How to Maximize Software Performance of Symmetric Primitives on Pentium III and 4 Processors

Mitsuru Matsui and Sayaka Fukuda

Information Technology R&D Center
Mitsubishi Electric Corporation
5-1-1 Ofuna Kamakura Kanagawa, Japan
matsui@iss.isl.melco.co.jp, sayaka@iss.isl.melco.co.jp

Abstract. This paper discusses the state-of-the-art software optimization methodology for symmetric cryptographic primitives on Pentium III and 4 processors. We aim at maximizing speed by considering the internal pipeline architecture of these processors. This is the first paper studying an optimization of ciphers on Prescott, a new core of Pentium 4. Our AES program with 128-bit key achieves 251 cycles/block on Pentium 4, which is, to our best knowledge, the fastest implementation of AES on Pentium 4. We also optimize SNOW2.0 keystream generator. Our program of SNOW2.0 for Pentium III runs at the rate of 2.75 μops/cycle, which seems the most efficient code ever made for a real-world cipher primitive. For FOX128 block cipher, we propose a technique for speeding-up by interleaving two independent blocks using a register group separation. Finally we consider fast implementation of SHA512 and Whirlpool, two hash functions with a genuine 64-bit architecture. It will be shown that new SIMD instruction sets introduced in Pentium 4 excellently contribute to fast hashing of SHA512.

1 Introduction

Recent microprocessors, especially Intel processors, have long pipeline stages to raise clock frequency, which, on the other side, often leads to new performance penalty factors. Now it is not rare that a program runs slower on a newer processor with a higher clock frequency. It seems that the clock-raising of modern processors is approaching to its margin. That is, for maximizing performance of a software program on a processor, it is becoming increasingly important for programmers to understand its hardware architecture and programming techniques specific to the processor.

This paper deals with Intel Pentium III and 4 processors, which are most widely used in modern PCs, and studies methodology for optimizing speed of recently proposed symmetric ciphers and hash functions on these processors. Intel recently shipped a new Pentium 4 (Prescott) with an architecture different from the previous Pentium 4 (Willamette, Northwood) under the same name.

Since we often have to discuss these different cores separately, we call the old cores Pentium 4-N, and the new core Pentium 4-P to distinguish them.

First, in section 2, we refer to Gladman's code of Serpent block cipher [9] to see to what extent an architecture of microprocessors affects performance of the same code. It will be seen that even if everything is on the first level cache, the number of execution cycles of a given code significantly varies on a processor with a different version. Then we briefly summarize structural characteristics of Pentium III and 4. It should be noted that Pentium 4 has successfully raised its clock frequency by increasing the number of pipeline stages, but in return, SIMD instructions work only in a longer latency on this processor.

In section 3, we show how to measure a speed of a target assembly code on these processors. We have adopted a common method for the measurement; i.e. we count the number of clock cycles of a target subroutine using an internal timer of the processor. However in repeating our measurement experiments, we have found that an accurate measurement of execution cycles is not a simple issue as it looks on Pentium 4 particularly with Hyperthread Technology. Since this is a separate matter of interest but a less cryptographic topic, we will give a further observation in an appendix.

In subsequent sections, we specifically discuss software optimization techniques for symmetric cryptographic primitives. Our first target algorithm is AES [6]. The structure of AES is very suitable for 32-bit processors, but if we aim at ultimate performance, a dependency chain and a register starvation are likely a bottleneck of the speed. We carefully selected and arranged registers and instructions, and as a result, our optimized code with 128-bit key runs in 251 cycles/block on Pentium 4, which is, to our best knowledge, the fastest implementation of AES on Pentium 4.

The next algorithm is stream cipher SNOW2.0 [4]. This algorithm has two highly independent functions inside, and hence is suitable for superscalar processors. We derive a possible minimum number of μops on Pentium III and 4, and show that this number can be achieved in practice. Our program generates a key stream very efficiently at the rate of 2.75 μops/cycle on Pentium III, which is very close to the structural limit of Pentium III and 4 (3 μops/cycle), and is, as far as we know, the most efficient code designed for a read-world cipher primitive.

We also give the first performance analysis of FOX128 block cipher [11]. Since this cipher has an 8-byte$\times$8-byte matrix inside, we should use the 64-bit MMX registers, but due to a long dependency chain, a straightforward program runs inefficiently. However, fortunately this algorithm does not require many registers, and we can improve performance by assigning two independent register sets to two independent blocks respectively and interleaving the two codes in an internal block loop. It will be seen that this technique excellently improves the speed of FOX128.

Finally we deal with hash functions SHA512 [7] and Whirlpool [3]. These hash functions have a genuine 64-bit structure, and use of 64-bit MMX instructions is essential. Nakajima et al. [14] studied performance analysis of these hash functions on Pentium III, but due to missing "64-bit add" instructions, SHA512 had a heavy performance penalty on Pentium III. This paper first gives detailed performance analysis of SHA512 and Whirlpool on Pentium 4. We show that a two-block parallel implementation (in the sense of [14]) using the 128-bit XMM registers significantly boosts its hashing speed.

All the results shown in this paper were obtained using the following PCs.

Processor	Pentium III	Pentium 4	Pentium 4
Core	Coppermine	Northwood	Prescott
Clock	800MHz	2.0GHz	2.8GHz
Hyperthread	no	no	yes
Memory	256MB	1GB	512MB
OS	Windows 2000	Windows XP Professional	Windows XP Professional
Compiler	Microsoft Visual Studio .NET 2003/Macro Assembler Version 7		

Table 1. Our reference machines and environments.

2 Pentium III and 4 Processors

2.1 Pentium III and 4 at a glance

Table 2 shows our performance measurement results of Gladman's implementation [9] of Serpent block cipher [1] written in an assembly language. In [9] we can find two assembly language source codes: one is coded using 32-bit x86 instructions only (Program 1) and the other encrypts two blocks in parallel using 64-bit MMX SIMD instructions, where the first block is put on the upper 32-bit half of the MMX registers and the second block on the lower half (Program 2). This parallel encryption technique works well because Serpent was designed so that the entire algorithm could be efficiently implemented using 32-bit logical and shift operations only. This implementation technique can be used for encrypting not only two independent message streams but also one single stream with a non-feedback mode of operation such as a counter mode. In addition, we modified Program 2 to enable us to encrypt four blocks in parallel using 128-bit XMM SIMD instructions (Program 3); this translation is very straightforward.

	Pentium III	Pentium 4-N	Pentium 4-P
Program 1 (32-bit code)	773	1267	689
Program 2 (64-bit code)	570	1052	1119
Program 3 (128-bit code)	-	656	681

Table 2. Encryption speed of Gladman's Serpent codes (cycles/block).

Program 1 is very slow on Pentium 4-N; this is probably because 32-bit shift instructions have long latencies (4 or 5 cycles) on this processor, which was later improved on the new Prescott core (more than one shift in one cycle). Program 2 runs faster on Pentium III but not twice as fast, because we need four instructions to do a rotate shift on the MMX registers. The reason why Program 2 is again so slow on Pentium 4 is totally different; the MMX units of Pentium 4 work only in half speed. On the other side, Program 3 is fast as expected due to the SIMD computation, as compared with Program 2. (Program 3 does not work on Pentium III because Pentium III does not have 128-bit XMM shift instructions).

Note that these programs are not optimized for Pentium 4, and hence this table should not be seen as a maximal performance figure of Serpent. It was intended to show a typical example where the same code runs in a totally different efficiency on a processor with a different version. Table 2 clearly shows that a selection of a processor and a careful optimization on the processor are critically important for maximizing performance.

2.2 Pentium III and 4 a bit more

We here sketch structural characteristics of Pentium III and 4 for later sections. For more details about the internal architecture and optimization tips of Pentium III and 4-N, see an excellent article written by Agner Fog [8], which tells us much more than any published documents about these processors.

[**Pentium III**] One of the biggest stall factors of Pentium III comes from the decoding stage, where a sequence of x86 instructions is broken down into RISC-style micro operations (μops). This break-down rule is quite complex, and a programmer must carefully arrange the order of instructions in order not to suffer a stall in this stage.

The executing stage has five independent pipes p0–p4, where p0 and p1 handle arithmetic and logical μops, p2 is used for reading from memory, and p3/p4 are used for writing to memory. This means that to aim at 3 μops/cycle, which is the maximal execution rate of Pentium III, we have to assign at least one μop out of three μops to memory read/write.

[**Pentium 4 Northwood**] In Northwood, instructions are cached after decoding. This means that the decoding stage is no longer a bottleneck of speed, assuming that the size of a critical loop is sufficiently small. An important feature of Northwood is that execution units for simple 32-bit μops run in double speed, but those for 64-bit/128-bit SIMD μops work only in half speed.

A special penalty of Northwood comes from 32-bit shift instructions, which have long latencies, typically 4 or 5 cycles. Also reading from memory to the MMX/XMM registers is very slow, taking approximately 8 cycles, according to [8]. The maximal execution rate of this processor remains 3 μops/cycle.

[**Pentium 4 Prescott**] This new core of Pentium 4 has not been well documented. The speed of a 32-bit shift instruction is greatly improved; more than one shift can be issued in a single cycle (but not exactly two shifts in our experiments, unlike what Intel's manual says). This is a good news.

However, many μops of Prescott have longer latencies than those of Northwood due to a deeper pipeline of this processor. The latency of a 32-bit load and an xor, for instance, is 4 and 1, respectively (2 and 0.5 for Northwood), which can be a new performance constraint.

Table 3 summarizes major differences of Pentium III and 4. The sixth and seventh rows show a latency of a sequence of two instructions, whose results were obtained by our own experiments. This type of sequence often appears on a dependency chain of block cipher codes.

	Pentium III	Pentium 4-N	Pentium 4-P
Pipeline Stages	10	20	32
L1 data cache	16KB	8KB	16KB
32-bit load latency/throughput	3/1	2/1	4/1
32-bit xor latency/throughput	1/0.5	0.5/0.5	1/0.5
32-bit shift latency/throughput	1/1	4/1	>0.5/1
`mov ebx,TABLE[eax] / mov eax,ebx`	4 cycles	3 cycles	5 cycles
`movq mm0,TABLE[eax] / movd eax,mm0`	4 cycles	13 cycles	18 cycles

Table 3. Pentium III vs. Pentium 4.

3 How to Measure Execution Cycles

A common method for measuring a speed of a piece of code is to insert the code to be measured between two `CPUID-RDTSC` sequences, where `CPUID` flushes the pipeline and `RDTSC` reads processor's internal clock value as follows:

```
xor    eax,eax
cpuid
rdtsc
mov    CLK1,eax
xor    eax,eax
cpuid

FUNCTION(..., int block)

xor    eax,eax
cpuid
rdtsc
mov    CLK2,eax
xor    eax,eax
cpuid
```

Code 1. Measurement of `FUNCTION`.

```
xor    eax,eax
cpuid
rdtsc
mov    CLK3,eax
xor    eax,eax
cpuid

; nothing here

xor    eax,eax
cpuid
rdtsc
mov    CLK4,eax
xor    eax,eax
cpuid
```

Code 2. Measurement of overhead.

Clearly `CLK2-CLK1` shows clock cycles from line 4 to line 11, but this value contains an overhead of measurement itself, which corresponds to `CLK4-CLK3`. Hence we define the speed of `FUNCTION` as `((CLK2-CLK1)-(CLK4-CLK3))/block` (cycles/block). In our reference PCs, the overhead `CLK4-CLK3` is 214, 632 and 847 cycles for Pentium III, 4-N and 4-P, respectively.

In practice, the measured number of cycles varies due to various reasons. We hence ran the code above many times and adopted an average value, not a minimal value, as the speed of `FUNCTION`. For more details about the measurement issue on Pentium 4, see appendix.

4 AES

The first example of our implementation is AES. For the description and notations of the AES algorithm, refer to [6]. The fastest AES codes currently known on Pentium III and Pentium 4-N were designed by Lipmaa [12][13]. However no information about his implementation details has been published. Our implementation below is hence independent of his works.

AES is a typical cipher from an implementation viewpoint in the sense that we have to make use of data registers also as address registers alternatively on its critical path, which means that a dependency chain is likely a performance bottleneck. A common x86 code of one round of AES consists of (1) a four-time repetition of the following sequence (with different input registers), which corresponds to `Subbytes+Shiftrows+Mixcolumns`, and (2) four xors, which correspond to `AddRoundKey`. Note that, while the final round of AES is different from other rounds, it can be implemented using the same sequence below with another tables, which will be referred as `table5` to `table8`. We hence need a total of 8KB memory for the lookup tables.

```
movzx    esi,al                      ; lowest byte of input eax
mov/xor  register1,table1[esi*4]     ; first table lookup (4 byte)
movzx    esi,ah                      ; second byte of input eax
mov/xor  register2,table2[esi*4]     ; second table lookup (4 byte)
shr      eax,16                      ; move higher 16 bits to lower side
movzx    esi,al                      ; third byte of input eax
mov/xor  register3,table3[esi*4]     ; third table lookup (4 byte)
movzx    esi,ah                      ; highest byte of input eax
mov/xor  register4,table4[esi*4]     ; fourth table lookup (4 byte)
```

Code 3. An example of 1/4 component of `Subbytes+Shiftrows+Mixcolumns` (`mov` for the first time and `xor` for the second to the forth times).

In an actual assembly program, how to minimize the latency of one round sequence is not trivial due to a "register starvation". Since we need four one-byte components of `registers1` to `register4` in the next round, these four registers should be `eax`, `ebx`, `ecx` and `edx`, but this is impossible without saving/restoring at least one input register in each round, which requires additional instructions. Assigning 64-bit MMX registers to registers1 to 4 also requires additional instructions for copying them to `eax`, `ebx`, `ecx` and `edx` for the next round, since we can not direct extract a byte of an MMX register to an x86 register.

[**Pentium III**] Our implementation on Pentium III uses four specially arranged lookup tables. These tables have an 8-bit input and a 64-bit output, where `table1` to `table4` (in **Code 3**) are put on the lower 32-bit half of each entry of our new tables, and `table5` to `table8` for the final round are put on the higher 32-bit half of the entry. We also assign two x86 registers and two MMX registers to `register1` to `register4` for all rounds except the final round, and assign four MMX registers to all of `register1` to `register4` in the final round. `movq/pxor` instructions are used to access MMX registers.

This lookup table structure contributes to reducing code size and hence increasing decoding efficiency. This structure also works very well in the final round, because the output of the final round no longer has to be copied to x86 registers and can be treated as full 64-bit data, as shown below. Our implementation of AES with 128-bit key on this strategy runs at the speed of 232 cycles/block in our measurement policy. When the block loop overhead (see appendix) is taken into consideration, this is almost the same performance as Lipmaa's best known result.

```
punpckhdq  mm0,mm1                      ; two upper 32-bit -> 64-bit
punpckhdq  mm2,mm3                      ; two upper 32-bit -> 64-bit
pxor       mm0, Final_Subkey1           ; AddRoundKey (8 bytes)
pxor       mm2, Final_Subkey2           ; AddRoundKey (8 bytes)
movq       [memory+0], mm0              ; store ciphertext (8 bytes)
movq       [memory+8], mm2              ; store ciphertext (8 bytes)
```

Code 4. Our AES code after the final round.

[**Pentium 4**] Since MMX memory instructions have a very long latency on Pentium 4 (for both Northwood and Prescott), we have to write a code using x86 registers and instructions only. In addition, using a high 8-bit partial register, such as `ah`, leads to a special penalty on Pentium 4, while no penalty takes place in using a low 8-bit partial register. Specifically, `movzx esi,ah` is decomposed into two μops on Pentium 4 unlike Pentium III. **Code 1** uses this type of instruction twice, but one of them can be avoided by changing the last two lines as follows:

```
shr       eax,8                       ; only one uop (upper 24 bits = 0)
mov/xor   register4, table4[eax*4]    ; fourth table lookup
```

Code 5. Modification of Code 1 for Pentium 4.

For Northwood, which has only 8KB L1 data cache, we reduced the size of our lookup tables to 6KB by removing `table5` and `table6` of the final round without increasing the number of instructions. This is possible by using a `movzx` instruction as shown in Table 4 (note that Pentium is a little-endian processor). As a result, our code runs at the speed of 251 cycles/block on Pentium 4 Northwood. This is, as far as we know, the fastest implementation of AES on Pentium 4. On the other hand, our implementation on Prescott is unfortunately slower than that on Northwood. We feel that this is due to a high latency of load instructions (4 cycles for Prescott and 2 cycles for Northwood).

Table 5 summarizes our performance results. Our codes have the following interface and we assume that the subkey has been given in the third argument. We set `block` to 128 or 256, which was fastest in our environments. We did not use any static memory except read-only lookup tables.

```
FUNCTION( uchar *plaintext, uchar *ciphertext, uint *subkey, int block )
```

	Operation	Instruction	Table Size
table5	x → (0‖0‖0‖S[x])	movzx Register,BYTE PTR table8+3[esi*4]	0
table6	x → (0‖0‖S[x]‖0)	movzx Register,WORD PTR Table8+2[esi*4]	0
table7	x → (0‖S[x]‖0‖0)	mov Register,table7[esi*4]	1KB
table8	x → (S[x]‖0‖0‖0)	mov Register,table8[esi*4]	1KB

Table 4. Reduction of lookup tables of the final round.

	Pentium III	Pentium 4-N	Pentium 4-P
μops/block	596	654	654
cycles/block	232	251	284
cycles/byte	14.5	15.7	17.8
μops/cycles	2.57	2.61	2.30

Table 5. Our implementation results of AES.

5 SNOW2.0

Our next example of implementation is stream cipher SNOW2.0, which was designed by Ekdahl and Johansson and presented at SAC2002 [4]. SNOW2.0 was intended to overcome a slight weakness of its earlier version, which was initially submitted to the NESSIE project [15]. SNOW2.0 is based on a firm theoretical background and is very fast. It is now under discussion for an inclusion in the next version of the ISO/IEC 18033 standard. Our paper gives the first detailed performance analysis of SNOW2.0 in an assembly language.

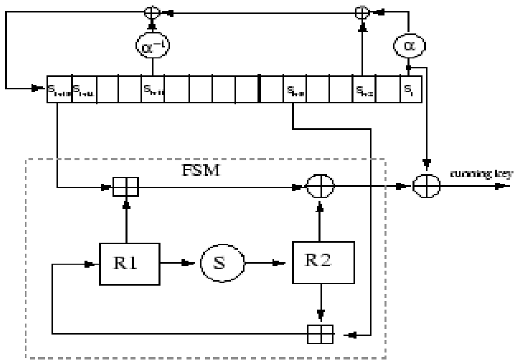

Fig. 1. SNOW 2.0.

Figure 1 illustrates the keystream generation algorithm of SNOW2.0, which consists of sixteen 32-bit registers s_i with feedback mechanism with two 32-bit memories $R1$ and $R2$. α and α^{-1} are multiplicative constants over $GF(2^{32})$, and S is an AES-like 4-byte×4-byte matrix multiplication. Clearly SNOW2.0 strongly targets at 32-bit processors from the implementation point of view.

According to the authors' document, α and α^{-1} were chosen so that the multiplications with these values over $GF(2^{32})$ could work efficiently using two pre-calculated tables `MUL_a` and `MUL_ainv` as follows:

```
s * α = (s << 8) xor MUL_a[s >> 24]
s * α^-1 = (s >> 8) xor MUL_ainv[s & 0xff]
```

The straightforward implementation of the first operation (multiplication with α) requires four instructions (five μops), but we have found that it can be done with fewer instructions as shown below by preparing a new table `MUL_a2` such that `MUL_a2[x] = MUL_a[x] ^ (a & 0xff)`. Since we need 2KB for `MUL_a2` and `MUL_ainv` and 4KB for S, the total size of the lookup tables is 6KB, which fits in L1 data cache of both Pentium III and Pentium 4.

```
rol    eax,8                          movzx  esi,al
movzx  esi,al                         shr    eax,8
xor    eax,MUL_a2[esi*4]              xor    eax,MUL_ainv[esi*4]
```

Code 6. Multiplication with α (left) and α^{-1} (right).

The structure of SNOW2.0 is very suitable for superscalar processors, since the LFSR part (the upper half of Figure 1) and the FSM part (the lower half) can be carried out mostly independently. For fast implementation of SNOW2.0, we should treat sixteen consecutive LFSR clocks as "one round" as suggested by the designers, which enables us to skip copy operations on the sixteen 32-bit registers. We implemented the keystream generation algorithm SNOW2.0 in an assembly language for Pentium III and Pentium 4, respecting the subroutine interface given by designers' C codes at [5]. We simply added an additional variable "block" in the second argument, so that the routine can generate block*64 keystream bytes at one subroutine call as follows (the state information on s_i, $R1$ and $R2$ are passed as static variables):

```
FUNCTION( uint *keystream_block, int block )
```

Our code requires 34 and 33 μops in one LFSR clock, i.e. in every four-byte keystream generation, on Pentium III and Pentium 4, respectively. We think that this already reaches the theoretical minimum number of μops. Table 6 shows a detailed breakdown of the μops of our code. We read s_t twice for reducing the latency of the LFSR part, which is the reason why the number of μops of "read from LFSR" is 5 (not 4 as naturally expected from Figure 6), and the number of μops of "s * α" is 3 (not 4).

Table 7 gives performance of our assembly codes. Our code runs at the speed of 203 cycles/block on Pentium 4 Northwood, which is 30% faster than designers' optimized C code. The remarkable aspect of our code is its high parallelism. For example, our program on Pentium III works at the rate of 2.75 μops/cycle, which is, as far as we know, the most efficient code that was achieved in a real cryptographic primitive. This result also shows that SNOW2.0 is essentially faster than RC4. If we take a close look at the structure of RC4, we will see that

RC4 requires at least 10 μops/byte including three reads and three writes. Hence even if we assume that an RC4 code works in 2.80 μops/cycle it takes at least 3.6 cycles/byte (much more in practice), while our SNOW2.0 code is running in 3.1–3.4 cycles/byte.

	S	s * α	s * α^{-1}	read from LFSR	write to LFSR/keystream	xor/add	Total μops
Pentium III	12	3	4	5	4	6	34
Pentium 4	13	3	4	5	2	6	33

Table 6. μops breakdown in one LFSR clock.

	Pentium III	Pentium 4-N	Pentium 4-P
μops/block	550	534	534
cycles/block	200	203	215
cycles/byte	3.13	3.17	3.36
μops/cycles	2.75	2.63	2.48

Table 7. SNOW2.0 key generation speed.

6 FOX128

FOX is a family of block ciphers, which was recently proposed by Junod and Vaudenay [11]. Here we treat a “generic version” of 128-bit block cipher FOX128 with 16 rounds. The left part of Figure 2 illustrates the round function of FOX128, and the right part gives the details of the $f64$ function in the round function. The $f64$ function consists of a sequence of (1) key xor, (2) eight parallel *sbox* lookups, (3) 8-byte $\times$ 8-byte matrix $mu8$, (4) key xor again, and (5) eight parallel *sbox* lookups again. This is essentially a 64-bit structure, suitable for use of the MMX instructions on Pentium III and 4.

The straightforward implementation of the $f64$ function requires eight 2KB tables for the first sbox layer and $mu8$, and additional one to four 1KB tables for the second sbox layer. If we take a close look at the $mu8$ matrix, it is easily seen that we can reduce one 2KB table in the first layer at the cost of three or four additional MMX μops. Our first implementation fully uses MMX registers and MMX instructions in the main stream and in the $f64$ function, reducing the table size to a total of 15KB (14KB/1KB for the first/second layer) so that the entire tables are covered within 16KB. This runs at the speed of 692 cycles/block on Pentium III, which is approximately 20% faster than designers’ optimized C implementation. However, this program becomes very slow in Pentium 4 because the $f64$ function has a long dependency chain and moreover Northwood suffers a lot of cache miss penalties.

On the other hand, our implementation is free from a “register starvation”; that is, four out of the eight MMX registers are enough to implement the entire cipher, which means that half of the MMX registers can remain free. This leads

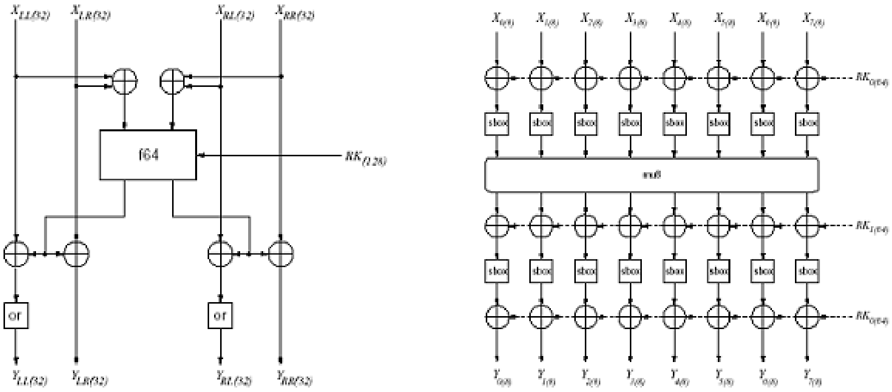

Fig. 2. FOX128.

us to a possibility of another parallel implementation technique “register group separation”. Specifically, we assign four MMX registers to one message stream and the remaining four to another message stream, and interleave the two independent codes inside a block loop. This technique is expected to contribute to an efficient use of superscalar pipelines and improve an overall performance accordingly. Our code remarkably reduces the number of execution cycles on Pentium 4. In particular, the improvement on Northwood is prominent; although the Northwood core is still paying the penalty of cache misses, its performance is now very close to that on Prescott.

Table 8 summarizes performance measurement results of our FOX128 codes, where (I) and (II) show the straightforward method and the two-block parallel method, respectively. We adopted the same subroutine interface and coding policy as that of the AES block cipher. In general, it is difficult to apply the register group separation technique to codes using x86 registers only, but new registers such as MMX and XMM have opened up a new possibility of this parallel computation technique.

	Pentium III		Pentium 4-N		Pentium 4-P	
	(I)	(II)	(I)	(II)	(I)	(II)
μops/block	1269	1388	1395	1505	1395	1505
cycles/block	692	622	1986	1187	1481	981
cycles/byte	43.3	38.9	124.1	74.2	92.6	61.3
μops/cycle	1.83	2.23	0.70	1.27	0.94	1.53

Table 8. Our implementation results of FOX128.

7 SHA512 vs. Whirlpool

We here discuss two genuine 64-bit hash functions SHA512 [7] and Whirlpool [3], both of which are now under consideration for an inclusion in the next version of the ISO/IEC 18033 standard. These algorithms well suit for 64-bit processors and it is expected that the 64-bit MMX instructions can be efficiently used for gaining performance. Nakajima et al. [14] discussed speed of these hash functions on Pentium III, and reported that Whirlpool is slightly faster than SHA512.

SHA512 suffered heavy penalty cycles on Pentium III because Pentium III does not have an instruction for 64-bit addition, which is an essential operation of this algorithm. Pentium 4 solves this problem, but a high latency of MMX memory instructions can be a possible penalty factor. On the other side, we can hash two independent messages in parallel using 128-bit XMM instructions, which is expected to boost the hashing speed.

The structure of Whirlpool is similar to AES. It uses an 8-byte × 8-byte matrix (a 4-byte × 4-byte matrix for AES), and hence a straightforward implementation requires eight 2KB tables. Our coding method is basically the same as [14]; that is, we have only four tables and generate other data when necessary using the `pshufw` (word shuffling) instruction.

Table 9 and Table 10 show our performance figures of SHA512 and Whirlpool, respectively. We also made our own programs for Pentium III, where Whirlpool runs faster than [14]. This is due to a better instruction scheduling. "single" and "double" denote straightforward single message hashing using 64-bit MMX instructions and double message hashing using 128-bit XMM instructions, respectively. In the single message hashing, Whirlpool is still faster than SHA512 on Pentium 4 Prescott, (Northwood is slow simply due to cache miss penalties), but the effect of double message hashing is evident; SHA512 then becomes more than 30% faster than Whirlpool.

1block = 128bytes	Pentium III	Pentium 4-N		Pentium 4-P	
	single	single	double	single	double
μops/block	13924	8710	4363	8710	4363
cycles/block	5148	4666	2826	5294	3111
cycles/byte	40.2	36.5	22.1	41.4	24.3
μops/cycles	2.70	1.87	1.54	1.65	1.40

Table 9. Our implementation results of SHA512.

1block = 64bytes	Pentium III	Pentium 4-N	Pentium 4-P
μops/block	5206	5526	5526
cycles/block	2061	3024	2319
cycles/byte	32.2	47.3	36.2
μops/cycles	2.53	1.83	2.38

Table 10. Our implementation results of Whirlpool.

8 Concluding Remarks

This paper discussed various implementation trade-offs of cryptographic primitives on Pentium III and 4 processors, introducing parallel encryption techniques. The clock-raising of modern processors is approaching to its margin and it seems that a next generation of processors goes toward independent multiple cores, rather than a deeper pipeline and a higher superscalability. Hence we believe that parallel encryption/hashing techniques will be increasingly important in a very near future.

References

[1] R. Anderson, E. Biham, L. Knudsen: “Serpent: A proposal for the Advanced Encryption Standard”, http://www.ftp.cl.cam.ac.uk/ftp/users/rja14/serpent.pdf

[2] K. Aoki, H. Lipmaa: “Fast Implementations of AES Candidates”, Proceedings of The Third AES Candidate Conference, 2000. Available at http://www.tcs.hut.fi/~helger/papers/al00/fastaes.pdf

[3] P. Barreto: “The Whirlpool Hash Function”, http://planeta.terra.com.br/informatica/paulobarreto/WhirlpoolPage.html

[4] P. Ekdahl, T. Johansson: “A new version of the stream cipher SNOW”, Proceedings of 9th Annual Workshop on Selected Areas in Cryptography SAC2002, Lecture Notes in Computer Science, Vol.2595, pp 47-61, Springer-Verlag, 2002.

[5] P. Ekdahl: SNOW Homepage, http://www.it.lth.se/cryptology/snow/

[6] Federal Information Processing Standards Publication 197, “Advanced Encryption Standard (AES)”, NIST, 2001.

[7] Federal Information Processing Standards Publication 180-2, “Secure Hash Standard”, NIST, 2002.

[8] A. Fog: “How To Optimize for Pentium Family Processors”, Available at http://www.agner.org/assem/

[9] B. Gladman: “Serpent Performance”, Available at http://fp.gladman.plus.com/cryptography_technology/serpent/

[10] IA-32 Intel Architecture Optimization Reference Manual, Order Number 248966-011, http://developer.intel.ru/download/design/Pentium4/manuals/24896611.pdf

[11] P. Junod, S. Vaudenay: “FOX : a new Family of Block Ciphers”, Preproceedings of 11th Annual Workshop on Selected Areas in Cryptography SAC2004, pp131-146, 2004.

[12] H. Lipmaa: “Fast Software Implementations of SC2000”, Proceedings of Information Security Conference ISC2002, Lecture Notes in Computer Science, Vol.2433, pp.63-74, Springer-Verlag, 2002.

[13] H. Lipmaa: “AES / Rijndael: speed”, http://www.tcs.hut.fi/~helger/aes/rijndael.html

[14] J. Nakajima, M. Matsui: “Performance Analysis and Parallel Implementation of Dedicated Hash Functions on Pentium III”, IEICE Trans. Fundamentals, Vol.E86-A, No.1, pp.54-63, 2003.

[15] New European Schemes for Signatures, Integrity, and Encryption (NESSIE), https://www.cosic.esat.kuleuven.ac.be/nessie/

Appendix: On Measuring Execution Cycles

Our measurement method shown in section 3 agrees with [14], but not with [2]. Aoki et al. [2] regarded execution time for decrementing the block counter and branching conditionally inside `FUNCTION` also as an overhead. Specifically, they subtracted the number of execution cycles of the following "Null function" from that of `FUNCTION`, and defined its result as performance of the target primitive.

```
    /* push all used registers */
    cmp   dword ptr [block], 0
    jz    L1
    align 16
L0:
    dec   dword ptr [block]
    jnz   L0
L1:
    /* pop these registers once more */
```

We do not adopt this method because our definition is more practical and visible for users (application programmers) and moreover it is difficult to measure the overhead of the loop processing accurately, due to the nature of superscalar and out-of-order architecture of the processors. It should be noted that in Pentium 4 micro-operations in a small loop are likely rearranged on the trace cache so that the number of branches can be reduced [8].

Also, it is common to count execution cycles many times and regard the minimum value as a "real" cycle count in practice. This is based on the assumption that an interruption by an operation system always increases execution cycles. But this does not always hold for Pentium 4 with Hyperthread Technology (HT), which enables a single processor to run two multi-threaded codes simultaneously. Our experiments show that **Code 2**, for example, runs in 632 cycles on Northwood without HT (or with disabled HT) in almost all cases, and takes more cycles in some rare cases; however on Northwood with HT, **Code 2** runs in 636 cycles in almost all cases and takes more or **less** cycles in some rare cases. We saw it run even in 600 cycles!

Another implicit assumption is that we should always obtain a constant cycle count if no interruption takes place during the measurement. To make sure this, we again measured the speed of **Code 2** under DOS with disabling interruptions for more than 20 processors with different stepping/revision numbers. As a result, we found that only one type of processor (Prescott Stepping 3 Revision 0) did not run in a constant time. We do not know reason of this instability.

This suggests that if we measure a target code many and many times, we might finally obtain an exceptionally fast result, but clearly this does not make sense in practice. We hence decided to take the most frequent value or an average value in measuring a speed of a code. Our experiments show that the number of cycles obtained by Aoki et al's method [2] is smaller than ours by typically 6, 14 and 7 cycles/block for Pentium III, 4-N and 4-P, respectively. Hence to compare our results with Aoki's or Lipmaa's, simply subtract these numbers from ours.