot t^{m-1} + \cdots + \alpha_1 \cdot t + \alpha_0 \qquad (2)$$

whereby each of the m coefficients is either 0 or 1. Similar to long integers, we can also store a binary polynomial of degree $m-1$ in an array of $s = \lceil m/w \rceil$ single-precision words. The i-th word a_i of a binary polynomial $a(t)$ contains the w coefficients $\alpha_{i \cdot w}$, $\alpha_{i \cdot w+1}$, ..., $\alpha_{i \cdot w+w-1}$.

Multiplication in GF(2^m) requires multiplying the two field elements (i.e., binary polynomials) together, yielding a binary polynomial of degree $\leq 2m-2$, and then finding the residue modulo the irreducible polynomial $p(t)$, which can be done very efficiently when $p(t)$ is a trinomial or a pentanomial [12]. The standard algorithm for multiplying two binary polynomials is the so-called *Shift-and-XOR* method (shown in Algorithm 1), which is similar to the shift-and-add algorithm for integer multiplication. Algorithm 1 forms the product $z(t) = a(t) \otimes b(t)$ by scanning the coefficients of the multiplier polynomial $b(t)$ from β_{m-1} to β_0 and adding the partial product $a(t) \cdot \beta_i$ to the intermediate result $z(t)$. Before adding $a(t) \cdot \beta_i$, the intermediate result $z(t)$ must be multiplied by t (i.e., left-shifted by 1 bit) to align if for the next partial product. After m steps, $z(t)$ is the product $a(t) \otimes b(t)$.

Algorithm 1. Shift-and-XOR multiplication.

Input: Binary polynomials $a(t)$ and $b(t)$ of degree $m-1$.
Output: Product $z(t) = a(t) \otimes b(t)$ of degree $2m-2$.

1: $z(t) \leftarrow 0$
2: **for** i from $m-1$ by 1 downto 0 **do**
3: $\quad z(t) \leftarrow z(t) \cdot t + a(t) \cdot \beta_i$
4: **end for**
5: **return** $z(t)$

While Algorithm 1 looks quite simple, it must be considered that the polynomials have a very high degree and are stored in arrays of s single-precision words. Therefore, an actual implementation of Algorithm 1 in software results in a nested loop structure, whereby the inner loop is iterated $32s^2$ times if a single word consists of 32 coefficients. Optimized variants of the Shift-and-XOR algorithm, such as the left-to-right comb method [12], use look-up tables to reduce the number of both shift and XOR operations .

Virtually all modern processors provide instructions for a $(w \times w)$-bit multiplication of integers, but not for binary polynomials. A number of researchers proposed to emulate

```
l1: lw    $t0, 0($t1)   # load A[j]
    addiu $t1, $t1, 4   # increment A pointer
    multu $t0, $t4      # multiply A[j] by B[i]
    lw    $t2, 0($t3)   # load Z[k]
    maddu $t5, $t7      # add previous U to product
    maddu $t2, $t7      # add Z[k] to product
    addiu $t3, $t3, 4   # increment Z pointer
    mflo  $t6           # read V
    mfhi  $t5           # read U
    sw    $t6, -4(t3)   # write V to Z[k]
    bne   $t1, $t8, l1  # branch if not end of loop
```

Figure 3. Inner loop of schoolbook multiplication.

this missing instruction in software with the help of shift and XOR operations [13]. This emulated instruction, which we call MULGF2 as in [13], opens up the possibility to use *word-level algorithms* for the multiplication of binary polynomials, similar to the schoolbook or Comba method for the multiplication of integers. Word-level algorithms for binary polynomials are discussed in detail in [10].

4. Implementation on MIPS32

Typical software implementations of an EC cryptosystem spend the majority of the execution time in the inner loop of the field multiplication. Speeding up this critical code section (e.g., through hand-crafted assembly code or dedicated instruction set extensions) can result in a tremendous performance gain. In the following, we discuss the efficient implementation of diverse algorithms for field multiplication on a MIPS32 processor [16], assuming also the possibility of extending the native instruction set. In addition, we provide experimental results obtained through simulations with SimpleScalar [5]. These experimental results include the timings for a single field multiplication, as well as the total execution time of a so-called *point multiplication*, which is the major building block of an EC cryptosystem [3]. A point multiplication is performed by a sequence of field operations (see [12] for details), and hence serves as a benchmark for the efficiency of the field arithmetic.

4.1. Multiplication in Prime Fields

The execution time of a multiplication in a prime field is proportional to s^2, whereby s is the number of 32-bit words needed to store an element element of the field. Both the schoolbook and Comba's method execute single-precision multiplications and additions in their inner loops. However, the inner loop of the schoolbook algorithm can be better optimized on MIPS32 processors.

The inner loop of the schoolbook method performs an operation of the form $z + a \cdot b + u$, with all four operands being 32-bit words. A whitepaper by MIPS Technologies [4] recommends to use the MADDU instruction to add the 32-bit words z and u to the 64-bit product $a \cdot b$. Figure 3 illustrates a highly-optimized assembly implementation of the inner loop of the schoolbook method (see [4] for a more detailed description). A MIPS32 core with a (32×16)-bit multiplier executes the instruction sequence shown in Figure 3 in 11 cycles, provided that the instructions are ordered properly to fill load/branch delay slots and that the load operations hit the data cache. Table 1 specifies the overall execution time of a multiplication in a 192-bit prime field ($s = 6$) and the execution time of a full point multiplication.

Operation	Schoolbook	Comba
Field mul. (conv. SW)	629	827
Field mul. (with ISE)	485	441
Point mul. (conv. SW)	$2.16 \cdot 10^6$	$2.84 \cdot 10^6$
Point mul. (with ISE)	$1.67 \cdot 10^6$	$1.47 \cdot 10^6$
Speed-up factor	1.29	1.93

Table 1. Simulation results for 192-bit prime field.

As mentioned before, the inner loop of the schoolbook method performs a single-precision multiplication and two additions. Obviously, the highest performance gain can be achieved when all operations of the inner loop are combined into a single custom instruction, which was first proposed in [8]. It was shown in [11] that the availability of a custom instruction for calculating $z + a \cdot b + u$ allows to implement the inner loop with only 7 instructions. As a consequence, the overall execution time of a field multiplication is reduced from 629 to 485 clock cycles (see Table 1), and the point multiplication becomes approximately 29% faster.

```
l1: lw    $t0, 0($t1)   # load A[j]
    lw    $t2, 0($t3)   # load B[k]
    addiu $t1, $t1, 4   # increment A pointer
    maddu $t0, $t2      # (HI|LO)=(HI|LO)+A[j]*B[k]
    bne   $t3, $t4, l1  # branch if not end of loop
    addiu $t3, $t3, -4  # decrement B pointer
```

Figure 4. Inner loop of Comba multiplication.

The inner loop of Comba's multiplication technique performs a "classical" MAC operation, i.e., two 32-bit words are multiplied and the product is added to a running sum. At a first glance, it seems that the MADDU instruction provides exactly the functionality needed for this operation. However, the problem is that the accumulator of a standard MIPS32 core is only 64 bits wide, and thus the MAC unit is not able to sum up several 64-bit products without overflow. According to our experiments, the inner loop of Comba's method can not be executed in less than 18 cycles on a MIPS32 core.

All limitations of the MIPS32 architecture can be easily mitigated by tailoring the MAC unit to suit the needs of long integer arithmetic. Comba's multiplication method requires

a MAC unit with a "wide" accumulator so that a number of 64-bit products can be summed up without overflow. On a MIPS32 processor with a "wide" accumulator, the inner loop of the Comba multiplication can be implemented as shown in Figure 4. Any iteration of the loop requires only six clock cycles to complete, even on a MIPS32 processor with a (32×16)-bit multiplier, since the `BNE` instruction can be executed during the second cycle of the `MADDU`. The two `ADDIU` instructions, which do simple pointer arithmetic, are used to fill a load and branch delay slot, respectively. Our simulations show that an extended MIPS32 core is able to execute a 192-bit Comba multiplication in 441 clock cycles (see Table 1), which is almost twice as fast as the implementation with native MIPS32 instructions. Also the full point multiplication is accelerated by roughly the same factor.

The most important aspect to observe is that instruction set extensions change the relative performance of the two algorithms: the schoolbook method is faster on a conventional MIPS32 processor, but the Comba method wins when custom instructions are available.

4.2. Multiplication in Binary Fields

A concrete implementation of the inner loop of the Shift-and-XOR method (Algorithm 1) performs simple operations like loads, stores, shifts, and XORs on 32-bit words (see [12] for for a detailed description). While the inner loop is quite fast on MIPS32 processors (less than 10 cycles), the performance of the algorithm suffers from the high number of iterations. The MIPS architecture has no special features from which the inner loop operation could profit. Therefore, the execution time for a multiplication in a binary field is relatively slow (e.g., 2758 cycles when the polynomials have a degree of 191), which also impacts the overall execution time of the full point multiplication (see Table 2).

Operation	Shift+XOR	Word-level
Field mul. (conv. SW)	2758	7848
Field mul. (with ISE)	2151	456
Point mul. (conv. SW)	$4.05 \cdot 10^6$	$10.42 \cdot 10^6$
Point mul. (with ISE)	$3.28 \cdot 10^6$	$0.87 \cdot 10^6$
Speed-up factor	1.23	11.97

Table 2. Simulation results for 191-bit binary field.

The inner loop of the Shift-and-XOR multiplication can be accelerated by combining a shift and an XOR operation to a custom instruction. This custom instruction can be simply realized by passing one operand of the XOR through a barrel shifter, similar to the ARM architecture. Using the custom instruction saves two clock cycles in any iteration of the inner loop, which reduces the overall execution time of a multiplication in GF(2^{191}) from 2758 to 2151 cycles.

The word-level multiplication algorithms are adoptions of the schoolbook and the Comba method for binary polynomials. Both have in common that the performance depends heavily on the efficiency of the MULGF2 instruction. However, since this instruction is not available on conventional MIPS32 processors, it must be emulated in software using shift and XOR instructions [13]. Despite our best effort, we were not able to reduce the execution time of the MULGF2 operation to less than 190 cycles. Therefore, the word-level algorithms are very slow on a MIPS32 processor.

An obvious way to speed up the word-level technique is to implement the MULGF2 instruction in hardware, e.g., on a dedicated polynomial multiplier. Executing the MULGF2 instruction in hardware in one or two cycles accelerates the word-level multiplication by a factor of as much as 12 compared to a conventional software implementation with emulated MULGF2. This tremendous performance gain is not surprising when considering that the emulation of the MULGF2 instruction requires 190 cycles. Table 2 contains the execution time of the word-level multiplication with and without hardware support for the MULGF2 operation.

Our simulation results clearly demonstrate that instruction set extensions change the relative performance of the two algorithms. The conventional software implementation of the word-level multiplication with emulated MULGF2 instruction is much slower than the Shift-and-XOR method. However, the situation is totally different when MULGF2 is executed in hardware. In this case, a field multiplication according to the word-level technique is almost five times faster than the Shift-and-XOR method (see Table 2).

5. Automatic Exploration

The typical design flow for manual selection of ISE is as follows: the designer takes the source code of the fastest possible software implementation as starting point and tries to identify the critical code sections. Thereafter, ISEs are defined and evaluated with the goal to speed up the critical code sections. However, this approach is not viable since it leads to sub-optimal results in application domains where a number of different but equivalent implementation options exist, e.g., different data structures or different algorithms. EC cryptography is a typical example of such an application domain since there exist many different algorithms for one and the same arithmetic operation, e.g., multiplication in a finite field. This calls for a systematic approach to explore the algorithmic design space.

Devising tools that attempt to automate the ISE identification process is an active research discipline. Several tools have been presented in the past years [2, 6, 18], and all aim at efficiently and quickly generating the best ISEs for a given application by automatic analysis of the application source code. We applied the automatic ISE (AutoISE) generation

Arithmetic algorithm	2/1	3/1	3/2	6/2
GF(p) Schoolbook	1.11	1.19	1.28	1.34
GF(p) Comba	1.11	1.13	1.21	1.21
GF(2^m) Shift-and-XOR	1.44	1.59	1.77	1.89
GF(2^m) Word-level	2.02	2.17	5.14	5.48

Table 3. Speed-up factors by automatic ISE.

technique proposed in [2] to the arithmetic algorithms for EC cryptography from Section 3. The AutoISE tool allows selection of any number of ISEs from an application source code, under user-given micro-architectural constraints defining the number of input/output ports that the chosen ISEs are allowed to use[1]. AutoISE uses a simple but effective estimation of the speed-up for selecting among millions of identified ISEs. Table 3 summarizes the achieved speed-up factors depending on the number of input/output ports.

The key point to observe is that, even when using a very rough performance estimation model, the results obtained through the automatic ISE tool are in good agreement with those specified in Section 3; in particular, it shows that the word-level algorithm for binary fields can be accelerated by a much higher degree than the other algorithms. In a manual ISE design flow, the word-level multiplication for GF(2^m) would likely be ignored due to its poor software performance. On the other hand, automatic ISE tools reduce the risk of overlooking a good candidate algorithm. These tools can predict the important trends correctly and guide system designers efficiently and effectively, while screening them from architectural details.

6. Conclusions

In this paper we have shown that automatic instruction set extension is not only a tool for improving the performance of embedded application execution or for achieving fast exploration of customized architecture solutions. An additional motivation to automate the ISE selection process is to help *algorithm exploration*. Via a study based on EC cryptography, we have shown that the availability of ISE can have a dramatic impact on the relative performance of different algorithmic choices. We have first manually selected ISEs for different EC implementations, and have measured achieved speed-up by simulation, using a detailed model of the ISEs chosen. Our study shows for the first time that the availability of ISE can reverse the relative interest of different algorithm choices. Furthermore, we have run an automatic ISE tool and demonstrated that, even without predicting speed-ups as precisely as detailed simulation can, it is able to show exactly and in a matter of seconds the correct trends that the system designer should follow.

[1] A detailed discussion of the instruction selection process of the AutoISE tool is beyond the scope of this paper. Here we simply want to demonstrate the usefulness of the AutoISE tool for a high-level exploration.

References

[1] ARM Limited. SecurCore™ Solutions. Product brief, available for download at `http://www.arm.com`, Feb. 2002.

[2] K. Atasu, L. Pozzi, and P. Ienne. Automatic application-specific instruction-set extensions under microarchitectural constraints. In *Proceedings of the 40th Design Automation Conference (DAC 2003)*, pp. 256–261. ACM Press, 2003.

[3] I. F. Blake, G. Seroussi, and N. P. Smart. *Elliptic Curves in Cryptography*. Cambridge University Press, 1999.

[4] W. Bond. 64-bit architecture speeds RSA by 4x. Whitepaper, available for download at `http://www.mips.com`, 2002.

[5] D. C. Burger and T. M. Austin. The SimpleScalar Tool Set, Version 2.0. Technical Report CS-TR-97-1342, University of Wisconsin, Madison, WI, USA, June 1997.

[6] N. Clark, H. Zhong, and S. Mahlke. Processor acceleration through automated instruction set customization. In *Proceedings of the 36th Annual International Symposium on Microarchitecture (MICRO-36)*, pp. 129–140. ACM Press, 2003.

[7] P. G. Comba. Exponentiation cryptosystems on the IBM PC. *IBM Systems Journal*, 29(4):526–538, Dec. 1990.

[8] J.-F. Dhem. *Design of an efficient public-key cryptographic library for RISC-based smart cards*. Ph.D. Thesis, Université Catholique de Louvain, Louvain-la-Neuve, Belgium, 1998.

[9] P. Faraboschi, G. Brown, J. A. Fisher, G. Desoli, and F. Homewood. Lx: A technology platform for customizable VLIW embedded processing. In *Proceedings of the 27th Annual International Symposium on Computer Architecture (ISCA 2000)*, pp. 203–213. ACM Press, 2000.

[10] J. Großschädl and G.-A. Kamendje. Instruction set extension for fast elliptic curve cryptography over binary finite fields GF(2^m). In *Proceedings of the 14th Conference on Application-specific Systems, Architectures and Processors (ASAP 2003)*, pp. 455–468. IEEE Computer Society Press, 2003.

[11] J. Großschädl and G.-A. Kamendje. Optimized RISC architecture for multiple-precision modular arithmetic. In *Security in Pervasive Computing — SPC 2003*, LNCS 2802, pp. 253–270. Springer Verlag, 2003.

[12] D. R. Hankerson, A. J. Menezes, and S. A. Vanstone. *Guide to Elliptic Curve Cryptography*. Springer Verlag, 2004.

[13] Ç. Koç and T. Acar. Montgomery multiplication in GF(2^k). *Designs, Codes and Cryptography*, 14(1):57–69, Mar. 1998.

[14] R. Lidl and H. Niederreiter. *Introduction to Finite Fields and Their Applications*. Cambridge University Press, 1994.

[15] A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1996.

[16] MIPS Technologies, Inc. MIPS32™ Architecture for Programmers. Available for download at `http://www.mips.com`, Mar. 2001.

[17] MIPS Technologies, Inc. SmartMIPS® Architecture Smart Card Extensions. Product brief, available for download at `http://www.mips.com`, Feb. 2001.

[18] L. Pozzi, K. Atasu, and P. Ienne. Exact and approximate algorithms for the extension of embedded processor instruction sets. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, to appear.

[19] J. Turley. Tensilica CPU bends to designers' will. *Microprocessor Report*, 13(3):12, Mar. 1999.