Accelerating Next-Generation Public-Key Cryptosystems on General-Purpose CPUs

This article describes low-cost techniques for accelerating the ECC and RSA public-key cryptosystems on general-purpose processor architectures. A prototype based on a Sparc CPU data path shows a clear performance advantage of ECC over RSA.

Hans Eberle
Sheueling Shantz
Vipul Gupta
Nils Gura
Leonard Rarick
Lawrence Spracklen
Sun Microsystems Laboratories

•••••• The demand for secure communication over the Internet, fueled by e-commerce and mobile applications, is growing. Connecting small, simple devices such as PDAs, mobile phones, and smart cards to the Internet is also a growing trend. Many of these devices require secure connections to ensure that the exchanged information remains confidential and that only authorized persons can access the devices.

Recognizing these trends, computer and processor designers are beginning to optimize general-purpose CPUs for high security performance. For example, the Intel Itanium processor executes 1,000 RSA-1024 operations per second using 128 integer registers and two 64-bit integer multipliers, and Sun Microsystems has outlined plans to provide integrated cryptographic features on upcoming Sparc processor chips.

The growing volume of secure transactions will challenge Web server systems. Moreover, a transition to larger key sizes is necessary to maintain strong security against the increasing computer power available to invade cryptosystems. Specifically, looking at SSL and IPsec, the most widely used secure protocols today, we can expect a transition from 1,024-bit to 2,048-bit key sizes for the public-key cryptosystem RSA and a replacement of the Data Encryption Standard (DES) and RC4 bulk encryption algorithms by the Advanced Encryption Standard (AES) cipher.

In this article, we focus on public-key cryptosystems. Although RSA has been the most widely used public-key cryptosystem for 20 years, its computational demands are becoming prohibitive for mobile and lightweight devices. Doubling the size of the RSA key leads to an approximate eightfold increase in computation time as the computational effort grows proportionally to the cube of the key size. An increase in RSA key size is becoming necessary because the NIST and RSA Laboratories recommend using the current 1,024-bit key size to protect data only until the year 2010.

Elliptic curve cryptography (ECC) is an attractive alternative to RSA: It is computationally more efficient and its computational demands increase less as key size increases. Therefore, ECC is in a better position than RSA to handle the increasing volume of secure transactions and to provide the next level of cryptographic strength. Even more important, ECC is the public-key cryptosystem of choice for small devices because its computational efficiency leads to savings in computer power, memory capacity, and power consumption.[1]

Implementations of cryptosystems differ in constraints and requirements. Here, we focus on hardware acceleration of public-key cryptosystems on 64-bit server machines. Servers are confronted with an aggregation of secure connections created by a multitude of heterogeneous clients. Thus, high throughput and flexibility in handling different standards are our optimization criteria for a public-key cryptosystem.

Adding instruction set extensions to general-purpose processor architectures to improve performance in certain application areas goes back several decades. While early computers such as the PDP-11 were microprogrammable and even allowed a user to add custom instructions, modern RISC processors with their hardwired control logic do not provide this flexibility anymore. Thus, any addition requires a careful analysis of the cost/performance tradeoff as well as of the compatibility with the existing data path. Examples of extensions targeted at multimedia applications are Intel's multimedia extension (MMX) and Sun's visual instruction set (VIS). Recently, extensions have also been considered for cryptographic applications. For example, instructions for accelerating symmetric key cryptosystems have been proposed by Burke et al.[2] and Lee et al.[3] More closely related to our work, Großschädl describes an instruction set extension for accelerating multiple-precision multiplications in RSA.[4]

Public-key cryptography

Compared with cryptosystems used for bulk encryption and for calculating the message digest, public-key cryptosystems are more computational intensive and less data intensive. Thus, they are well suited for today's processor architectures that favor data processing over data moving, especially if data accesses exhibit locality. Public-key cryptosystems rely heavily on multiplication operations, which are typically well supported on general-purpose processors.

The acceleration techniques described here target today's dominant cryptosystem RSA and the emerging ECC system. RSA is based on modular exponentiation of large integers which can be implemented through repeated multiplication and squaring. RSA's security relies on the difficulty of factoring large integer values. The cryptographic function defined in ECC is called scalar point multiplication. Here, the underlying hard problem is known as the elliptic curve discrete logarithm problem (ECDLP). While subexponential algorithms exist for factoring large numbers, only exponential algorithms exist for the ECDLP. It is for this reason that RSA requires larger key sizes than ECC to provide equivalent security strength. To illustrate the key size advantage of ECC: A 139-bit ECC key offers the same security strength as a 1,024-bit RSA key. And the ratio of key sizes will favor ECC even more as larger keys are adopted.[5]

The basis of both RSA's modular exponentiation and ECC's point multiplication is modular arithmetic. RSA uses integer rings, whereas ECC is defined over prime integer fields $GF(p)$ and binary polynomial fields $GF(2^m)$. We can easily implement arithmetic operations in integer rings and over fields $GF(p)$ with the standard integer operations available on a general-purpose processor. This is not the case for arithmetic operations over fields $GF(2^m)$. Although hardware can handle these operations rather efficiently, executing them on standard processors is prohibitively slow.

ECC

ECC scalar point multiplication computes $Q = kP$; that is, it multiplies a scalar k (a large integer number) and a point P on an elliptic curve. The result is another point Q on the elliptic curve. In a public-key operation, k is the private key, Q is the public key, and P is a curve parameter called the base point.

Scalar point multiplication can be decomposed into point additions and point doublings. For example, $9P$ can be computed with one point addition and three point doublings because $9P = (2 * (2 * (2P))) + P$. Note that point additions and point doublings are not simple arithmetic operations but rather correspond to sequences of modular arithmetic operations. Hankerson, Menezes, and Vanstone provide a detailed mathematical background of ECC.[6]

The following ECC implementations use a double-and-add algorithm for point multiplications over fields $GF(p)$ and Montgomery scalar multiplication for point multiplications over fields $GF(2^m)$. We use projective coordinates for $GF(2^m)$ and mixed coordinates for $GF(p)$.[7,8]

ECC arithmetic over GF (2^m)

Here, we introduce the arithmetic operations needed to implement ECC point multiplication over binary polynomial fields $GF(2^m)$, which require instructions typically not found in standard processors. The operations include modular addition, subtraction, multiplication, squaring, and division, in which the operands are polynomials with coefficients of either 0 or 1. We use the polynomial basis representation in which a polynomial $a(t) \in GF(2^m)$ in canonical form is written $a(t) = a_{m-1}t^{m-1} + a_{m-2}t^{m-2} + \ldots + a_1 t + a_0$, $a_{m-1..0} \in GF(2)$. For computation, an m-bit binary vector can represent the coefficients. For example, polynomial $t^4 + t^3 + 1$ can be written 11001.

Addition. We compute the addition of two elements $a(t), b(t) \in GF(2^m)$ by adding the coefficients a_i and b_i modulo 2, where i= m-1..0, which corresponds to a bit-wise XOR operation:

$$a(t) + b(t) = \sum_{i=0}^{m-1} \left((a_i + b_i) \bmod 2 \right) * t^i = \sum_{i=0}^{m-1} \left(a_i \oplus b_i \right) * t^i$$

Multiplication. We define the multiplication of two elements $a(t), b(t) \in GF(2^m)$ modulo an irreducible polynomial $M(t)$ as

$$a(t) * b(t) = \sum_{i=0}^{m-1} \sum_{j=0}^{m-1} \left(a_i b_j \right) t^{i+j} \bmod M(t)$$

Modular multiplication can be executed in two steps: first, the unreduced product is computed, and then the product is reduced to the field size m. In a corresponding multiplier circuit AND gates generate the partial products $a_i b_j$, and XOR gates perform the addition of the partial products. Because of this XOR summation, we refer to multiplications over $GF(2^m)$ as XOR multiplications. The reduction mod $M(t)$ can be performed subtracting multiples of $M(t)$ repetitively until the degree of the result is less than m.

Modular multiplication

Modular multiplication is the most performance-critical operation underlying both the RSA and the ECC cryptosystems. Because both cryptosystems operate on operands wider than a 64-bit data path, they require efficient multiple-precision operations. We will first examine how modular multiplication is implemented as a multiple-precision operation. Next, we will describe a multiplier architecture that efficiently supports multiple-precision modular multiplication.

Multiple-precision multiplication

Figure 1 depicts the calculation of a multiple-precision multiplication and how it is broken up into multiplication instructions. The given example assumes 2,048-bit operands A and B. Operands are broken up into 64-bit words a_i and b_i, where $i = 0\ldots31$. Modular product C is the sum of partial products $a_i * B$. To calculate a partial product, 32 multiplications are required, and to calculate the final product, 1,024 multiplication operations have to be executed.

RSA and ECC are both based on modular arithmetic. Thus, the multiple-precision multiplication operations require an additional reduction step. An efficient and widely used technique is attributed to Montgomery.[9] This technique replaces the costly division operation required to calculate the remainder of the modulo operation with a simple addition of a multiple of the modulus. Again, this step requires a multiple-precision multiplication. Thus, the reduction step requires as many multiplications as the calculation of the partial products. For the given example, the calculation of the reduced product requires 2,048 64-bit multiplication operations.

Acceleration techniques

Analyzing the distribution of instructions executed in ECC or RSA operations, we find that the multiplication clearly dominates. We can illustrate this with numbers obtained for the second-generation processor described later in this article. The fraction of instructions spent on multiple-precision multiplications is 86 percent for RSA-1024, 63 percent

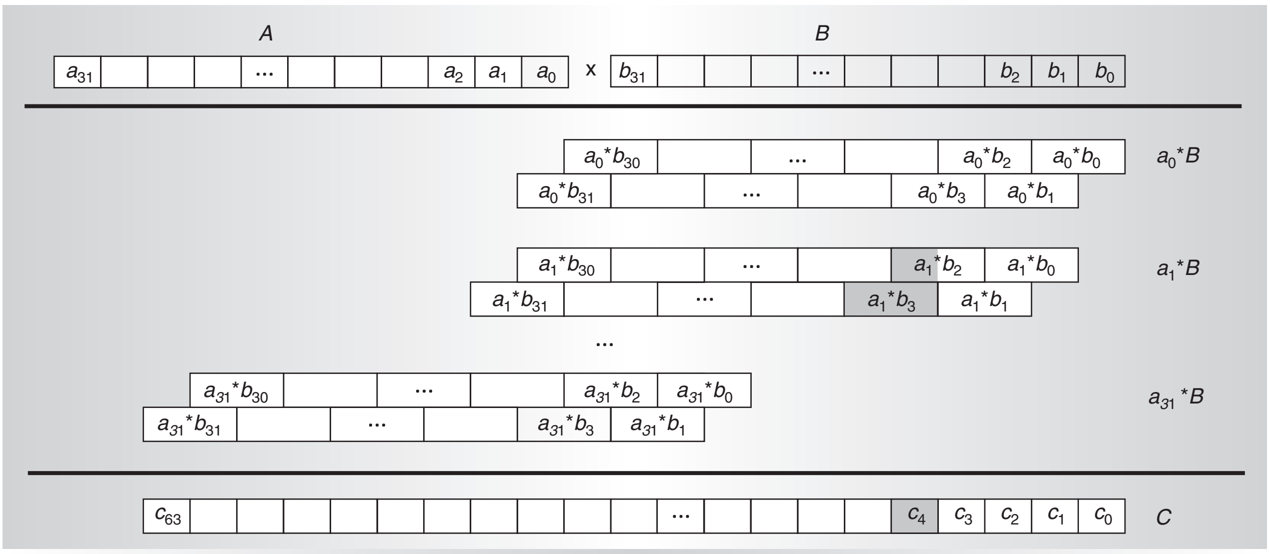

Figure 1. Multiple-precision multiplication.

for ECC-160, and 90 percent for ECC-163. Consequently, we have focused our efforts on accelerating multiple-precision multiplication operations. Here, we describe two techniques for accelerating multiplication. First, we introduce a dual-field multiplier that generates not only the integer results needed for RSA and ECC over fields *GF*(*p*) but also the results needed for ECC operations over fields *GF*(2^m). Next, we describe a multiply-accumulate instruction that allows efficient scheduling of multiplication operations.

Dual-field multiplier

Parallel multipliers typically use a carry-save adder (CSA) tree and a carry-propagate adder (CPA). The CSA tree calculates the sum of the partial products in a redundant carry/sum representation, and the CPA performs the final addition of the carry and sum bits. We modified the CSA tree so that it generates an XOR multiplication product in addition to the integer product. The former is needed for ECC over fields *GF*(2^m), and the latter for RSA and ECC over fields *GF*(*p*).

A CSA tree consists of full-adder (FA) elements and half-adder (HA) elements. In its simplest form, a tree for an n-bit multiplier uses 2*n* chains, consisting of 1 to *n-1* FAs and HAs to sum up *n* partial products. Techniques such as tree compression and Booth encoding can be used to reduce the chain length and, with it, the logic delay to obtain the carry/sum result.

We can modify the CSA tree to obtain the XOR result in addition to the integer result. Looking at the functions realized by the FAs and the HAs, we notice that sum *S* already provides the XOR function: (*FA*) $S = A \oplus B \oplus C_{in}$, C_{out} = (*A* Ÿ *B*) / (*A* Ÿ C_{in}) / (*B* Ÿ C_{in}); (*HA*) $S = A \oplus B$, C_{out} = *A* Ÿ *B*.

Thus, we obtain the XOR result by chaining the FAs and HAs so that inputs to the CSA tree are added first and carry bits C_{out} are added as late as possible at the bottom of the tree. Figure 2 illustrates how we modify a column of a CSA tree for a 6 ¥ 6-bit multiplier to generate an XOR multiplication result in addition to an integer multiplication result. The modifications require little extra circuitry—some columns require the addition of an XOR gate—and don't increase logic delays. (Because the XOR multiplication result is generated early, the critical path is not affected even if the extra XOR gate is needed.) Analyzing the extra cost of adding support for XOR multiplication results, we found an average increase of 5 percent in logic required. The outlined modifications can easily be applied to multiplier designs in general-purpose processors.

Adding support for XOR multiplications results in a dramatic performance gain. An XOR multiplication replaces approximately 100 Sparc instructions. We obtained speedups of 13.8 times for a 256-bit XOR multiplication and 8.5 times for scalar-point multiplication.

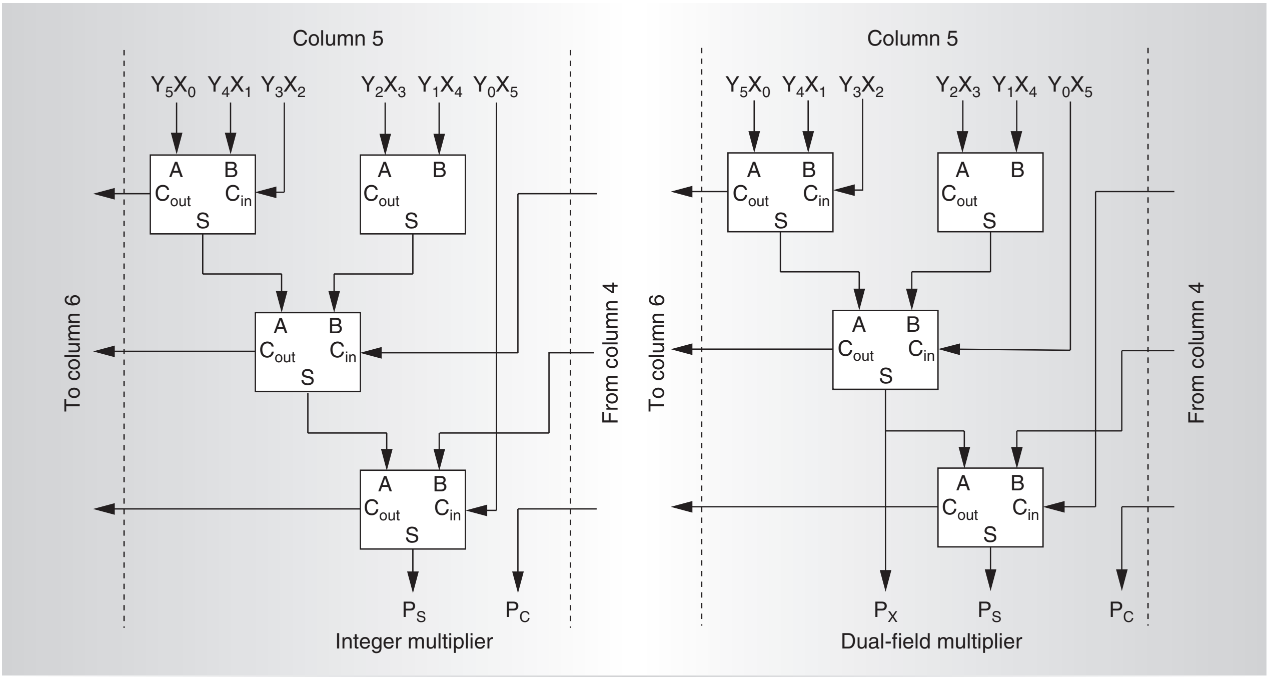

Figure 2. CSA tree of an integer multiplier (a) and modified CSA tree of a dual-field multiplier (b).

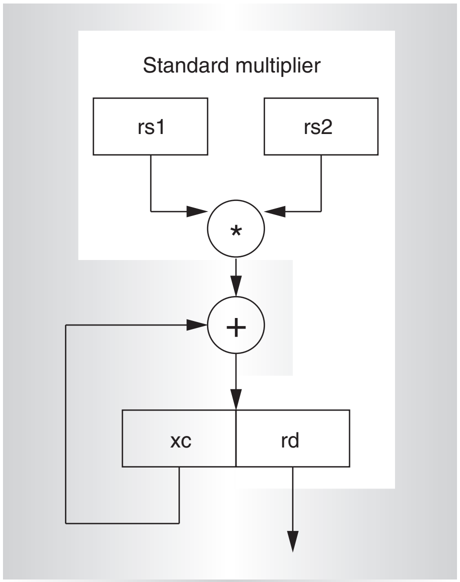

Figure 3. Extended multiplier for the *mulacc* instruction.

Multiply-accumulate instruction

We introduce multiply-accumulate instruction *mulacc* to efficiently execute multiple-precision multiplications. Figure 3 shows the extended multiplier used by the instruction. In addition to source registers rs1 and rs2 and destination register rd, the multiplier uses nonarchitectural register xc. We call this register the extended carry because it stores the result's high word. As Figure 4 shows, *mulacc* combines a 64-bit multiplication and a 128-bit addition. (We need 128-bit additions because the multiplication's result is 128 bits wide.) This instruction retains both the low and the high words of the result, unlike other processor architectures that deliver either the low or the high word of the two-word result.

Referring to the highlighted fields in Figure 1, the partial product of a multiple-precision multiplication is calculated through a series of *mulacc* instructions: At each step, one word each of the multiplier (rs1=a_1) and multiplicand (rs2=b_3) are multiplied, then the high word of the result of the previous step (xc) is added.

To support operations over both fields $GF(p)$ and fields $GF(2^m)$, we must define two versions of the multiply-accumulate instruction: a regular integer version and an XOR version that combines an XOR multiplication and an XOR addition.

We optimize the multiply-accumulate instruction's implementation by folding the accumulation operation into the CSA tree needed for the multiplication operation. The CSA tree already performs additions (of the partial products), and including the accumulation operation simply adds *xc* as another input term of the CSA tree.

Prototype systems

We have implemented two generations of public-key cryptographic processors. As a proof of concept, we prototyped them in a Xilinx Virtex-II XC2V6000 FPGA. We interfaced the FPGA to the host system via a PCI bus. The 66-MHz PCI clock serves as the processor's main clock. Figure 5 shows the prototypes' floorplans.

The first-generation processor has a 256-bit data path optimized for processing ECC point multiplication over fields $GF(2^m)$.[10] Figure 6a shows the data path, which contains a data memory, a register file, and several function units including a multiplier, a divider, an adder, and a squarer. There are two 256-bit wide buses for transferring source operands and destination operands. As Figure 5a shows, we allocated most chip resources to the multiplier circuit in an attempt to optimize its performance. We achieved a performance of 6,955 ECC-163 operations per second.

Our second-generation processor is more versatile in functionality and more general in architecture.[11] It can perform both RSA modular exponentiation and ECC point multiplication. With respect to ECC, it supports fields $GF(p)$ and $GF(2^m)$. Our goal was a design that could be easily integrated in a general-purpose CPU. The data path shown in Figure 6b reflects this goal. We adhered to a 64-bit data path containing general-purpose function units including a multiplier, an adder, and a subtracter. We omitted any special-purpose function units. Specifically, we abandoned the divider of the previous design and implemented division with the help of Fermat's little theorem, which relies on multiplication operations only. We found that the lack of a hardware divider increases execution time for point multiplications over $GF(2^m)$ by about 7 percent. Furthermore, we didn't provide a squarer for $GF(2^m)$, and we replaced squaring operations with regular multiplication operations. In this case, the performance impact is much more dramatic. For our first-generation prototype, the lack of a squarer would increase a point multiplication's execution time by 44 percent. However, because squaring operations at the instruction level can be optimized only for $GF(2^m)$ and not for $GF(p)$, it is questionable to provide this optimization if we want to support both field types and strive for balanced performance.

The second-generation processor's data path lacks a register file. This is less an architectural choice than a simplification of the implementation. Because the ECC or RSA algorithms' long operands would not fit into 64-bit registers, register management would have been complicated. We simplified the design by omitting this memory indirection

```
mulacc rs1,rs2,rd

rd ← (rs1*rs2+xc)[63:0]
xc ← (rs1*rs2+xc)[127:64]
```

Figure 4. Multiply-accumulate instruction *mulacc*.

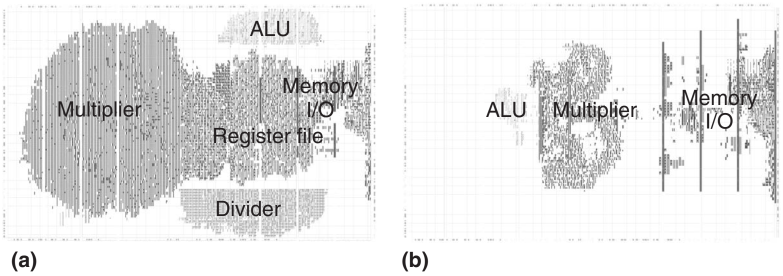

Figure 5. Floorplans of first-generation (a) and second-generation (b) prototypes.

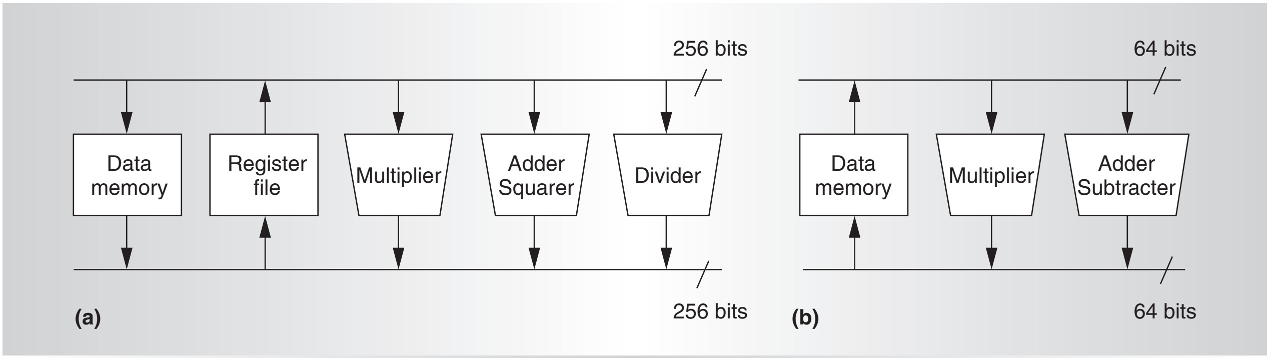

Figure 6. Data paths of first-generation (a) and second-generation (b) prototypes.

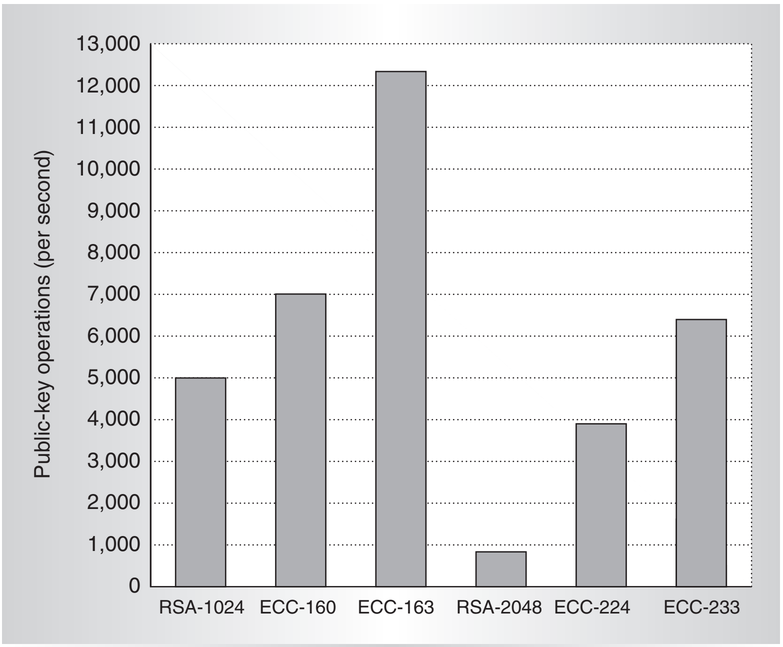

Figure 7. ECC and RSA performance.

and keeping all operands in memory. We did, however, provide an indexed addressing mode to make memory accesses more efficient.

We did not optimize the second-generation processor's performance because it mainly serves as a proof of concept. Specifically, it executes instructions in several cycles. The counts of executed instructions, however, are similar to those of the prototype described next.

Performance analysis

We analyzed the performance of an implementation of the ECC and RSA algorithms in a prototype Sparc CPU. The implementation is based on the acceleration techniques previously described. In particular, we used a dual-field multiplier. The multiplier is fully pipelined and capable of initiating a new multiplication operation every cycle. The operating frequency is 1.5 GHz.

Figure 7 lists the number of operations per second for RSA, ECC over GF(p) (ECC-160, ECC-224) and ECC over GF (2^m) (ECC-163, ECC-233). More specifically, we considered RSA decrypt operations using the Chinese Remainder Theorem and ECC point multiplications. The numbers show a clear performance advantage of ECC over RSA. At present security levels, ECC-160 offers a 1.4 speedup and ECC-163 a 2.5 speedup over RSA-1024. At future security levels, the comparison favors ECC even more: ECC-224 is 4.7 times and ECC-233 is 7.7 times faster than RSA-2048.

These numbers not only show a significant performance advantage for ECC, they further illustrate that the ratio of RSA and ECC computation times will dramatically increase as higher security strength is needed.

We are currently investigating implementations of public-key cryptosystems on small devices. We have implemented both the RSA and the ECC cryptosystems, in particular of ECC over fields GF (2^n) (ECC-163 and ECC-223). and the ECC cryptosystems on 8-bit microcontrollers and shown that public-key cryptography is a viable option on these devices.[1] We find that techniques similar to the ones described here can be applied to accelerate public-key cryptography on microcontrollers without having to rely on costlier solutions based on cryptographic coprocessors. Comparing the two cryptosystems we find that the computional effectiveness of ECC over RSA is even greater for the narrow data path found in microprocessors. MICRO

Acknowledgments

We thank Daniel Finchelstein, Edouard Goupy, and Shreyas Sundaram for their help in implementing the prototype systems.

References

1. N. Gura et al., "Elliptic Curve Cryptography and RSA on 8-bit CPUs," *Proc. 6th Int'l Workshop Cryptographic Hardware and Embedded Systems* (CHES 04), Springer, 2004, pp. 92-106.
2. J. Burke, J. McDonald, and T. Austin, "Architectural Support for Fast Symmetric-key Cryptography," Proc. *Proc. 9th Int'l Conf. in Architectural Support for Programming Languages and Operating Systems* (ASPLOS 00), ACM Press, 2000, pp. 178-189.
3. R. Lee, Z. Shi, X. Yang, "Efficient Permutation Instructions for Fast Software Cryptography," *IEEE Micro*, vol. 21, no. 6, Dec. 2001, pp. 56-69.
4. J. Großschädl, "Instruction Set Extension for Long Integer Modulo Arithmetic on RISC-Based Smart Cards," *Proc. 14th Symp. Computer Architecture and High Performance*

Computing, IEEE Press, 2002, pp. 13-19.

5. A. Lenstra and E. Verheul, "Selecting Cryptographic Key Sizes," *J. Cryptology: J. Int'l Assoc. Cryptologic Research*, vol. 14, no. 4, 2001, pp. 255-293.
6. D. Hankerson, A. Menezes, and S. Vanstone, *Guide to Elliptic Curve Cryptography*, Springer, 2004.
7. H. Cohen, A. Miyaji, and T. Ono, "Efficient Elliptic Curve Exponentiation Using Mixed Coordinates," *Int'l Conf. Theory and Application of Cryptology (AsiaCrypt 98), Advances in Cryptology*, Springer Verlag, 1998, pp. 51-65.
8. J. López and R. Dahab, "Fast Multiplication on Elliptic Curves over *GF*(2^m) without Precomputation," *Proc. 1st Int'l Workshop Cryptographic Hardware and Embedded Systems* (CHES 99), Lecture Notes in Computer Science 1717, Springer-Verlag, 1999, pp. 316-327.
9. P. Montgomery, "Modular Multiplication without Trial Division," *Mathematics of Computation*, vol. 44, no. 170, Apr. 1985, pp. 519-521.
10. H. Eberle, N. Gura, and S. Chang Shantz, "A Cryptographic Processor for Arbitrary Elliptic Curves over *GF*(2^m), *Proc. IEEE 14th Int'l Conf. Application-Specific Systems, Architectures and Processors* (ASAP 03), IEEE Press, 2003, pp. 444-454.
11. H. Eberle et al., "A Public-Key Cryptographic Processor for RSA and ECC," *Proc. IEEE 15th Int'l Conf. Application-Specific Systems, Architectures and Processors* (ASAP 04), IEEE Press, 2004, pp. 98-110.

Hans Eberle is a distinguished engineer at Sun Microsystems Laboratories. His research interests include networks and security. Eberle has a MS in electrical engineering and a PhD in technical sciences from the Swiss Federal Institute of Technology, Zurich. He is a member of the IEEE.

Sheueling Chang Shantz is a distinguished engineer at Sun Microsystems Laboratories. Her research interests include next-generation Internet security, efficient software implementation and hardware acceleration of Elliptic Curve Cryptosystems. Chang Shantz has a PhD in computer science from California Institute of Technology.

Vipul Gupta is a senior staff engineer at Sun Microsystems Laboratories. His research interests include Internet security and small wireless devices. Gupta has a MS and PhD in computer science from Rutgers University. He is an active participant in the Internet Engineering Task Force (IETF).

Nils Gura is a member of technical staff at Sun Microsystems Laboratories. His research interests include efficient algorithms and hardware acceleration for next-generation cryptographic technologies, primarily Elliptic Curve Cryptography. Gura has a MS in computer science from the University of Ulm, Germany. He is a member of the IEEE.

Leonard Rarick is a senior staff hardware engineer at Sun Microsystems Laboratories. His research interests include cryptographic algorithms, floating point hardware design, and linear programming. Rarick has a PhD degree in computer science from Columbia University.

Lawrence Spracklen is a senior processor architect at Sun Microsystems Laboratories in the Advanced Processor Architecture group. His research interests include next-generation chip multithreaded processors. Spracklen has a PhD in electrical engineering from the University of Aberdeen.

Direct questions and comments about this article to Hans Eberle, Sun Microsystems Laboratories, UMPK16-159, 16 Network Circle, Menlo Park, CA 94025; hans.eberle@sun.com.