Fast Modular Reduction

William Hasenplaugh, Gunnar Gaubatz, Vinodh Gopal
Intel
{william.c.hasenplaugh, gunnar.gaubatz, vinodh.gopal}@intel.com

Abstract

It is widely acknowledged that efficient modular multiplication is a key to high-performance implementation of public-key cryptography, be it classical RSA, Diffie-Hellman, or (hyper-) elliptic curve algorithms. In the recent decade, practitioners have relied mainly on two popular methods: Montgomery Multiplication and regular long-integer multiplication in combination with Barrett's modular reduction technique. In this paper, we propose a modification to Barrett's algorithm that leads to a significant reduction (25% to 75%) in multiplications and additions.

1. Introduction

One of the cornerstones of public-key cryptography is modular arithmetic, on which nearly all established schemes are based. An efficient software implementation of modular arithmetic is therefore desirable. While modular additions and subtractions are rather trivial cases, efficient modular multiplication remains an elusive target for optimization. The two most widely used algorithms for modular multiplication are Montgomery's method [1], and regular 'schoolbook' multiplication in combination with Barrett's reduction technique [2]. They are the main representatives of two classes of modular reduction algorithms, the left-to-right and right-to-left algorithms. For the sake of simplicity we will skip over most of the details and simply mention that Barrett's method progresses from left to right, using quotient estimation to subtract a suitable multiple of the modulus. In Montgomery's method the computation progresses from right to left. The least significant portion of an intermediate result is used to determine a multiple of the modulus, which when added, zeros out that least significant portion. Shifting to the right produces a shorter result congruent to the remainder. Variations of these two schemes were reported in [3, 4], but the essential idea is the same. A common thread to all schemes is the requirement to pre-compute certain values which depend on the size of the operands and/or the modulus. In addition, Quisquater's and Montgomery's method both require normalization and de-normalization steps, although their impact on performance is negligible as long as the cost can be amortized over sufficiently many modular reductions, which is indeed the case in most public-key algorithms. In their most naive version, both algorithms have a runtime in $O\left(n^2\right)$, which is due to the use of Schoolbook Multiplication. Better asymptotic behavior is possible if one applies sub-quadratic time algorithms such as Karatsuba-Ofman [5], however, this also depends on the relative cost of multiplications and additions. In the case of Montgomery's method, this requires the use of the separate operand scanning (SOS) variant of the algorithm [3]. This is because all other variants employ interleaving of multiplication and reduction steps, which make the recursive decomposition steps of the Karatsuba-Ofman method difficult or impossible.

Recently, Fischer and Seifert [6] made an interesting observation that there exists a duality between multiplication and modular reduction. Specifically, they show that there exists a duality between the well-known Booth-recoding technique for efficient multiplication and Sedlak's modular reduction method [7]. Based upon this observation and the premise that Booth multiplication is optimal, they conclude that Sedlak's method is also optimal. It is unclear to us, however, how Booth multiplication can be viewed as optimal, when other asymptotically faster algorithms exist, like the ones by Karatsuba-Ofman and Schönhage-Strassen [8].

18th IEEE Symposium on Computer Arithmetic(ARITH'07)
0-7695-2854-6/07 $20.00 © 2007 IEEE

In this paper we present a modification to Barrett's method which, in combination with the Karatsuba-Ofman algorithm, results in a significant reduction of the number of multiplications and additions on typical computer architectures.

2. Problem definition

Let N and M be multi-precision integers whose lengths are $2n$ and n bits, respectively. We want to efficiently reduce N with respect to the modulus M by computing the remainder of the division $\frac{N}{M}$. This need arises due to the fact that the most popular and widely used public-key algorithms require modular exponentiation, which is performed via a sequence of multiply and square operations, each immediately followed by reduction [9]. Thus, at any given point during exponentiation, we have 2 operands, A and B, that are n bits long and we generate the $2n$-bit product $N = AB$ using multi-precision multiplication. We then need to produce $R = N \bmod M$, another n-bit integer. However, one does not need to guarantee that $R < M$ at every intermediate step, only that R is an n-bit integer; the final R computed at the end of the exponentiation can then be reduced to be less than M. This allows fewer total computations to be performed.

Many cryptographic algorithms are defined over very large integer rings, where the operand sizes n are typically 512 to 4096 bits. We shall assume the use of a conventional processor which has a native word-size w, which is smaller than n. We also assume that on this hypothetical processor, ALU operations (i.e. additions, subtractions, shift operations) are less expensive in clock-cycles and / or power than multiplication for any given word-size w. Specifically, for the purpose of comparison, an ALU operation will cost one cycle and a multiply will cost m cycles. We will calculate the run-time of Word-Serial Montgomery (T_W) and our new algorithm, Modified Barrett (T_M), showing the latter to be the superior algorithm.

3. Montgomery Modular Multiplication

There are different forms of Montgomery that can be applied to the modular exponentiation problem. There are bit-serial architectures [10], where special purpose circuits perform multiplication and reduction simultaneously. However, our analysis assumes a general purpose processor and we therefore describe the Montgomery algorithm in terms of word-serial operations. Word-Serial Montgomery interleaves vector multiplication and reduction steps and has an asymptotic run-time of $O\left(\left(\frac{n}{w}\right)^2\right)$. The precise run-time is given by $\tilde{T}_W(n,w)$:

$$\tilde{T}_W(n,w) = \left(\tfrac{n}{w}\right)^2 (2m+4) + \left(\tfrac{n}{w}\right) m$$

In order to more easily compare with other methods, we let $k = \log_2\left(\frac{n}{w}\right)$. Then, equivalently, $T_W(k)$ is the number of cycles required to execute one n-bit Word-Serial Montgomery Modular Multiplication [3]:

$$T_W(k) = 4^k(2m+4) + 2^k m$$

4. Barrett Modular Multiplication

The Barrett reduction method requires the pre-computation of one parameter, $\mu = \left\lfloor \frac{2^{2n}}{M} \right\rfloor$, which does not change as long as the modulus remains constant. The reduction then takes the form $R = N - \left\lfloor \left\lfloor \frac{N}{2^n} \right\rfloor \frac{\mu}{2^n} \right\rfloor M$, which requires two n-bit multiplies and one n-bit subtract, leaving the total at three multiplications and one subtraction. Clearly, R is congruent to $N \bmod M$, and it can be shown that $R < 3M$ [2].

5. Karatsuba Multiplication

A simple approach to multiplying large integers is an $O\left(\left(\frac{n}{w}\right)^2\right)$ technique, Schoolbook Multiplication. Let $n = 2s$. Here, the n-bit multiply is broken down into four s-bit multiplies and two n-bit adds. An optimization to the carry handling problem is that at each recursion we do two $2^k + 1$ word adds and one branch (on carry), which almost always falls through.

$$\begin{aligned} N &= \left(2^s a_1 + a_0\right) \cdot \left(2^s b_1 + b_0\right) \\ &= 2^{2s} a_1 b_1 + 2^s \left[a_1 b_0 + a_0 b_1\right] + a_0 b_0 \end{aligned}$$

This technique can be applied recursively such that two n-bit integers can be multiplied in $T_S(k)$ cycles:

18th IEEE Symposium on Computer Arithmetic(ARITH'07)
0-7695-2854-6/07 $20.00 © 2007 IEEE

$$
\begin{aligned}
T_S(k) &= 4T_S(k-1) + 2^{k+1} + 3 \\
&= \sum_{i=0}^{k-1} 4^i \left[2^{k+1-i} + 3 \right] + 4^k m \\
&= 4^k(3+m) - 2^{k+1} - 1
\end{aligned}
$$

Karatsuba's algorithm can be used to trade multiplications for adds. Specifically, an n-bit multiply is reduced to three s-bit multiplies, two s-bit adds and three n-bit adds.

$$
\begin{aligned}
N &= \left(2^s a_1 + a_0\right) \cdot \left(2^s b_1 + b_0\right) \\
&= 2^{2s} a_1 b_1 + 2^s \left[\left(a_1 + a_0\right) \cdot \left(b_1 + b_0\right) - a_1 b_1 - a_0 b_0 \right] + a_0 b_0
\end{aligned}
$$

This technique can also be applied recursively such that two n-bit integers can be multiplied in $T_K(k)$ cycles. Notice that when s is much greater than the machine word-size, w, this tradeoff is quite sensible. However, when s is comparatively small, it could be less compelling. So, in this derivation, we assume that the last L recursions make use of Schoolbook Multiplication, then we optimize for L.

$$
\begin{aligned}
T_K(k) &= 3T_K(k-1) + 2^{k+2} + 8 \\
&= \sum_{i=0}^{k-L-1} 3^i \left[2^{k+2-i} + 8 \right] + 3^{k-L} S(L) \\
&= \left(\tfrac{4}{3}\right)^L 3^k m - 2^{k+3} - 4 + 3^{k-L+1}\left(2^L + 1\right)^2 \\
&= 3^{k+2} - 2^{k+3} - 4 + \tfrac{4}{3} 3^k m \quad \text{when } L=1
\end{aligned}
$$

The derivation has been omitted for brevity, however, if $2 \le m \le 9$, $L = 1$ in order to minimize $T_K(k)$. In the event that $m = 1$, only two percent more work is done via the use of $L = 1$ instead of the optimal point, $L = 2$. The astute reader may also notice that Karatsuba presents an awkward challenge concerning the handling of carries. The derivation above assumes the use of an important software technique; rather than multiply two integers of inconvenient size, we use branches and adds to assemble the correct product. For instance, suppose $N = \left(2^s a_h + a_l\right) \cdot \left(2^s b_h + b_l\right)$, where a_l and b_l are s-bit integers and a_h and b_h are booleans. Then, $N = 2^{2s} a_h b_h + 2^s \left[a_h b_l + b_h a_l\right] + a_l b_l$, which can be computed with an average of three branches and one s-bit add. Furthermore, this construction does not propagate inconveniently sized operands to lower level recursions; at each recursion, we call three multiply routines of power-of-two size and patch up the result, as above.

6. Modification to Barrett's Method

We propose a modification to Barrett's method that requires incrementally more pre-computation, but reduces the overall amount of multiplication and addition that is required. The modification is called folding. We will use it to partially reduce *N*, then use classical Barrett to complete the reduction step. We will describe our proposed technique with a single folding step, and then extend the technique to F foldings, showing the optimal point.

As with classical Barrett, the basic principle of this reduction method is to efficiently compute an estimate of the quotient $q \simeq \frac{N}{M}$ followed by the subtraction $R = N - qM$. The resulting bit-length of R is sufficiently close to the bit-length of the modulus, *M*, that the expected number of additional subtractions, $R = R - M$, necessary to ensure that $R < M$, is less than one.

Given an n-bit modulus, *M*, we require the pre-computation of two values which are constant with respect to the modulus: $M' = 2^{3s} \bmod M$ and $\mu = \left\lfloor \frac{2^{3s}}{M} \right\rfloor$. Notice that both values are by-products of the division, $\frac{2^{3s}}{M}$. The sequence of operations in Figure 1 illustrates the modular reduction of a $2n$-bit integer, *N*, by an n-bit modulus, *M*, according to our proposed modification of Barrett's technique with a single fold.

$$
N' = N \bmod 2^{3s} + \left\lfloor \frac{N}{2^{3s}} \right\rfloor M'
$$

$$
N' < 2^{3s+1}
$$

The result of the folding step is N', a $(3s+1)$-bit integer, calculated via two s-bit multiplications. This is convenient as it allows the Barrett reduction step to be computed via smaller multiplications.

$$
R = N' - \left\lfloor \left\lfloor \frac{N'}{2^{2s}} \right\rfloor \frac{\mu}{2^s} \right\rfloor M
$$

We see that R is calculated via three s-bit multiplications. So, with one fold, Modified Barrett Reduction requires only five s-bit multiplications. This is favorable when compared to the two n-bit multiplications that would be required using classical Barrett or Large-Digit Montgomery. Assuming the use of Karatsuba Multiplication, two n-bit multiplications would be equivalent to six s-bit multiplications, or 20% more work. It should be noted that Word-Serial Montgomery can not make use of Karatsuba Multiplication, putting it at an even greater disadvantage.

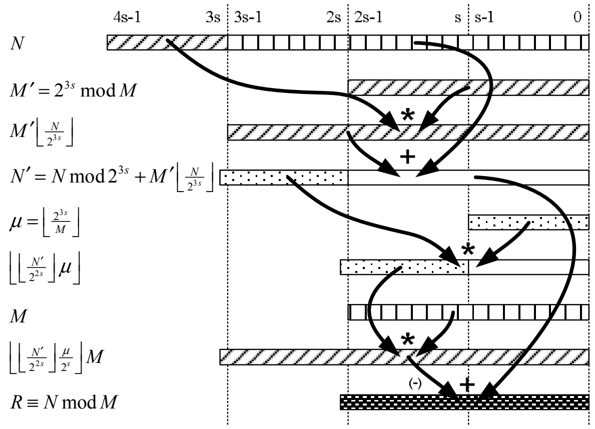
4s-1
3s
3s-1
2s
2s-1
s
s-1
0
N
$M' = 2^{3s} \bmod M$
$M'\left\lfloor \frac{N}{2^{3s}} \right\rfloor$
$N' = N \bmod 2^{3s} + M'\left\lfloor \frac{N}{2^{3s}} \right\rfloor$
$\mu = \left\lfloor \frac{2^{3s}}{M} \right\rfloor$
$\left\lfloor \left\lfloor \frac{N'}{2^{2s}} \right\rfloor \mu \right\rfloor$
M
$\left\lfloor \left\lfloor \frac{N'}{2^{2s}} \right\rfloor \frac{\mu}{2^{s}} \right\rfloor M$
$R \equiv N \bmod M$
*
+
*
*
(-)
+

Figure 1: Modified Barrett with Single Fold

7. Iterative Folding

The ambitious reader may notice that the folding process can be repeated up to $\log_2\left(\frac{n}{w}\right)$ times. It might be, however, that there is a point of diminishing returns on the application of folding. So, we assume that we will apply the folding technique F times, after which we will apply the classical Barrett technique to complete the reduction. For each of the F folds, we pre-compute each $M^{(i)} = 2^{\left(1+2^{-i}\right)n} \bmod M$, for $1 \le i \le F$. Then, for the final classical Barrett step, we pre-compute the quotient estimate, $\mu = \left\lfloor \frac{2^n}{M} 2^{2^{-F} n} \right\rfloor$, which is a $2^{-F} n$-bit integer. The steps are similar to the single-fold case, except with a loop in the folding step:

$$N^{(0)} = N$$

$$N^{(i)} = N^{(i-1)} \bmod 2^{\left(1+2^{-i}\right)n} + \left\lfloor \frac{N^{(i-1)}}{2^{\left(1+2^{-i}\right)n}} \right\rfloor M^{(i)} \ \forall 1 \le i \le F$$

$$R = N^{(F)} - \left\lfloor \left\lfloor \frac{N^{(F)}}{2^{\left(1+2^{-F}\right)n}} \right\rfloor \frac{\mu}{2^{2^{-F} n}} \right\rfloor M$$

Again, we have omitted the proof of convergence for brevity, so the thorough reader may verify that $R < (3+F)M$. The number of cycles required to implement an *F*-fold Modified Barrett Modular Multiplication is $T_B(k,F)$:

$$T_B(k,F) = \sum_{i=1}^{F} \left[2^{k-1} + 2^i \left(T_K(k-i) + 2^{k+1-i} + 8 \right) \right] + \left(2^F + 1\right) T_K(k-F) + 2^k + 1 + T_K(k)$$

In order to find the optimal value of *F*, we consider the limiting behavior of $T_B(k,F)$:

$$\lim_{n\to\infty} \frac{T_B(k,F)}{T_B(k,0)} = 1 - \tfrac{1}{2} 3^{-F} \left(2^{-F} - 1\right)$$

$$\frac{d}{dF} \lim_{n\to\infty} \frac{T_B(k,F)}{T_B(k,0)} = 2^F \ln \tfrac{2}{3} - \ln \tfrac{1}{3} = 0$$

$$F = -\log_2\left(1 - \log_3 2\right) \approx 1.44$$

Of course, *F* is necessarily an integer, so we evaluate the difference between the two nearest integers, since $\frac{T_B(k,F)}{T_B(k,0)}$ is uni-modal in *F*.

$$T_B(k,F=1) - T_B(k,F=2) = \tfrac{7}{2} n + 16$$

Observe that this result is positive and lacks the term m, the implication of which is that there is no difference in the number of multiplies. This is convenient, as we can always choose $F = 2$.

$$T_B(k,F=2) = 32 \cdot 3^{n-2} m + 8 \cdot 3^{n+1} - 7\left(5 + 2^{n+2}\right)$$

8. Results

We demonstrate the expected performance on three prominent architectures. First, we consider the Atmel ATmega128, which is a single-issue 8-bit microcontroller (AVR architecture), which requires two cycles per multiply. The ATmega128 has 128 KB of program memory and 4 KB of data memory, both of which are sufficient for modern key sizes. Second, we consider Intel's Microengine, MEv2, from the IXP family of network processors. The MEv2 is a single-issue 32-bit processor, which requires seven cycles per multiply. Finally, we consider Intel's 64-bit Itanium 2 processor, which can issue either six adds per cycle or four adds and two 64-bit integer multiplies per cycle. A constraint of the Itanium 2 is that the multiplier can accept new operands only every other clock cycle. So, the relative throughput of integer adds is five times that of integer multiplies.

The ratio of the runtime of Modified Barrett to Word-Serial Montgomery can be found in figure 2. Generally, Modified Barrett is superior to Montgomery in all cases, particularly as key sizes grow. Note that the relative runtimes represented here are limited to mathematical operations; loads, stores and other architecture-specific operations are not considered. We acknowledge that there may be some architectures which are particularly well-suited to

Word-Serial Montgomery. However, any architecture which has sufficient system resources to be dominated by multiplies and adds should render Modified Barrett superior. Indeed, the potential speedup of Modified Barrett is so significant that, at minimum, an architecture-specific evaluation would be justified.

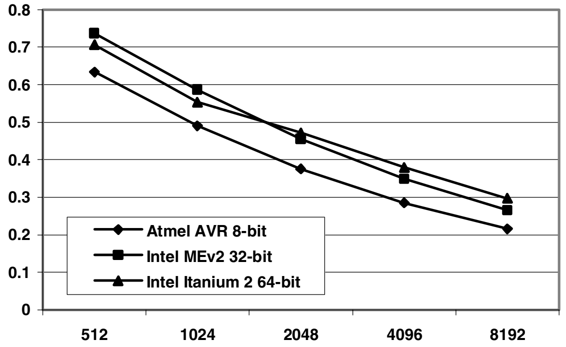

Figure 2: Ratio of Runtime: Modified Barrett / Montgomery ($\frac{T_B}{T_M}$) vs. Modulus Length (in bits) on three architectures.

9. Summary

The performance of Modified Barrett is similar on all three architectures, lending credence to the supposition that Modified Barrett is likely to be compelling for many applications. Relevant to current implementations, we see that the throughput / power savings of 1024-bit IKE exchanges would roughly double through the use of Modified Barrett. As key sizes grow to address heightened security needs, Modified Barrett continues to pay dividends; note that the throughput and power savings of 4096-bit exponentiations would roughly triple. Also, hardware implementations of Modified Barrett could be designed using off-the-shelf multipliers and adders, obviating the need for the custom logic one might find in a Bit-Serial Montgomery implementation.

In this paper, we have done two things. First, we have mitigated the overhead that has previously rendered Karatsuba Multiplication unattractive to a general programming environment with clever carry handling. Second, we have introduced a new pre-processing step to Barrett Modular Reduction which is computationally efficient. Finally, we have shown that Modified Barrett has a sufficiently significant algorithmic advantage that it should be evaluated in any application that requires modular multiplication.

10. References

[1] P. L. Montgomery, *Modular Multiplication without Trial Division*. Mathematics of Computation, vol. 44, no. 170, pp. 519—521, April 1985.

[2] P. Barrett, *Implementing the Rivest Shamir and Adleman Public Key Encryption Algorithm on a Standard Digital Signal Processor*. Advances in Cryptology – CRYPTO'86, Santa Barbara, Calif., AM Odlyzko (Ed.), LNCS 263, Springer, 1987.

[3] C. K. Koç, T. Açar and B. S. Kaliski Jr., *Analyzing and Comparing Montgomery Multiplication Algorithms*, IEEE Micro, June 1996.

[4] J.-J. Quisquater, *Encoding system according to the so-called RSA-method, by means of a microcontroller and arrangement implementing this system*. U.S. Patent #5,166,978, November 1992.

[5] A. Karatsuba and Y. Ofman. *Multiplication of Multidigit Numbers on Automata*. Soviet Phys. Doklady, 7(7):595–596, January 1963.

[6] W. Fischer and J.-P. Seifert, *Duality between Multiplication and Modular Reduction*, IACR ePrint Archive, 2005.

[7] H. Sedlak, *The RSA Cryptography Processor*. Advances in Cryptology – Eurocrypt'88, Amsterdam. D. Chaum and W.L. Price (Eds.), LNCS 304, Springer, 1988.

[8] A. Schönhage and V. Strassen. *Schnelle Multiplikation grosser Zahlen*. Computing, 7:281–292, 1971.

[9] J.-F. Dhem, *Design of an efficient public-key cryptographic library for RISC-based smart cards*. Ph.D. Dissertation, Université Catholique de Louvain, May, 1998.

[10] A. F. Tenca and Ç. K. Koç, *A scalable architecture for Montgomery multiplication*. In Proc. 1st Int. Workshop on Cryptographic Hardware and Embedded Systems (CHES'99), LNCS 1717, p. 94 ff., Springer, Heidelberg, August, 1999.