Software Optimization of Decorrelation Module

Fabrice Noilhan

Université Paris-Sud, LRI
Bât. 490, 91405 Orsay, Cedex, France
Fabrice.Noilhan@lri.fr

Abstract. This paper investigates software optimization of special multiplication. In particular we concentrate on $ax + b \bmod 2^{64} + 13 \bmod 2^{64}$ which is the bottleneck operation in the DFC cipher. We show that we can take advantage of the language and architecture properties in order to get efficient implementations.
In this paper we use the ANSI C and the Java languages. We also investigate assembly code, and data structure alternatives. Finally, we show that we can also use floating point arithmetic.

1 Introduction

Several cryptographic algorithms require that some particular multiplication is optimized. In particular, the DFC AES candidate [1] was believed to be substantially slower than the others because its main operation $ax + b \bmod 2^{64} + 13 \bmod 2^{64}$ was believed to be necessarily slow. In this paper we concentrate on optimization techniques for this operation. The results are of course not restricted to DFC since other cryptographic algorithms use this kind of primitive. For instance, the MMH MAC algorithm [3] uses the $\sum_{i=1}^{k} m_i x_i \bmod (2^{32} + 15) \bmod 2^{32}$ function, the Shazam [5] algorithm uses the $(x + k)^2 \bmod p \bmod n$ operation.

We will first introduce how to do a division-less modular reduction. Then, we will see what are the choices to implement the multiplication and this modular reduction. We will point out some security issues about the implementation itself since there are many ways to implement it and see that most concerns can be solved using proper operations. We finally will show what are the best choices to get optimal performances with generic ANSI C, 64-bit C, assembly language and Java on Alpha, Pentium II, UltraSparc processors and on IA64 architecture.

2 Calculation of $ax + b \bmod 2^{64} + 13 \bmod 2^{64}$

The multi-precision multiplication is best implemented in a straightforward manner since optimizations such as Karatsuba do not seem worthwhile for such small operands.

As a division is rather slow, we must have another method to do modular reduction. We will use the following method:

Let $P = ax + b$. Note that since a,x and b are 64-bit numbers, P is a 128-bit number ($ax + b \leq 2^{128} - 2^{64}$). We first write

$$P = Q2^{64} + R$$

where R is the remainder of the Euclidean division of P by 2^{64}. It follows that

$$P = Q(2^{64} + 13) + R - 13Q$$

and this is equal to $R - 13Q$ modulo $2^{64} + 13$.

The subtraction is a problem since it can lead to two different cases: $R - 13Q \geq 0$ and $R - 13Q < 0$. Dealing with negative values is tedious since we have to take care about timing attacks and use sometimes back and forth conversions between signed and unsigned integers (we want to use all register bits for multiplication). This can be avoided while splitting the value into smaller words but, usually, it is not effective.

As we are doing arithmetic modulo $2^{64} + 13$, we can use the bitwise complement to do subtraction:

$$\begin{aligned} P' = R - 13Q &= R + 13(2^{64} - 1 - Q) - 13(2^{64} - 1) \mod 2^{64} + 13 \\ &= R + 13(2^{64} - 1 - Q) + 182 \mod 2^{64} + 13. \end{aligned}$$

$2^{64} - 1 - Q$ is the 64-bit bitwise complement of Q. The result is always positive and is most of the time greater than $2^{64} + 13$. Thus, we can perform a similar reduction for P': $P' = Q'2^{64} + R'$, and $Q' \leq 14$. Then, we can use a small table to compute the final values.

3 Some Possibilities

3.1 Multiplication

Implementations should use arithmetic on numbers of the largest size that is efficiently available on the target processor.

The key factor for speed is the multiplication of two 64-bit quantities yielding a 128-bit result. We have to do a number of multiplications which depends on the multiplier of the processor as shown in the table 1.

Table 1. Number of multiplications required

operands(bits)	result(bits)	# multiplications
64	128	1
32 or 64	64	4
16 or 32	32	16
8 or 16	16	64

We also have to add the resulting values and the number of additions depends on the size of the registers used (cf table 2).

Table 2. Number of additions required

registers(bits)	# additions
64	4
32	18
16	88

The total cost of additions is typically less than the cost of the multiplications but is not negligible. Thus, optimizations of this part of the algorithm are of prime importance, especially when dealing with carries.

It is sometimes worth doing more operations on smaller operands, since they may be faster. Tables 1 and 2 should give a fair estimation of the cycle counts.

3.2 Modular Reduction

Before the modular reduction, we need to add the constant b. This can be done either before or after the multiplication. The choice is generally made by the language used: a low-level language with "add with carry" implies usually a straightforward implementation of the multiplication. Otherwise, adding the constant in the same time as we multiply can save some cycles.

Then, we can implement the modular reduction in many different ways. First, we have to split the 128-bit result of the multiplication into many words. On the one hand, if we use words of the maximum size available, then we will not have any room left to store carries so that we will need to propagate them. On the other hand, using words of smaller size will imply more operations. Depending on the processor and its parallelization level, one solution is faster than the other.

In the reduction itself, the multiplication by the constant 13, as explained in the previous section is usually optimized by the compiler. This is not the case on UltraSparc (Sun WC 5.0 compiler) or in Java; we have to do it manually with shifts and adds. The second step of the reduction implies another multiplication of a low operand. On most processors, this operation is faster using a small lookup table. Not only it is faster but it avoids some problems of timings attacks with such small operands.

4 Security Issues

Since the implementation of the algorithm can be done in many distinct ways, one has to take care about security issues of the implementation itself.

First of all, the multipliers on some chips can compute the product of small operands in fewer cycles than for large operands. This feature may make timing attacks possible, as is the case with most algorithms using integer multiplications (e.g. RC6, Mars).

Recently, Harvey [2] has noticed that an attack of DFC can be made on careless implementations. He supposes a different approach than the one we present here. Our implementations can be easily made resistant to timings attacks (except on the multiplier itself), and rare code paths are easy to test.

Care must be taken with the propagation of carries. On some chips, the fastest implementation uses branches and thus is vulnerable to timing attacks. On low end processors, the cost of such a protection is noticeable since there are many carries to propagate. But on Pentium Pro for instance, the cost is only about 40 cycles.

5 Dedicated Optimizations

5.1 ANSI C

Writing portable ANSI C code entails that you do not know anything about the representation of the objects. When you need a 32-bit unsigned integer, you have to use an `unsigned long`.

So, using 32-bit integers for both multiplication and additions, an implementation of the round function requires at least 16 multiplications and 18 additions.

On most processors, this implementation will not have a high speed compared to an assembly coded function: the processor may be able to deal with larger registers or the processor may have some particular properties (e.g. a larger multiplier, or an add-with-carry opcode). Those characteristics can not be used in a portable ANSI C code.

Still, an ANSI code should compile and produce the same results whichever system and (ANSI) compiler you use. The tradeoff between portability and efficiency is generally costly.

As regards to the modular reduction, since we do not know anything about the processor, it does not make much sense to choose between alternatives. The addition of b can be made within the multiplication. We should use 32-bit words to do the modular reduction, since it would otherwise require too many operations.

5.2 C with 64-Bit Integers

A new norm, called C9X, will allow to use 64-bit integers (such as in JAVA for instance). Indeed, it will not only help 64-bit processors to give their full power, but also it will be helpful for 32-bit processors (such as Pentium II) which have a larger multiplier.

Using 64-bit `unsigned long long`, one can use only 4 multiplications to compute the 128-bit multiplication of the round function. One should switch back to the 32-bit integers next, since other operations on 64-bit integers are emulated by the compiler. Thus they yield to poor performance.

For native 64-bit processors, not only does the use of 64-bit types reduce the number of operations, but these operations are faster: 32-bit operations are often emulated by the processor (using integer masks for example) and are slower than 64-bit opcodes.

In this code, we can use 32-bit words, even on 64-bit processors, to do the modular reduction. With 32-bit processors, it is obvious. With 64-bit processors, many instructions are executed each cycle. So you may group into 32-bit words for free if it helps parallelization. The main advantage is that the propagation of carries is eased since we do them only once at the end of the computing (they are stored in the 32 most significant bits of the registers during the calculation).

5.3 Alpha

The Alpha 21164 has a multiplier which takes two 64-bit inputs and provides the least significant or most significant half of the 128-bit result after 12 or 14 cycles respectively. Also, the multiplications can overlap and can run in parallel with other operations.

If we use portable ANSI C code, which only guarantees arithmetic up to 32 bits, then performance is relatively poor on 64-bit chips. We have to break numbers into 32-bit pieces and cannot use any 64-bit capabilities, in particular fast multipliers such as that on the 21164.

ANSI C permits implementations to provide 64-bit arithmetic however, and by taking advantage of this we gain a lot of speed. We still cannot get the most significant half of a 64×64-bit product however. On Alpha we use the assembly language instruction, `umulh` from C (by implementation-dependent methods) to attain the best performance on this architecture.

Since the multiplication is atomic, the addition of $b + 182$ is made after it. We use 64-bit words, the first multiplication by 13 is done via shifts and the second one is left to the compiler (which implements it correctly). This can be done efficiently in C.

More extensive use of assembly language does not appear to yield any significant improvement.

5.4 Pentium II

The Pentium II is a 32-bit processor with a multiplier which takes two 32-bit inputs and returns the 64-bit result. The multiplication instruction is fast: it only takes 4 cycles. So, the entire multiplication and the modular reduction should be fast as well.

However, the expected speed is not achievable in C. Using only ANSI C, we are not able to use the multiplication with 64-bit result. Even if we do use 64-bit integers via the `long long` type (which will be part of C9X), compiled code does not use addition with carry instruction and it does a lot of unnecessary data movement between registers and memory.

To implement the integer multiplication in assembly, we take advantage of two operations: addition with carry and 32-by-32 multiplication.

The modular reduction is done with 32-bit words, and use a table lookup. Once again, we need the addition with carry, so that it is also done in assembly language.

When the whole computation is done in registers using assembly language, we get the expected speed of the function.

5.5 UltraSparc

The UltraSparc is a 64-bit processor. Until recently, it could not be used as such in C since there was no compiler for 64-bit mode. Sun's new C compiler 5.0 handles 64-bit integers and performs well on 64-bit C code.

The multiplier takes two 64-bit inputs and computes the least significant half of their product. Unlike the situation on Alpha, there is no method to get the most significant half. Thus, we have to do four multiplications to get the full result. These multiplications are rather slow: each of them takes about 20 cycles so that the time for the entire multiplication is large. The time for modular reduction is insignificant in comparison.

In order to achieve better results, we can use the Floating Point Unit. The FPU has a double precision multiplier which we will use in a slightly unusual way. Since the FPU uses the IEEE 754 [1] representation of numbers, we can use 52 significant bits with double precision floating-point numbers. This means that we can multiply 24-bit values and add several of the results without any round-off error occurring. Thus, we can do a 64×64-bit product using nine 24×24-bit multiplications (and some additions). Alternatively we can do eight 16×32-bit multiplications. These methods are faster than using four integer multiplications.

The only problem is to convert from integers to floats. When done with casts in C, it uses a function of the C library and is very slow. When done in assembler using the `FiTOd` instruction, it is not much better. For reasonable speed we have to do it manually via a bit mask and an addition. Similar tricks are used to convert from floating-point numbers back to integers in order to do the modular reduction.

This enables us to achieve better performances for the multiplication than the standard 64-bit C code.

The modular reduction is implemented with 32 bit words and carries are stored into high order bits of the registers. Multiplications by 13 are all made using shifts and additions.

5.6 IA64 Architecture

Intel has recently unveiled specifications of its next architecture, called IA64. This enables us to estimate the cycle counts of the Merced processor.

The Merced is a 64-bit processor. We have no idea at the moment whether compilers will be able to deal with the full set of instructions so that we will

[1] ANSI/IEEE Standard 754-1985: Standard for Binary Floating Point Arithmetic

only consider assembly language. The key point is that IA64 has a full 64-bit to 128-bit unsigned integer multiply so it will end up in the fast category. The instruction `xma` is an integer $a * x + b$ (which can never overflow 128 bits).

Terje Mathisen has written an IA64 implementation of the round function and gets a round timing of 30 cycles with some more or less obvious possibilities to save a few more cycles. Of these 30 cycles, 10 are taken by a pair of sequential `xma` operation, but the second can be handled with integer code instead, since the multiplier is small and known (13).

The one possible problem is that integer multiplication uses the floating point registers, so the (currently unknown but believed to be 1 cycle) time to convert back and forth between integers and floating point mantisses is in addition to the two `xma.u` operations needed for the low and high halves of the result. For best performance, all the integers and floating point constants need to be placed in registers before starting the inner loop.

As predication replaces branches in the carry propagation, there should be no possibility for a timing attack based on key or data values.

Thus, the 240 cycles count compares well to the 231 cycles on a 21264 Alpha. This answers to the criticism that the decorrelation module should be slow on Merced and shows that some non-trivial optimizations of the code can give a huge improvement of speed. We also note that the numerous registers and the multiplier should enable a fast RSA implementation.

5.7 Java

There is a very simple way to implement the multiplication and the modular reduction in JAVA, using BigIntegers. Unfortunately, this slows down the speed dramatically. It has the advantage of only taking two or three lines of code and can provide a reference but not an optimized implementation.

Java provides a 64-bit integer data type, which is always signed. Anyway, the signed-ness can just be ignored in the arithmetic operations we need: additions, subtractions and multiplications are defined in the standard as if they were modulo 2^{64} and bitwise logical operations use the sign bit as in normal twos-complement representation. There is an unsigned right shift operator, as well as a signed one. The only restriction is that we can neither use comparisons (which could have been useful to propagate carries) nor use signed divisions (which we do not need anyway). These characteristics are described in the Java Virtual Machine specifications [4].

Thus, the implementation is essentially the same as a 64-bit C version. Since we cannot do casts (even using "assembly" Java bytecode) and since we cannot use comparisons, the implementation is naturally resistant to endianess issues and to most timing attacks (except for the multiplications).

Even if one could produce specific Java code for a processor, it does not gain very much. Optimized versions for Pentium II and UltraSparc uses the same tricks as optimized C codes. Most of the optimizations dedicated to the processors are done in JIT compilers. Even hand-written bytecode, which should produce faster code, does not have a noticeable speedup. The reason why we can not do optimizations is that the set of opcodes is very small and the whole job is given to the JIT compiler which optimizes the code for the given processor. Since 64-bit operations are part of the language, they are well optimized by the compiler, which reorders and expands some instructions.

Some errors should be avoided to help the compiler: splitting into functions as we can do in C, using a multiplication by 13 on UltraSparc processor. But the Java compilers are relatively new and improve quickly.

With the development of Web services and online payment, these JAVA implementations become all the more important and huge improvements of speed have been done during the past year: JIT compilers and now HotSpot technology should give a speed equivalent to C++ according to Sun Microsystems (that is roughly the speed of C in cryptography).

6 Conclusion

When going to results, there are two facts to outline: one should use the largest integers available and the round function is not always well optimized by compilers.

On Alpha, on account of the lack of a C instruction to get the most significant half of the multiplication, 200 cycles are lost. The same problem may exist on IA64. On Pentium II, compilers do not achieve the expected speed because there are too few registers. Results on the UltraSparc are disappointing as a result of the slow multiplier.

In the following table 3, we compare ANSI C portable code and 64-bit C code to the best implementation available in order to show the importance of the optimizations.

Table 3. Number of cycles for DFC

Processor	JAVA	ANSI C	64-bit C	Best
Alpha 21164	n/a	2562	526	310 (ASM)
Pentium II	1481	2592	1262	392 (ASM)
UltraSparc	4087	4160	875	775 (C with floats)
Alpha 21264	n/a	n/a	335	231 (ASM)
IA64	n/a	n/a	n/a	240 (estimated)

With generic ANSI C code, DFC is one of the slowest AES candidates on all platforms. Using assembly language, it becomes the fastest on Alpha processors and among the fastest on Intel Pentium II and Merced processors.

We have seen what are the best solutions on various microprocessors and languages. Harvey [2] thought that "correct implementations may be difficult to achieve". We have shown that correct and fast implementations can be easily made.

As the cost of this decorrelation module is nearly the same as the cost of the multiplication, it could be used as a plug-in in many other algorithm without significant decrease of the performances. The drawback is that its optimization requires access to some low-level instructions (add with carry, most significant bits of the multiplication) which are generally not available in a high-level language such as C.

Acknowledgments

We thank Serge Vaudenay, Robert Harley and Terje Mathisen for comments and suggestions on this work. We also thank Dominik Behr, Robert Harley, Danjel McCougan, Terje Mathisen and David Seal for their implementations of DFC from which most of the work is derived.

Java is a trademark of Sun Microsystems, Inc. in the United States and other countries.

References

1. H. Gilbert, M. Girault, P. Hoogvorst, F. Noilhan, T. Pornin, G. Poupard, J. Stern, S. Vaudenay. Decorrelated Fast Cipher: an AES Candidate. (Extended Abstract.) In *Proceedings from the First Advanced Encryption Standard Candidate Conference*, National Institute of Standards and Technology (NIST), August 1998.
2. Harvey. The DFC Cipher: an attack on careless implementations In *Proceedings of the second AES Workshop*, 1999
3. Haveli, Krawczyk MMH: Message authentication in software in the Gbit/sec rates In *Proceedings of the 4th Workshop on Fast Software Encryption*, 1997
4. Lindholm, Yellin The Java[tm] Virtual Machine Specification, Second Edition Sun Microsystems, ISBN: 0-201-43294-3
5. Patel, Ramzan, Sundaram. Towards Making Luby-Rackoff Ciphers Optimal and Practical To appear in *Proceedings of the 6th Workshop on Fast Software Encryption*, 1999