IMPLEMENTATIONS OF MONTGOMERY MULTIPLICATION ALGORITHMS IN MACHINE LANGUAGES

Emre Çelebi, Mesut Gözütok
Dept. of Information Security
Havelsan Corporation
Ankara – TURKEY
ecelebi, mgozutok@havelsan.com.tr

Levent Ertaul
Dept. of Mathematics & Computer Science
California State University, EastBay
Hayward, CA, USA.
levent.ertaul@csueastbay.edu

Abstract- *Software implementations of public-key algorithms such RSA and Diffie-Hellman are often desired because of their flexibility and cost effectiveness. In order to obtain the required level of performance on a selected platform, developers turn to implement efficient algorithms in machine (assembly) languages for basic (kernel) operations. Among these basic operations, modular multiplication and exponentiation operations play an important role. This study concerns with fast software implementations of the Montgomery multiplication algorithms. Montgomery multiplication algorithms, which are the most popular algorithms used in public-key cryptography, serve as efficient algorithms for modular multiplication and exponentiation operations. In this paper implementations of five well known algorithms of Montgomery multiplication are given. Algorithms are implemented in assembly languages of Intel Pentium family and Sun Sparc family microprocessors. In order to get comprehensive performance results, they run on Windows 95, Windows 98, Windows NT 4.0, and Solaris 2.5.1 operating systems. The performance results are summarized with a comparison of the algorithms based on their software performances.*

Index Terms — Montgomery Multiplication, RSA and Diffie-Hellman Cryptosystems, Public-key cryptography.

1 Introduction

The design and evaluation of a cryptosystem is a special topic which requires advanced knowledge of combinatorial mathematics, number theory, abstract algebra and theoretical computer science. In addition to that, high-speed implementations of cryptosystems are always an issue in the design and evaluation of a cryptosystem. Particularly, many public-key cryptosystems [1] [2], such as RSA [3] and Diffie-Hellman [4] require fast, efficient implementations. These cryptosystems employ basic modular operations (addition, multiplication and exponentiation) with large integer numbers. These integer numbers are generally in the range of 1024 to 2048 bits, if not bigger, in many today's public-key cryptosystems. This indicates that efficient implementations for these basic modular operations are critical for the overall performance of many public-key cryptosystem. The performance criteria for the modular operations in these public-key cryptosystems may be vital since modular operations with such large numbers usually cause timing variations. This unintentional timing characteristic can be used by an attacker to mount timing attacks against the public-key cryptosystem to discover the entire private secret key [5]. These considerations turn developers into implementations of high-speed and space efficient algorithms [6] [7] [8].

By choosing proper mathematical algorithms for implementation of basic modular operations, especially for multiplication and exponential operations that are clearly the most time consuming operations, overall performances of much public-key cryptosystems can be improved considerably and this way the non-fixed execution time can be avoided.

There exist some mathematical algorithms to perform fast and efficient modular multiplication and modular exponentiation operations [9]. In this study, one of these efficient algorithms called as Montgomery multiplication algorithms, introduced by Peter L. Montgomery [10], is investigated and software implementations of these algorithms are performed on different platforms. Montgomery multiplication algorithms constitute the core of the modular exponentiation operation used in many public-key algorithms. Particularly, Diffie-Hellman and RSA private key operations consist of computing $C = M^a \bmod n$, where M is a massage, n is modulus which is public and a *is* the secret private key. Modular reduction steps in this operation usually cause most of the timing variations. Montgomery multiplication eliminates the mod n reduction steps and as a result, tends to reduce the size of the timing characteristics. Also Montgomery multiplication algorithms speed up the computation operation for modular exponentiations and modular multiplications [11].

In general, Montgomery multiplication algorithm computes the Montgomery product as defined by

$$MonPro(a, b) = a.b.r^{-1} \pmod n$$

where the integers a and b are smaller than the modulus n and r is relatively prime number to n ($gcd(n, r) = 1$). Even if the algorithm works for any randomly chosen r, it is more reasonable to choose r as a power of 2. This is because that the operations, particularly divisions with power of 2 are fast operations on general-purpose microprocessors, resulting in faster implementations. The modulus value n then can be taken as an odd integer to satisfy the condition $gcd(n, r) = 1$.

The computation of $MonPro(a^-, b^-)$ is achieved as follow,

Function $MonPro(a^-, b^-)$
Step1. $t := a^- . b^-$
Step2. $u := (t + [t . n' \bmod r] . n) / r$
Step3. *if* $u \geq n$ *then return* $(u-n)$ *else return* u

As seen from the *MonPro ( )* function above, a^-, b^- numbers which represent the n-residues [12] of the numbers a and b should be precalculated and also the number n' with property $r.r^{-1}-n.n'=1$ should be precalculated. The integers r^{-1} and n' can be both calculated by using the Extended Euclidean algorithm [13].

In modular multiplication with $c = a.b \pmod n$ the division by n operation is required [14] [15]. In Montgomery multiplication the division by r in Step2 is a fast operation since r is a power of 2. This shows that Montgomery multiplication is a faster algorithm than the ordinary multiplication algorithm. However *MonPro ( )* function is useful when there is a need for repeated modular multiplication operations. Otherwise conversion from an ordinary residue to an n-residue and conversion back from an n residue to an ordinary residue and computation of n' operations are time consuming for a single modular multiplication operation.

In this paper, the five well-known algorithms of Montgomery multiplication are studied and these algorithms are implemented in the assembly languages of Intel Pentium family and Sun Sparc family microprocessors. The main objective of the study focuses on finding the most efficient algorithm, between these five algorithms, that gives best software performance results. In order to obtain comprehensive performance results, algorithms are run on Windows 95, Windows 98, Windows NT 4.0 and Solaris 2.5.1 operating systems. These algorithms are namely [23],

- Separated Operand Scanning (SOS)
- Coarsely Integrated Hybrid Scanning (CIHS)
- Coarsely Integrated Operand Scanning (CIOS)
- Finely Integrated Operand Scanning (FIOS)
- Finely Integrated Product Scanning (FIPS)

These algorithms can be organized basically based on two factors. The first factor is the one whether algorithms perform *multiplication* and *reduction* operations by integrating them or by separating them. In the separated approach, algorithms firstly perform *multiplication* and then perform *reduction* operations. In the integrated approach, algorithms integrate *multiplication* and *reduction* operations. This integration can be either coarse-gained or fine-gained; depending on how often algorithm integrates *multiplication* and *reduction* operations.

The second factor is the general form of the *multiplication* and *reduction*. One form is the operand scanning and the other form is the product scanning. In the operand scanning method, an outer loop moves through the words of one of the operands, on the other hand in the product scanning method, the loop moves through the word of the product itself. This second factor is independent from the first factor.

In the next section implementation methods of the Montgomery multiplication algorithms are presented. In section 3, Implementation results are discussed and finally the conclusions are given in section 4.

2 Implementation of the Montgomery Multiplication Algorithms

Almost all the implementations of basic functions in many cryptosystem are performed in assembly languages in order to take the advantage of the specific architectural properties of the processor [16] [17]. The Montgomery multiplication algorithms listed above, require operations with elements of a large finite group and need to be optimized on the chosen platform in order to obtain the desired high-speed performance. Therefore, in the implementations of these algorithms, assembly languages of Intel Pentium and Sun Sparc family processors are used.

Each algorithm is implemented as '**in-line**' assembly functions with specific assembly language of related microprocessors. Particularly, on Pentium platforms, algorithms are written as C functions with in-lined assembly and on Sparc platforms they are written as complete assembly functions since in-line assembly is not flexible due to inability to access the C variables within the in-lined assembly code in UNIX (Solaris 2.5.1) machines.

Algorithms require nearly same or slightly different amount of memory spaces during the execution. This

results from the general form of the algorithms and from the way they perform Montgomery multiplication.

The algorithmic codes for the implemented Montgomery multiplication algorithms are presented in Table 1 and in Table 2.

Table 1. Algorithmic codes of SOS, CIHS and CIOS

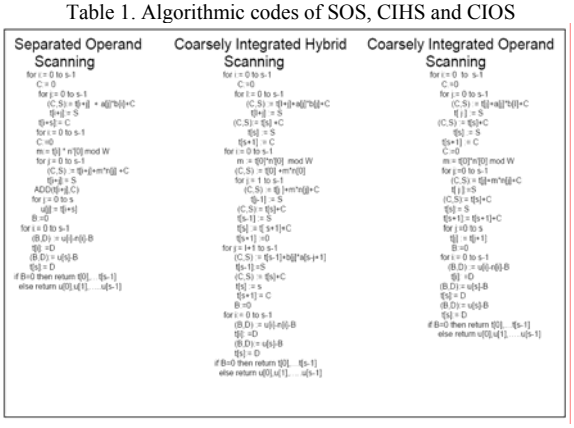
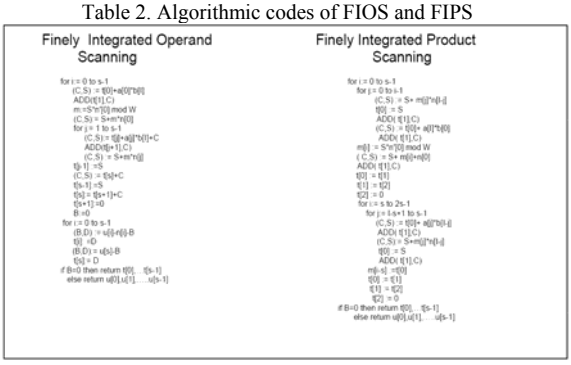


Table 2. Algorithmic codes of FIOS and FIPS



After investigating the algorithmic codes of the Montgomery multiplication methods, it can be seen that some algorithms are similar to the other one in a way they perform Montgomery multiplication such as CIOS and FIOS methods. This similarity comes from their embedded shifting and interleaving the products.

The five algorithms perform the Montgomery multiplication with the combination of the factors mentioned before. For example, Separated Operand Scanning (SOS) method performs Montgomery multiplication by separating the *multiplication* and *reduction* operations. In this method, first the product *a.b* is computed with the pseudo code giving below.

```
for i := 0 to s-1
C:=0
for j:= 0 to s-1
(C,S):=t[i+j]+a[j].b[i]+C
t[i+j]:=S
t[i+s]:=C
```

where the variables *a*, *b* and *t* are arrays of type word, *s* is the number of words of these arrays, C and S are one word variables and C represents carry operations. The computed product *a.b* is hold in the memory space of *t* array where *t* is initially assumed to be zero. Since both *a* and *b* numbers are *s*-word numbers, their product will result in a *t* array number with 2s-word length (2**ws**). This operation is illustrated in Figure 1 for input arrays whose *s=3*,

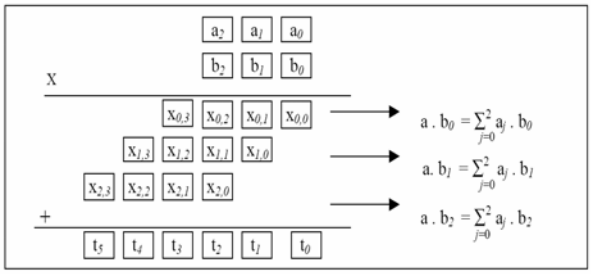


Figure 1. The Partial Products Generated in a 3x3 Array Multiplication and Computation of the Product *a .b*

In order to multiply two unsigned integers, partial products are accumulated in the 2sword array *t* as shown above. The accumulation requires the addition of the previous product *ti, j*, to the partial product *xi, j*. In order to add the previous carry, computed product and the accumulated result, the following operation is performed,

$$(C\,i,j\,,ti\,,j\,) = a\,i\,.\,b\,j + C\,i,j\text{-}1 + t\,i,j$$

This operation can be accomplished using the code segments shown in Table 3 for Intel assembly and for Sun Sparc assembly languages respectively.

Table 3. Implementation of Multiplication Operation for Intel and Sparc Processors.

A- Multiplication Operation on Pentium	B - Multiplication Operation on Sparc
mov eax, dword ptr [a]	mov %o1, [%i1]
mov ebx, dword ptr [b]	mov %o2, [%i2]
mov ecx, dword ptr [t]	mov %o3, [%i3]
mul ebx	mulx %o1, %o2, %g1
add eax, ecx	add %g1, %o3, %o3
adc edx, 0	stuw %o3, [%i3]
mov dword ptr [t], eax	retl
mov dword ptr [C], edx	srlx %o3, 32, %o2
	stuw %o2, [C]

After finding the *t* value, SOS algorithm performs the second operation in *MonPro ( )* function and it computes the *u* using the formula;

$$u := (t + m\,.n\,) / r$$

where $m := t.n' (mod\ r)$. In order to compute *u* value, the product *m.n* is computed with multiplication as shown above and this product is added to the *t* array. This operation is shown with next pseudo code below;

```
for i:= 0 to s-1
C:=0
m:=t[i]. n' mod W
for j:=0 to s-1
(C,S):= t[i+j]+m.n[j]+C
t[i+j]:= S
ADD(t[i+s],C)
```

The ADD function above performs a carry propagation adding C to *t* array. This addition starts from its given argument and propagates it until no further carry is generated. This ADD function is also present in the FIOS and FIPS algorithms as seen in Table 1. In Table 4, the codes for performing ADD function are shown for both Intel and Sparc assembly languages.

Table 4: Implementation of ADD Function for Intel and Sparc Processors.

C- ADD Function Operation on Pentium	D- ADD Function Operation on Sparc

Addfunc: mov eax, dword ptr [t [i]]
mov ebx, dword ptr [C]
addc eax, ebx
mov dword ptr [t [i]], eax
add i, 4
comp ebx, 0
jnz Addfunc
	Addfunc: mov %o1, [%i3]
	lduw %o2, [C]
	addcc %o2, %o1, %o1
	stw %o1, [%i3]
	addcc %g0, %g0, %o2
	inc %i3, 4
	comp %o2, 0
	brnz Addfunc

Finally the resultant *t* value is divided with $r=2^{ws}$ to find the *u* array. The division of computed *t* with $r=2^{ws}$ is accomplished by ignoring the lower *s* words of *t* which is shown with the following pseudo code,

```
for j:=0 to s
u[ j]:=t[i+s]
```

After this shift operation, the *u* array occupies *s+1* word-length. Finally the third and last operation in *MonPro( )* function is carried. The multi-precision subtraction in Step3 of *MonPro( )* is performed to reduce *u* if necessary.

```
B:=0
for i:=0 to s-1
(B,D):=u[i]-n[i]-B
t[i]:=D
(B,D):=u[s]-B
t[s]:=D
if B=0 then return t[0],t[1],....t[s-1]
else return u[0],u[1],...,u[s-1]
```

In above code, B indicates borrow operations. This last operation, performs a subtraction of modulus from the computed *u* value, is the same in all implemented algorithms. This is also shown in the in Table 1 and 2. [for a detailed computation of this final subtraction refer to [18]].

On Pentium platforms, the efficiency and the performance of an algorithm depends on a number of factors, such as parameter size, time-memory tradeoffs, processing power availability, software optimization and the properties of the algorithms. Also an efficient assembler software development requires many considerations about the algorithms with a full understanding of the microprocessor architecture, like pipeline structures, properties of the assembler instructions, the rules of the instruction issuing, and alignment, the operation multiple functioning units and cache and memory structures [19].

During the implementation process of the algorithms on Pentium and on Sparc platforms, some software optimization techniques are used to improve the performances of the algorithms. Even if the algorithms perform Montgomery multiplication differently, the main operations involved in the programs are multiplications, additions, addressing and indexing of the arrays. In each algorithm the number of these additions, multiplications and memory read-write operations change slightly. As an example of this operation-counting process, the number of each operation in the first *j*-loop of the SOS algorithm is calculated and result is given in Figure 2.

Operation Statement	Multiplications	Adds	Reads	Writes	Iterations
for j:=0 to s-1	-	-	-	-	-
(C,S):=t[i+j]+a[j].b[i]+C	1	2	3	0	s^2
t[i+j]:=S	0	0	0	1	s^2
Total	$1s^2$	$2s^2$	$3s^2$	$1s^2$	

Figure 2: Calculating the Operations of the inner *j-loop in* SOS Method

As shown in Figure 2, this inner *j* loop has s^2 iterations and requires $7s^2$ total number of operations to be performed.

Multiplication involves two basic operations, the generation of partial products, and their accumulation. Consequently, there are two ways to speed up multiplication; reduce the number of partial products or accelerate their accumulation. In order to perform these operations, all the registers on both kinds of processors are used and also some extra memory spaces, for the temporary intermediate values, are allocated on Pentium platforms. Particularly in the first approach, since Pentium machines have a fewer number of general-purpose registers than Sparc machines, intermediate values like C (carry) and *m* values are hold in memory spaces to let more registers to be used in the accumulation. Therefore, in this first approach, general-purpose registers are used for indexing, loop establishment, and for arithmetic operations on Pentium platforms. In addition to allocating memory locations for these temporary variables, instruction selection for integer operations is used whenever it can be applied. Complex instructions like *'loop, addcc'* are avoided in the implementation and more efficient instructions that require fewer clock cycles to decode, are used. This type of implementation approach enabled the implementation simplicity as well as the rearrangement of the integer instructions of the programs by placing memory operations between them. However it is observed that implementing algorithms with this

approach did not result in the optimum implementations. In this first approach, algorithms show slow performance results and there are performance differences between them. Since the algorithms are not using the full advantages of the processors, particularly parallel integer pipelines [20], this is an expected result. Although the processor could execute two instructions simultaneously in this pipelining technology, the data dependencies between the consequent instructions prevented these instructions to be pipelined in the pipeline stages of the processors.

In second approach, only the necessary memory allocations are assigned and all the main operations are performed with registers by using them interchangeably. Performing memory operations with this approach certainly decreased the number of memory read and write operations that normally need more clock cycles then register operations. However more importantly, performing operations this way enabled us to increase the parallelism between the integer instructions. The software optimization techniques that are applied to Intel architecture processor became more effective in this second approach [21] [22] [23]. As a result we obtained improved the performances of the algorithms. As the algorithm's performances increase, the performance differences between the algorithms decreased. Figure 3 shows a simple example of this optimization. The code that is written for first approach, which requires 5 clock cycles to execute, because of the dependency between the instructions, are reordered in second approach that requires 3 clock cycles to execute.

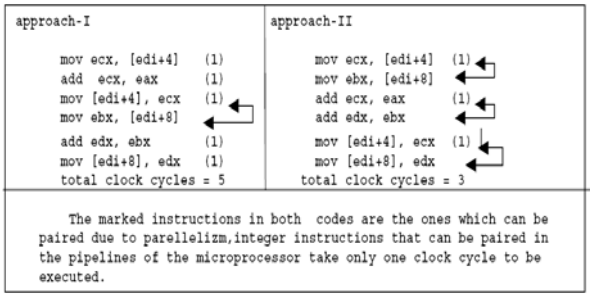


Figure 3: Example of Reordering Optimization Method.

Even if the clock differences between two approaches seem unimportant, it becomes considerably large in the loops where more integer instructions executed repeatedly with larger loop iterations. The parallel execution of the instructions in the pipeline stages of the processor reduced the necessary operations which affects the speed performance of the programs. This result can be seen in the Figure 4 where the source code of the first inner *j-loop* of the SOS algorithm is given with the computation of the effective instruction number. As shown in Figure 4, the numbers of instructions that needed to be executed in order to perform Montgomery multiplication is decreased. Most of the instructions in Figure 4 require one clock cycle to execute.

Statement	Multiplications	Adds	Reads	Writes
Mul dword ptr [ebx]	1	0	0	0
Mov ecx, dword ptr [edi]		0	1	0
Add esi, edx	1	0	1	0
Add ebx, 4		0	1	0
Add eac,ecx	1	0	1	0
Adc eax, esi	1	0	1	0
Mov dword ptr [edi],eax		0	0	1
Mov esi,0	1	0	0	1
Adc esi,0	1	0	1	0
Add edi, 4				
Total:	$6s^2$			

Figure 4: Calculating the Effective Number of Operations for the *inner j-loop* in SOS Method for Pentium Platforms.

The same calculations can be done for the Sparc platforms. On Sparc processor, since there are enough number of registers for all arithmetic, addressing and indexing operations, there were no memory allocations for the intermediate values, which may let more registers to be used for arithmetic operations. Particularly, for Sparc processors, the software optimization is focused on instructions alignment to increase parallel execution of instructions in the pipelines of the processor. UltraSparc-II processor uses a double-instruction-issue pipeline with 9 stages. UltraSparc-II is capable of sustaining the execution of up to four instructions per cycle, even in the presence of conditional branches and cache misses. In addition to these properties, UltraSparc-II provides 64-bit RISC architecture with on-chip caches and dynamic branch prediction capability [24]. Therefore during the implementation process on Sparc processors, instructions are carefully arranged not to have any data dependencies. Also in order to reduce the loop and branch mismatch delays during the execution process, labels and branches are carefully addressed and placed in the programs. Therefore after performing software optimization techniques for both Sparc and for Intel processors, and by enabling the parallel execution of the program's instructions resulted in performance increase.

3 Implementation Results

The Montgomery multiplication algorithms are implemented in the assembly languages of Intel Pentium, and Sun Sparc machines. The first Intel Pentium processor has the speed of 120 MHz and the Intel Pentium-II processor has the speed of 266 MHz. The Microsoft Visual C is used in the development of the programs and the Microsoft C complier is configured to obtain the speed-optimized code for the Intel Pentium platform where programs are written as in-lined assembly language. The GNU C compiler was

used for the UltraSparc-II 250 MHz platform. The programs are embedded into a RSA implementation as *MonPro( )* functions and performance results are taken by executing this RSA implementation over 1000 times for a message file with 3KB-file size. In these implementations, numbers whose array size s=64 words corresponding to 2048 bits are used. The algorithms are run on Windows 95, Windows 98, Windows NT 4.0, and Solaris 2.5.1 operating systems. The programs' performance results are summarized in Table 5, 6 and 7.

Table 5: Performance Results for Pentium-120 on Windows 95.

Processor	Operating System	
Pentuim-120	Windows 95	
	I-Approach	II-Approach
SOS	106ms	95ms
CIOS	98ms	97ms
FIOS	105ms	100ms
FIPS	101ms	97ms
CIHS	104ms	97ms

Table 6: Performance Results for Pentium-II 266

Processor	Operating System			
Pentium-II 266	Windows 98		Windows NT 4.0	
	I-Approach	II-Approach	I-Approach	II-Approach
SOS	46ms	45ms	46ms	44ms
CIOS	45ms	45ms	44ms	44ms
FIOS	48ms	46ms	47ms	45ms
FIPS	45ms	44ms	44ms	44ms
CIHS	47ms	45ms	46ms	44ms

Table 7: Performance Results for UltraSparc-II 250 on Solaris 2.5.1

Processor	Operating System
UltraSparc-II 250	Solaris 2.5.1
SOS	11.19ms
CIOS	11.36ms
FIOS	11.49ms
FIPS	11.58ms
CIHS	11.20ms

In Table 5 and 6, in I-Approach some memory allocations and therefore some memory operations are used to see their effects on the performance. On the other hand, in II-Approach, registers are used as much as possible, in all operations, to increase the parallelism between integer instructions by removing the data dependencies seen in I-Approach.

As seen from the results, in II-Approach, algorithms show better performance results. More importantly, the performance differences between each algorithm decreases. For the Ultra-Sparc-II, Table 7, algorithms show almost the same result.

The performance increase, in percentage, of each algorithm in II-Approach is shown in Table 8 below.

Table 8: Percentage Performance Improvements of Algorithms in Approach-II

Processor	Pentium-120		Pentium-II 266			
Operating System	Windows - 95		Windows98		Windows NT 4.0	
	Improvement	%	Improvement	%	Improvement	%
SOS	11ms	10	1ms	2,1	2ms	4,3
CIOS	1ms	1	-	-	-	-
FIOS	5ms	5	2ms	4,1	2ms	4,2
FIPS	4ms	3,9	1ms	2,2	-	-
CIHS	7ms	6,7	2ms	2,1	2ms	4,3

As seen in Table 8, each algorithm shows performance increase. This performance increase resulted from the applied optimization methods. These optimization methods are more effective on Pentium processors in II-Approach where the data dependencies between integer instructions are removed. The performance increase changes with each specific algorithm. For example SOS algorithm shows 10 % performance increase while CIOS algorithm shows 1% increase on the same platform for Pentium-120 after optimization process. Because of the code arrangements of the programs, this different performance increase occurs. In I-Approach for Pentuim-120, CIOS algorithm shows best performance. The CIOS and FIPS algorithms seem to be the most efficient algorithm on Pentium-II 266 platform, with each having 45 and 44 ms performance values on Windows 98, and Windows NT 4.0 platforms.

In II-Approach, the performance results of Montgomery multiplication algorithms show nearly the same performance results. With the applied optimization methods, program's performance results improved, and they reached an optimum value, which is nearly the same for the five algorithms. The reason for the programs to have same performance results on Pentium-II 266 processor is that the effective number of executed instructions decreased with parallel execution and therefore all the algorithms have nearly same number of executed instructions during the execution process. These results show that with an efficient, optimized implementation of the algorithms, the performance differences presented between programs could be up to 5 ms between SOS and FIOS in pentium-120 machines. This performance differences disappears on newer technology processors like Pentium-II 266, and it becomes only 1 ms with FIOS and with no differences with the others. This result can also be best seen from the performance results of programs on Sparc

processor where all algorithms show same performance results, only differing in micro seconds. As known, Sparc processors have deeply pipelined architecture and can execute up to 4 instructions per unit cycle. This reduces the effective number of operations. Therefore with the same reason as with Pentium machines, the effective number of executed instructions becomes almost equal for all algorithms and therefore they show the same performance results on Sparc platforms.

4 Conclusion

The Montgomery multiplication algorithms to perform fast large integer modular multiplication and exponentiation operations with stable timing characteristics are important algorithms for the overall performance results of many public-key cryptosystems. After performing the software implementations of five well-known algorithms of the Montgomery multiplication method in assembly languages of Intel Pentium family and Sun Sparc family microprocessors, it is found that the performance results of the five algorithms are almost the same. Study indicates that the algorithm's performance results improve on newer technology product processors and accordingly the performance differences between algorithms disappear. This is the result of the fact that since the main operations involved in the algorithms are integer arithmetic operations, the effective number of operations needed to perform these integer operations can be reduced by using the architectural benefits of processors such as parallel execution of integer instructions, deeper pipeline stages, additional functional instructions and bigger amount of on-cache availability. Therefore, by using the architectural benefits of these microprocessors combined with the usage of efficient and optimum software implementation methods, all the implemented Montgomery multiplication algorithms show almost the same performance results.

5 REFERENCES

1. A.Menezes, P.van Oorschot & S.Vanstone, Public-Key Cryptography,*Handbook of Applied Cryptography*, 1997, pp. 25- 33.
2. Bruce Schneider , Public-Key Algorithms, *Applied Cryptography*, 1996, pp. 461-482.
3. R.L. Rivest, A. Shamir & L.M. Adleman, " A method for obtaining Digital Signatures and Public-Key Cryptosystems ", *Communications of the ACM*, vol.21, 1978, pp.120-126.
4. W. Diffie and M.E. Hellman, " New Directions in Cryptography ", *IEEE Transactions on Information Theory*, IT-22, n.6, Nov 1976, pp. 644-654.
5. Paul C. Kocher, " Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems ", *Advances in Cryptology*, *Proc. Crypto96*. Lecture Notes in Computer Science, vol 1109, N. Koblizt editor, Spring-Verlag, 1996,pp. 104-113.
6. A.Bosselaers, R.Govaerts, J.Vandewalle, *Comparison of three modular reduction functions*, Crypto'93, pp.175-186.
7. Y.Yacobi, Exponentiating faster with addition chains, *Eurocrypt'90*, 1991, pp. 222-229.
8. A.Selby, C.Mitcheil, *Algorithms for software implementations of RSA*. IEE Proceedings(E), vol.136, NO.3, May, 1989, pp. 166-170.
9. P.D Barrett, " Implementing the Rivest Shamir and Adleman public key encryption algorithm on a standard signal processor," Advances in Cryptography, Proc. Crypto '86, LNCS 263, A.M. Odlyzko, Ed., Springer - Verlag, 1987, pp. 311-323.
10. P.L Montgomery, " Modular Multiplication without Trial Division", *Math. Computation*, vol. 44, 1985, pp. 519-521.
11. Thierry Moreau, *Software Acceleration for Public Key Cryptography* Connotech Experts-conceals Inc, May 1977.
12. A.Menezes, P.van Oorschot & S.Vanstone, Number Theory , *Handbook of Applied Cryptography*, 1997, pp. 242-248.
13. D.E. Knuth. The Art of Computer programming: Seminumerical Algorithms,volume 2. Reading, MA: Addison-Wesley, Second edition, 1981.
14. A.Menezes, P.van Oorschot and S.Vanstone, Multiple precision modular arithmetic, *Handbook of Applied Cryptography*, 1997, pp 599 – 606.
15. Bruce Schneider, Modular arithmetic, *Applied Cryptography*,1996, pp.242-245.
16. S.R Dusse and B.S. Kaliska, Jr. A cryptographic library for Motorola DSP56000. In I.B.Damgard, editor, Advances in Cryptology – EUROCYPT90, Lecture Notes in Computer Science, No. 473, New York, NY : Springer-Verlag, 1990, pages 230-244.
17. P.G. Comba. Exponentiation cryptosystems on the IBM PC. IBM SystemsJournal, 29(4) 1990, pages 526-538.
18. G. Hachez and J.-J. Quisquater, Montgomery exponentiation with no finalsubtraction: Improved results. *Workshop on Cryptographic Hardware and Embedded Systems,* CHES 2000.
19. Intel Architecture Optimization Manual.
20. Pentium Processor Family Developer's Manual, Volume 3: Architecture and Programming Manual, Part IV Optimization.
21. "Developers' Performance Reference" http://developer.intel.com/software/idap/asc/performance.htm
22. Intel Architecture Optimization Manual.
23. C. K. Koc, T. Acar, and B. S. Kaliski Jr., Analyzing and Comparing Montgomery Multiplication Algorithms *IEEE Micro,* June 1996, 16(3):26-33.
24. UltraSparc-II Processor Architecture Manual, White Papers, Sun Microsystems.