# Performance Analysis and Parallel Implementation of Dedicated Hash Functions

Junko Nakajima, Mitsuru Matsui

Mitsubishi Electric Corporation
5-1-1 Ofuna, Kamakura, Kanagawa 247-8501, Japan
{june15,matsui}@iss.isl.melco.co.jp

**Abstract.** This paper shows an extensive software performance analysis of dedicated hash functions, particularly concentrating on Pentium III, which is a current dominant processor. The targeted hash functions are MD5, RIPEMD-128 -160, SHA-1 -256 -512 and Whirlpool, which fully cover currently used and future promising hashing algorithms. We try to optimize hashing speed not only by carefully arranging pipeline scheduling but also by processing two or even three message blocks in parallel using MMX registers for 32-bit oriented hash functions. Moreover we thoroughly utilize 64-bit MMX instructions for maximizing performance of 64-bit oriented hash functions, SHA-512 and Whirlpool. To our best knowledge, this paper gives the first detailed measured performance analysis of SHA-256, SHA-512 and Whirlpool.

**Keywords.** dedicated hash functions, parallel implementations, Pentium III

## 1 Introduction

A one-way and collision resistant hash function is one of the most important cryptographic primitives that require utmost speed particularly in software due to its heavy use for creating a digital finger printing of a long message. Historically the first constructions of hash functions were based on strong block ciphers and many efforts have since been done for their design and proof of security. However since this design approach does not necessarily result in fast hash functions in practice and often their hashing speed is much slower than underlying block ciphers, many "dedicated hash functions" suitable for software implementation on modern processors have been proposed and are now widely used in real world applications.

Therefore, performance analysis of hash functions in real environments is recognized as an important research topic and many studies have been done on this topic. Among them, a paper presented at CRYPTO'96 by Bosselaers et al. [BGV96] showed an excellent fast implementation and performance evaluation of dedicated hash functions (of that time) on the Pentium processor, which was a dominant processor at the time of the publication. They also gave a thorough critical path analysis of the MD-family, particularly concentrated on SHA-1 in the paper presented at Eurocrypt'97 [BGV97].

Recently NIST published three new dedicated hash functions with a larger hash size; SHA-256, SHA-384 and SHA-512 [FIP01], of which SHA-386 is essentially a truncation of the hashed value of SHA-512. These new hash functions have much more complex structure than SHA-1. Also in the European NESSIE project, a new 512-bit hash function Whirlpool was proposed [BR00]. The structure of Whirlpool is very similar to Rijndael, where the block size of its underlying block cipher is 512. All these new hash algorithms are under discussion for an inclusion in the next version of the ISO/IEC 10118 standard.

The purpose of this paper is to include these new generations as well as currently used dedicated hash functions and give an extensive performance analysis with actual implementations and measured cycle counts in a real processor platform. We particularly concentrate on the Pentium III processor, but most of our programs also run on Pentium II and Celeron in the same efficiency since these processors largely share their internal architecture. Note that Pentium 4 has a new architecture with richer SIMD instructions (but some instructions take a larger number of cycles now), which we do not deal with in this paper.

For the 32-bit oriented hash functions (MD5, RIPEMD-128, RIPEMD-160, SHA-1 and SHA-256), we perform not only a straightforward coding using 32-bit x86 registers, but also give an implementation method that enables fast parallel hashing of two or even three independent message blocks in parallel using 64-bit MMX registers (and x86 registers simultaneously) on Pentium III. Specifically, the two-block parallel method assigns the two blocks to upper and lower 32-bit halves of the MMX registers, and the three-block parallel method moreover assigns the third block to the x86 registers. For the 64-bit oriented hash functions (SHA-512 and Whirlpool), we fully utilize the MMX registers and instructions to extract maximal hashing performance. For another example of an optimization of a cryptographic algorithm using the MMX technology, see [Lip98].

The internal architecture of Pentium II/III is totally different from that of Pentium. Coding on Pentium was a programmers' paradise; estimating a cycle count of a given piece of code is not very difficult and, consequently, serious efforts to optimize a program by carefully arranging instructions were always rewarded. Unfortunately this is not the case for Pentium II/III. Intel documented the hardware architecture of Pentium II/III well [Int01][Int02][Int03], but still it is no longer possible to correctly predict how many cycles a given code takes without an actual measurement. This is partly due to an out-of-order execution nature of the processor, and also probably due to undocumented and unknown pipeline stall factors that only the hardware designers of Pentium II/III know.

In our experiences, even a well tuned-up code on Pentium II/III usually runs 10%-15% slower than what we expect. Filling this gap is a programmers' nightmare; it is a groping task with endless trial and error and very often such efforts are not rewarded at all. So in our implementations of hash functions, we first tried to write a code so that its data dependency chain could be as short as possible, and then re-scheduled the code to remove possible pipeline stall factors until the measured performance became up to 10%-15% slower than our best (fastest) estimation. This means that if a long dependency chain dominates

**Table 1.** Feature of dedicated hash functions

| Algorithms | Endianess | Message Block Size (bits) | Digest Size (bits) | Word Size (bits) | The Number of Steps | Message Scheduling |
|---|---|---|---|---|---|---|
| MD5 | Little | 512 | 128 | 32 | 64 | NO |
| RIPEMD-128 | Little | 512 | 128 | 32 | $2 \times 64$ | NO |
| RIPEMD-160 | Little | 512 | 160 | 32 | $2 \times 80$ | NO |
| SHA-1 | Big | 512 | 160 | 32 | 80 | YES |
| SHA-256 | Big | 512 | 256 | 32 | 64 | YES |
| SHA-512 | Big | 1024 | 512 | 64 | 80 | YES |
| Whirlpool | Neutral | 512 | 512 | - | $10^*$ | YES |

(*) Since Whirlpool has an architecture based on a block cipher, we will use the term "round" instead of "step" in this paper.

speed of an algorithm, Pentium may run in a smaller number of cycles than Pentium II/III.

Our implementation and performance measurement results show that, for MD5 and the RIPEMD family, the three-block parallel hashing on Pentium III reduces a cycle count of one block operation significantly as compared with the straightforward implementation. Also our instruction scheduling of SHA-1 and SHA-256 works excellently on Pentium III. In particular, the pipeline efficiency of SHA-1 reaches 2.52 $\mu$ops/cycle. Since at most two integer/logical $\mu$ops can be executed simultaneously on Pentium III, this shows that memory access instructions in the message scheduling part that was introduced in the SHA family are "hidden" in other logical and arithmetic instructions, which contributes to an effective use of the pipelines.

Another interesting result is that the two 512-bit hash functions SHA-512 and Whirlpool run almost at the same speed on Pentium III, while these algorithms have totally different architectures and design philosophies (and Whirlpool looks a much simpler algorithm). This does not seem to be a part of design principles of Whirlpool as far as we know. It should be also noted that since SHA-512 is designed for a pure 64-bit environment, it suffers performance penalties from some missing 64-bit instructions on Pentium III. Hence on a genuine 64-bit processor, SHA-512 might outperform Whirlpool.

## 2 Dedicated Hash Functions

Table 1 shows features of seven dedicated hash functions we deal with in this paper, and Table 2 summarizes the definitions of operations in these hash functions. So far there have been a lot of concrete proposals for efficient hash functions. The first constructions for hash functions were based on an efficient conventional encryption scheme such as DES. Although some trust has been built up in the security of these proposals, their software performance was not very well for the practical use, since they are typically a couple of times slower than the corresponding block cipher.

**MD4**, proposed by R. Rivest in 1990 [R90,R492], is the first dedicated hash function that targeted at *speed* in software (MD2 has a totally different archi-

**Table 2.** Definitions of the operations for dedicated hash functions

| **Algorithm:** Operations in one step | Nonlinear round functions at bit level |
|---|---|
| **MD5:**<br>Main stream<br>$A := B + (A + f_i(B,C,D) + X_{t[i]} + K_i)^{\lll s_i}$ | $f_i(x,y,z) = (x \wedge y) \vee (\neg x \wedge z)$ $\quad 0 \leq i \leq 15$<br>$f_i(x,y,z) = (x \wedge z) \vee (y \wedge \neg z)$ $\quad 16 \leq i \leq 31$<br>$f_i(x,y,z) = x \oplus y \oplus z$ $\quad 32 \leq i \leq 47$<br>$f_i(x,y,z) = y \oplus (x \vee \neg z)$ $\quad 48 \leq i \leq 63$ |
| **RIPEMD-128:**<br>Main streams 1 and 2<br>$A := B + (A + f_i(B,C,D) + X_{t[i]} + K_i)^{\lll s_i}$ | $f_i(x,y,z) = x \oplus y \oplus z$ $\quad 0 \leq i \leq 15$<br>$f_i(x,y,z) = (x \wedge y) \vee (\neg x \wedge z)$ $\quad 16 \leq i \leq 31$<br>$f_i(x,y,z) = (x \vee \neg y) \oplus z$ $\quad 32 \leq i \leq 47$<br>$f_i(x,y,z) = (x \wedge z) \vee (y \wedge \neg z)$ $\quad 48 \leq i \leq 63$<br>In their second stream,<br>$f_i$ is applied in the reversed order. |
| **RIPEMD-160:**<br>Main streams 1 and 2<br>$A := (A + f_i(B,C,D) + X_{t[i]} + K_i)^{\lll s_i} + E$<br>$C := C^{\lll 10}$ | $f_i(x,y,z) = x \oplus y \oplus z$ $\quad 0 \leq i \leq 15$<br>$f_i(x,y,z) = (x \wedge y) \vee (\neg x \wedge z)$ $\quad 16 \leq i \leq 31$<br>$f_i(x,y,z) = (x \vee \neg y) \oplus z$ $\quad 32 \leq i \leq 47$<br>$f_i(x,y,z) = (x \wedge z) \vee (y \wedge \neg z)$ $\quad 48 \leq i \leq 63$<br>$f_i(x,y,z) = x \oplus (y \vee \neg z)$ $\quad 64 \leq i \leq 79$<br>In their second stream,<br>$f_i$ is applied in the reversed order. |
| **SHA-1:**<br>Message scheduling<br>$X_i := (X_{i-3} \oplus X_{i-8} \oplus X_{i-14} \oplus X_{i-16})^{\lll 1}$<br>Main stream<br>$E := A^{\lll 5} + f_i(B,C,D) + X_i + K_i + E$<br>$B := B^{\lll 30}$ | $f_i(x,y,z) = (x \wedge y) \vee (\neg x \wedge z)$ $\quad 0 \leq i \leq 19$<br>$f_i(x,y,z) = x \oplus y \oplus z$ $\quad 20 \leq i \leq 39$<br>$f_i(x,y,z) = (x \wedge y) \oplus (x \wedge z) \oplus (y \wedge z)$ $\quad 40 \leq i \leq 59$<br>$f_i(x,y,z) = x \oplus y \oplus z$ $\quad 60 \leq i \leq 79$ |
| **SHA-256:**<br>Message scheduling<br>$X_i := \sigma_1(X_{i-2}) + X_{i-7} + \sigma_0(X_{i-15}) + X_{i-16}$<br>$16 \leq i \leq 63$<br>Main stream<br>$T_1 := H + \Sigma_1(E) + Ch(E,F,G) + K_i + X_i$<br>$T_2 := \Sigma_0(A) + Maj(A,B,C)$<br>$D := D + T_1$<br>$H := T_1 + T_2$ | $Ch(x,y,z) = (x \wedge y) \vee (\neg x \wedge z)$<br>$Maj(x,y,z)$<br>$= (x \wedge y) \oplus (x \wedge z) \oplus (y \wedge z)$<br>$\Sigma_0(x) = x^{\ggg 2} \oplus x^{\ggg 13} \oplus x^{\ggg 22}$<br>$\Sigma_1(x) = x^{\ggg 6} \oplus x^{\ggg 11} \oplus x^{\ggg 25}$<br>$\sigma_0(x) = x^{\ggg 7} \oplus x^{\ggg 18} \oplus x^{\gg 3}$<br>$\sigma_1(x) = x^{\ggg 17} \oplus x^{\ggg 19} \oplus x^{\gg 10}$ |
| **SHA-512:**<br>Message scheduling<br>$X_i := \sigma_1(X_{i-2}) + X_{i-7} + \sigma_0(X_{i-15}) + X_{i-16}$<br>$16 \leq i \leq 79$<br>Main stream<br>$T_1 := H + \Sigma_1(E) + Ch(E,F,G) + K_i + X_i$<br>$T_2 := \Sigma_0(A) + Maj(A,B,C)$<br>$D := D + T_1$<br>$H := T_1 + T_2$ | $Ch(x,y,z) = (x \wedge y) \vee (\neg x \wedge z)$<br>$Maj(x,y,z)$<br>$= (x \wedge y) \oplus (x \wedge z) \oplus (y \wedge z)$<br>$\Sigma_0(x) = x^{\ggg 28} \oplus x^{\ggg 34} \oplus x^{\ggg 39}$<br>$\Sigma_1(x) = x^{\ggg 14} \oplus x^{\ggg 18} \oplus x^{\ggg 41}$<br>$\sigma_0(x) = x^{\ggg 1} \oplus x^{\ggg 8} \oplus x^{\gg 7}$<br>$\sigma_1(x) = x^{\ggg 19} \oplus x^{\ggg 61} \oplus x^{\gg 6}$ |
| **Whirlpool:**<br>Message scheduling<br>$E_i^r = \sum_{j=0}^{7} \mathrm{Table}_j[(E^{r-1} \oplus K^{r-1})_{(i-j) mod 8, j}]$<br>Main stream<br>$X_i^r = \sum_{j=0}^{7} \mathrm{Table}_j[(X^{r-1} \oplus E^r)_{(i-j) mod 8, j}]$<br>$(1 \leq r \leq 10)$ | $a = \begin{pmatrix} a_0 \\ a_1 \\ \vdots \\ a_7 \end{pmatrix} = \begin{pmatrix} a_{00} & a_{01} & \cdots & a_{07} \\ a_{10} & a_{11} & \cdots & a_{17} \\ \vdots & \vdots & \ddots & \vdots \\ a_{70} & a_{71} & \cdots & a_{77} \end{pmatrix}$<br>$a = X, E$ or $K$ |

| | |
|---|---|
| $A, B, C, D, E, F, G, H$ | : Intermediate variables |
| $K$ | : Fixed constant value |
| $X$ | : Message block (and its derivatives) |
| $x^{\gg n}$, $x^{\ll n}$ | : Right/Left shift of $x$ by $n$ bits |
| $x^{\ggg n}$, $x^{\lll n}$ | : Right/Left rotate shift of $x$ by $n$ bits |

tecture and is slow). It was particularly designed to archive high performance on 32-bit processors that were the architecture of the future at the time. The algorithm is based on a simple set of primitive operations such as `add`, `xor`, `or` etc. on 32-bit words. Additionally, MD4 was designed to be favorable to a "little-endian" architecture, which obviously fits the Intel processor architecture perfectly. We do not deal with MD4 in this paper because it was broken several years ago [Dob98]. Almost all hash functions discussed hereafter are direct descendants of MD4 and inherit its structural characteristics. The **MD5** algorithm is an extension of the MD4 algorithm, increasing the number of steps from 48 to 64 [R592], and it is one of the most widely used dedicated hash algorithms in real applications. Some weaknesses of MD5 are reported in [Dob96][Rob96], although a collision of MD5 has not been found.

**RIPEMD-128** and **RIPEMD-160** [DBP96] evolved out of RIPEMD, which was developed as a strengthened version of MD4 by the RIPE consortium. RIPEMD-128 was developed out of RIPEMD-160 as a plug-in substitute for RIPEMD. They also have a little-endian architecture. One of the new characteristics of these hash functions is that they have essentially two parallel instances of MD4. Each of these instances includes 64 and 80 steps for RIPE MD-128 and RIPEMD-160, respectively. This structure has a potential ability to realize high performance when implemented on pipelined or superscalar execution mechanisms. Additionaly, each step of RIPEMD-160 has a rotate shift operation by 10 bits. Both of RIPEMD-128 and RIPEMD-160 are standardized as dedicated hash functions in ISO/IEC 10118 [ISO97].

**SHA-1** [FIP95] designed by NSA and published by NIST was also based on the design of MD4. The round functions of SHA-1 are exactly the same as those of MD4 while the total number of steps is 80. However, the SHA family went over to "big-endian", throwing away suitability for Intel processors. Moreover, it newly introduced a "message scheduling part". The design criteria of these new characteristics have not been made public. Although the role of the message schedule in the SHA family is similar to that of the key schedule in block ciphers, the message schedule must be done for each block like the on-the-fly implementation in block ciphers. The message schedule and main stream can be done independently, which means that it has a high capability for parallel execution, considering its rather high complexity. SHA-1 is also standardized as one of the dedicated hash functions in ISO/IEC 10118 [ISO97].

Recently **SHA-256** and **SHA-512** [FIP01] were proposed to keep up with possible future applications where longer message digests are required . They have almost the same basic components, except that SHA-256 operates on eight 32-bit words, while SHA-512 operates on eight 64-bit words. SHA-512 is the first dedicated hash function designed for a genuine 64-bit processor. SHA-256 and SHA-512 also have a message schedule part, which has a much more complex structure than that of SHA-1, including many shift operations in both message scheduling part and main stream. Another feature of these algorithms is that exactly the same operations are used in all steps.

**Whirlpool** does not belong to the MD-family nor to the SHA-family. It directly inherits its structure from Rijndael. The state of Whirlpool consists of a 8×8 matrix over $GF(2^8)$, while that of Rijndael was designed on a 4×4 matrix over $GF(2^8)$. A transformation from a state to a next state is given by eight 8-bit to 64-bit look-up tables. Hence Whirlpool demonstrates best performance in true 64-bit environments, but can also realize good performance on any processor. Whirlpool has a message scheduling part, which is exactly the same as its main stream. Whirlpool is endian-neutral. Recently the designer of Whirlpool announced a tweak of the algorithm to improve its hardware efficiency. SHA-256, -512 and this tweak of Whirlpool are under discussion for an inclusion in the next version of the ISO/IEC 10118 standard.

## 3 A Brief Overview of Pentium III Processor

Pentium III supports all x86 instructions with eight 32-bit registers, and additional MMX instructions with eight 64-bit registers (MMX registers). The main motivation of the MMX instructions is to enable 16×4 / 32×2 parallel operations for multimedia applications. Although Pentium III is not a full 64-bit processor such as the Alpha processor, — a 64-bit addition instruction is missing, for instance —, the MMX instructions are attractive to wider applications since most of them work in one cycle including a 64-bit memory load instruction. Pentium III also provides XMM instructions (SSE) with eight 128-bit registers, but we are not able to use them for fast hashing because they operate on floating-point data elements.

The following shows some of the essential topics for optimizing on Pentium II/III, which greatly owes to an excellent guidebook for optimizing Pentium family written by Agner Fog [Fog00].

**Instruction Decoding** In the decoding stage, instruction codes are broken down into simpler micro-operations ($\mu$ops), where one instruction usually consists of one to four $\mu$ops. Pentium II/III has three decoders that can work in parallel and theoretically can generate six $\mu$ops per cycle. However due to limitations coming from instruction fetch rules and decoder capabilities, the code length and order of instructions heavily affect efficiency of the decoding speed. A typical decoding rate of real applications is two to three $\mu$ops per cycle on an average.
The decoding speed is not an important issue for MD5 and the RIPEMD family since the performance bottleneck of these hash functions is actually a long data dependency chain, but it can be a critical factor for algorithms with high parallel computation capabilities such as the SHA family. Generally, optimizing decoding performance without causing other pipeline stalls is an extremely difficult puzzle to solve.

**Register Renaming** After the decoding stage, all permanent registers (x86 and MMX registers) are renamed into internal registers. This mechanism solves fake data dependency chains caused by register starvation. The renaming is controlled by the register alias table (RAT), which can handle only three $\mu$ops per cycle. This means that overall performance of Pentium II/III can not exceed three $\mu$ops per cycle. Moreover the RAT can read two permanent registers in a cycle, and hence depending on the situation we should use an absolute addressing mode instead of a register indirect addressing mode.

Some hash functions repeatedly use a fixed constant value. If a register is free, it is common to assign the constant value to the register, which is best in terms of instruction length. However in order to avoid the register read stalls, we often handled the value using an immediate addressing mode if possible, or an absolute addressing mode via memory, particularly in the case of MMX instructions.

**Execution Units** Pentium II/III has five independent execution ports p0 to p4 to carry out a sequence of $\mu$ops in an out-of-order manner. p0 and p1 are mainly for arithmetic and logical operations, p2 for load and p3 and p4 for store operations. An important consequence of this architecture is that if arithmetic and logical operations are a dominant factor of performance, load and store operations can be "hidden" behind them. In other words, we should store a temporary value not in a free register but in memory in such a case.

The SHA family requires memory operations more frequently than MD5 and the RIPEMD family due to the existence of the message scheduling part, and also due to a large hash size for SHA-256 and SHA-512. If properly implemented these hash functions can take full advantage of hiding load/store operations behind other operations.

**Other topics** In Pentium II/III, a partial register/memory access (reading from a register/memory after writing to a part of it) causes a heavy performance penalty; i.e. a register/memory must be basically read/written in the same size at the same memory address. In this paper, the partial memory access can be a problem only in the 64-bit oriented hash functions SHA-512 and Whirlpool as will be shown in a later section.

According to [Fog00], the retirement stage of $\mu$ops can be a bottleneck of performance. We however can do little about this stall factor, while the partial register/memory stalls can be avoided by programmers.

## 4 API and How to Measure Performance

We developed our assembly language programs on the following hardware and software environments. The size of RAM memory is not an important issue because we designed the programs so that everything could be on the first level cache (code 16KB + data 16KB).

**Hardware:** IBM Compatible PC with Pentium III 800MHz and 256MB RAM
**Software:** Windows 98, Visual C++ 6.0, MASM 6.15

We described a hashing logic in a subroutine form callable from C language and measured its execution time from outside the subroutine, which reflects hashing performance in real applications. The subroutine API is that an input message to be compressed and a resultant hashed value are represented as a byte sequence passed by pointers, which is also common as an interface of hash functions. Note that this means that big-endian algorithms, specifically the SHA family, must perform a byte-order swap operation inside the subroutine and it is counted as a part of hashing time. We assume a padding operation is done outside this routine.

In the next section, we will give a method of coding for hashing two or three independent messages of the same block length in parallel. For simplicity, we adopted the same API for this method as that for a straightforward implementation, appending the second/third message to the first/second message for each block. Hence a procedure for combining two separate 4-byte words into one 8-byte MMX register is also counted as a part of hashing time. If we interleave different messages in every word (4-byte), we can skip this procedure, but did not adopt this approach because assuming such interface looks uncommon in real applications.

We actually measured time for hashing a total of 8KB message bytes, which is half of the first level data cache size. To maximize speed, we unrolled an inner loop as long as the code is fully covered by the first level code cache. The cycle counting was done using the `rdtsc` (read time stamp counter) instruction, as shown in [Fog00]. The timing was measured several times to remove possible negative effects on performance due to interrupts by an operating system. Also note that our programs are neither data-dependent nor self-modified.

# 5 Implementations of Hash Functions on Pentium III

## 5.1 Implementation methods

We designed three types of assembly language codes for 32-bit oriented hash functions (MD5, RIPEMD-128 -160, SHA-1 -256); namely, straightforward, two-block parallel and three-block parallel as follows. The second and third methods fully extract the power of parallel execution capabilities of MMX. For 64-bit oriented hash functions (SHA-512 and Whirlpool), we implemented the straightforward version only (no way to parallelize).

Method 1: Straightforward

**MD5, RIPEMD-128, RIPEMD-160, SHA-1**

All words of the main stream are always held in four or five x86 registers and the remaining registers are used as temporary variables. This is the most common implementation of dedicated hash functions in software.

**SHA-256, SHA-512**

All eight 32-bit/64-bit words of the main stream are stored in memory because of the register starvation of the Pentium family, and x86/MMX registers are basically used only as temporary variables. Additionally, for SHA-512, 32-bit x86 registers are essentially used to realize a 64-bit addition operation, which is a missing instruction in Pentium III.

**Whirlpool**

A 64-byte state matrix is always held in all eight MMX registers. However, since the state information of the preceding round is necessary to generate the state matrix of the next round, it must be, after all, stored in memory at the end of each round.

### Method 2: Two-block parallel

This method is applied only to 32-bit oriented hash functions (**MD5, RIPEMD-128, RIPEMD-160, SHA-1, SHA-256**), where a word of one message is loaded on the upper half of 64-bit register/memory and a word of another message is loaded on the lower half. This enables fast parallel hashing computation of two independent messages, but the following penalties peculiar to Pentium III should be taken into consideration.

1. The code length of an MMX instruction is usually longer (typically by one byte) than that of an equivalent x86 instruction, which may lead to a performance penalty due to an inefficient instruction decoding.
2. Since MMX instructions do not have an immediate addressing mode, a 64-bit immediate value must be processed via memory, which increases the number of memory access $\mu$ops (p2, p3 and p4). But this penalty is often able to be hidden in integer $\mu$ops (p0 and p1).
3. A parallel rotate shift instruction is missing (if a shift count is not a multiple of 16). To do this on Pentium III, four instructions are needed; that is, `movq` (copy), `psrld` (right shift), `pslld` (left shift) and `pxor` (xor).

Our implementation will show that gains by parallel computation exceed the penalties caused by these factors.

### Method 3: Three-block parallel

This method is a combination of the two methods above, which is hence applicable to 32-bit oriented dedicated hash functions. Since methods 1 and 2 use different types of registers, these two programs can "coexist", that is, can be executed in parallel without depriving each other of hardware resources. Although this is not an essential methodological improvement of implementation, it is expected that a better instruction scheduling leads to further improvement of hashing speed, particularly if a long data dependency chain dominates the speed of a target algorithm. Possible penalties that should be noted in this case are:

1. Code size: The code size becomes big because the entire code is simply a merged combination of the two implementations. But this does not lead to an actual big penalty issue as long as the entire code is within the first level cache.
2. Register read stall: The heavy simultaneous use of x86 and MMX registers easily create register read stalls. It is often very difficult to completely remove the possibility of this stall without causing other penalties.

Our implementation will show that this three-block method actually gives a performance improvement for MD5, RIPEMD-128, RIPEMD-160.

### 5.2 Implementation results and discussions

Table 7 shows our implementation results of the seven dedicated hash functions. We manually counted the number of $\mu$ops of one block operation, referring to [Fog00]. "p01" denotes logical and integer $\mu$ops that use pipes p0 and/or p1. "p2" and "p34" denote memory read/write $\mu$ops that use pipes p2 and p3, p4, respectively. Note again that the cycle counts of the SHA family include time for endian conversion and that the cycle counts of methods 2 and 3 include time for merging two 32-bit words into one 64-bit register.

**MD5**

The performance of our straightforward implementation of MD5 on Pentium III is approximately the same as that shown in [Bos97] on Pentium. This reflects the fact that the frequency of memory access operations is very low and a long data dependency chain is an actual dominant factor of its hashing speed. In fact, the $\mu$ops/cycle value of our program is only 1.64 while the maximal performance that Pentium III can achieve is 3 $\mu$ops/cycle.

The two-block and three-block parallel implementations significantly improve the efficiency of hashing. In particular, it can be seen that the three-block version is almost perfectly scheduled (1.83 p01 $\mu$ops/cycle) and 30% faster than the straightforward version.

**RIPEMD-128, RIPEMD-160**

The same tendency can be seen for RIPEMD-128 and RIPEMD-160 as for MD5. One possible reason that RIPEMD-160 has a better $\mu$ops/cycle value for two-block and three-block versions is the existence of the operation "$B^{\lll 10}$" that was introduced in RIPEMD-160. This operation takes four $\mu$ops on MMX, which can be executed independently with other operations.

[**Note**] Since the RIPEMD family has two parallel instances, it is possible to assign each of the two instances to each of two types of registers, that is, one to 32-bit registers and the other to 64-bit MMX registers. This makes a parallel execution *inside* one message block possible, but unfortunately this did not run very fast, probably because assigning the full 64-bit registers for only one instance execution was too inefficient reducing overall performance. Another possibility to utilize the parallelism of the RIPEMD family might be interleaving the first instance and the second instance (our current code executes the second one after finishing the first one). Although this method leads to an increase in a total number of $\mu$ops due to the register starvation, the parallelism of the resultant code will be improved. However we have not succeeded, so far, in a speed-up of a straightforward implementation of RIPEMD-128 or RIPEMD-160 using this technique.

**SHA-1**

One big difference between the SHA family and the (RIPE)MD family is

**Table 3.** Coding example of one step of round 2 of RIPEMD-128 on a Pentium III processor. The chaining variable $A, B, C, D$ is stored in x86 registers `eax` through `edx` or MMX registers `mm0` through `mm3`. In the three-block parallel implementation, these codes are interleaved.

| Straightforward code using x86 | | Two-block code using MMX | | |
|---|---|---|---|---|
| mov | esi, edx | paddd | mm0, [eax+8*X] | |
| add | eax, (X+4)[edi] | movq | mm5, mm3 | |
| xor | esi, ecx | pxor | mm5, mm2 | |
| and | esi, ebx | paddd | mm0, K | |
| xor | esi, edx | pand | mm5, mm1 | |
| lea | eax, [eax+esi+K] | pxor | mm5, mm3 | |
| rol | eax, s | paddd | mm0, mm5 | |
| | | movq | mm5, mm0 | ; left rotate |
| | | pslld | mm0, s | ; shift of `mm0` |
| | | psrld | mm5, 32-s | ; by $s$ bits |
| | | pxor | mm0, mm5 | ; |

the existence of a message scheduling part. The message scheduling part of the SHA family is independent of the main stream and contains many memory access operations enabling the straightforward SHA-1 implementation to optimally exploit the increased hardware parallelism of Pentium III. The value 2.52 $\mu$ops/cycles of our straightforward version is the highest of our programs.

Because of this highly parallel feature of SHA-1, the performance improvement of the two-block version is not so big as that of the (RIPE)MD family. Moreover the three-block version is rather slower than the two-block version. This suggests that, as far as an instruction scheduling efficiency of SHA-1 is concerned, our implementation has reached almost an optimal level on Pentium III (1.82 p01 $\mu$ops/cycle).

**SHA-256**

SHA-256 (and SHA-512) uses eight words and it is no longer possible to keep all internal values on the permanent registers. They must be stored in memory and be read from/written to memory in each step, which inevitably increases the frequency of memory access. However, since these memory access $\mu$ops can be mostly carried out in parallel with logical and integer uops, the pipeline scheduling of SHA-256 works excellently (2.32 $\mu$ops/cycle).

The two-block parallel implementation of SHA-256 runs 15% faster than the straightforward implementation, but this improvement is smaller than that in other hash algorithms. This is because SHA-256 (and SHA-512) has many rotate shift operations in both the message scheduling part and the main stream, which causes a non-negligible increase of the number of $\mu$ops when realized in 64-bit MMX registers. The performance of the three-block parallel version was not good for the same reason as for SHA-1.

**Table 4.** Coding example of one step of round 2 of SHA-1 on a Pentium III processor. The chaining variable $A, B, C, D, E$ is stored in x86 registers `eax,ebx,ecx,edx,ebp` or MMX registers `mm0` through `mm4`. In the three-block parallel implementation, these codes are interleaved. Instructions marked "m.s." are for message scheduling part.

| Straightforward code using x86 | | | Two-block code using MMX | | |
|---|---|---|---|---|---|
| `mov` | `esi,W+(s*4)` | ; m.s. | `movq` | `mm6,WW+(s*8)` | ; m.s. |
| `mov` | `edi,edx` | | `movq` | `mm5,mm1` | |
| `xor` | `esi,W+(s1*4)` | ; m.s. | `pxor` | `mm6,WW+(s1*8)` | ; m.s. |
| `xor` | `edi,ecx` | | `pxor` | `mm5,mm2` | |
| `xor` | `esi,W+(s2*4)` | ; m.s. | `pxor` | `mm6,WW+(s2*8)` | ; m.s. |
| `xor` | `edi,ebx` | | `pxor` | `mm5,mm3` | |
| `xor` | `esi,W+(s3*4)` | ; m.s. | `pxor` | `mm6,WW+(s3*8)` | ; m.s. |
| `add` | `ebp,edi` | | `paddd` | `mm4,mm5` | |
| `rol` | `esi,1` | ; m.s. | `movq` | `mm7,mm6` | ; m.s. |
| `mov` | `edi,eax` | | `paddd` | `mm4,CONSTonMEM` | |
| `add` | `ebp,esi` | | `paddd` | `mm6,mm6` | ; m.s. |
| `rol` | `edi,5` | | `psrld` | `mm7,31` | ; m.s. |
| `rol` | `ebx,30` | | `movq` | `mm5,mm1` | |
| `lea` | `ebp,[ebp+edi+CONST]` | | `pxor` | `mm6,mm7` | ; m.s. |
| `mov` | `W+(s*4),esi` | ; m.s. | `psrld` | `mm1, 2` | |
| | | | `paddd` | `mm4, mm6` | |
| | | | `movq` | `WW+(s*8),mm6` | ; m.s. |
| | | | `pslld` | `mm5,30` | |
| | | | `movq` | `mm6,mm0` | |
| | | | `movq` | `mm7,mm0` | |
| | | | `pslld` | `mm6,5` | |
| | | | `paddd` | `mm4, mm6` | |
| | | | `psrld` | `mm7,27` | |
| | | | `pxor` | `mm1,mm5` | |
| | | | `paddd` | `mm4,mm7` | |

[**Note**] The architecture of SHA-256 is quite different from that of SHA-1. But interestingly, the rate of the number of memory access $\mu$ops to all $\mu$ops is almost the same (approximately 30%), which is ideal in terms of Pentium III scheduling. Is this a hidden design criteria of the SHA family??

**SHA-512**

SHA-512 is a 64-bit oriented hash function and hence it is essential to utilize MMX registers and instructions. However, since Pentium III does not have a 64-bit addition instruction (but fortunately it does have a 64-bit shift operation! Pentium 4 supports both instructions), we realized it by simply combining two 32-bit additions. Although this naive method obviously suffers a non-negligible performance penalty from 32-bit from/to 64-bit register transfer, we do not know a faster method to do this. Another penalty that we should note is a partial memory access. Below left is a straightforward method for coding the last part of one step, but this causes a partial memory access at the beginning of the next step. Below right is the corrected code we adopted, which is free from the penalty and additionally saves one $\mu$op.

The pipeline scheduling of our implementation is good (2.24 $\mu$ops/cycle), but we feel that there is still room for performance improvement since the hash algorithm itself allows higher parallel execution capability.

**Table 5.** 64-bit addition and partial memory stall on a Pentium III processor.

```
                        μops
add  [mem+0],eax ; 4
adc  [mem+4],ebx ; 6



(next step)
movq mm0,[mem]   ; 1 ← Stall
```

```
                                μops
add        eax,[mem+0] ; 2
adc        ebx,[mem+4] ; 3
movd       mm0,eax     ; 1
movd       mm1,ebx     ; 1
punpckldq  mm0,mm1     ; 1
movq       [mem],mm0   ; 2
```

**Whirlpool**

The simplest realization of Whirlpool is to keep the state matrix in eight MMX registers and eight 8-bit to 64-bit look-up tables on memory, but these tables completely cover all the 16KB data cache of Pentium III, which leads to data cache miss penalties. We hence had only four tables on memory and generated the remaining from them when necessary using the `pshufw` (packed shuffle word) instruction.

This instruction works only in Pentium III (not in Pentium II and Celeron); all other instructions throughout our codes run on Pentium II and Celeron. Also taking into consideration reducing the frequency of memory access and removing the possibility of partial register/memory stalls, we wrote the entire algorithm by repeating the piece of code shown in Table 6.

The pipeline of this program works very well, achieving 2.29 $\mu$ops/cycle. The resultant hashing speed (cycles/byte) is more than 3.5 times faster than that

**Table 6.** An essential part of Whirlpool on a Pentium III processor.

```
mov   edx,[Matrix Address] ; preceding state matrix
movzx esi, dl              ; address generation
pxor  mm0, Table1[esi*8]   ; current state matrix
movzx esi, dh
pxor  mm1, Table2[esi*8]
shr   esi, 16
movzx esi, dl
pxor  mm2, Table3[esi*8]
movzx esi, dh
pxor  mm3, Table4[esi*8]
```

**Table 7.** Performance Figure

| Algorithm | method | $\mu$ops/block [p01] | $\mu$ops/block [p2] | $\mu$ops/block p[34] | $\mu$ops/block [total] | cycles /block | cycles /byte | $\mu$ops/cycle [p01] | $\mu$ops/cycle [total] | size (bytes) code | size (bytes) data | Pentium[Bos97] #inst. | Pentium[Bos97] cycles | Pentium[Bos97] i/c |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| MD5 | 1 | 503 | 69 | 10 | 582 | 354 | 5.53 | 1.42 | 1.64 | 1750 | 32 | 577 | 337 | 1.71 |
| MD5 | 2 | 783 | 148 | 40 | 971 | 276 | 4.31 | 1.42 | 1.76 | 3331 | 704 | | | |
| MD5 | 3 | 1283 | 217 | 50 | 1550 | 234 | 3.66 | 1.83 | 2.21 | 5259 | 736 | | | |
| RIPEMD-128 | 1 | 875 | 141 | 18 | 1034 | 602 | 9.41 | 1.45 | 1.72 | 2707 | 48 | 1024 | 592 | 1.73 |
| RIPEMD-128 | 2 | 1379 | 252 | 48 | 1679 | 477 | 7.45 | 1.45 | 1.76 | 5703 | 280 | | | |
| RIPEMD-128 | 3 | 2251 | 393 | 66 | 2710 | 425 | 6.64 | 1.77 | 2.13 | 8379 | 320 | | | |
| RIPEMD-160 | 1 | 1421 | 176 | 22 | 1619 | 911 | 14.23 | 1.56 | 1.78 | 4227 | 56 | 1639 | 1013 | 1.62 |
| RIPEMD-160 | 2 | 2533 | 383 | 52 | 2968 | 738 | 11.53 | 1.72 | 2.01 | 10110 | 320 | | | |
| RIPEMD-160 | 3 | 3951 | 559 | 74 | 4584 | 726 | 11.34 | 1.81 | 2.10 | 14305 | 376 | | | |
| SHA-1 | 1 | 1100 | 295 | 174 | 1569 | 623 | 9.73 | 1.77 | 2.52 | 4664 | 356 | 1469 | 837 | 1.76 |
| SHA-1 | 2 | 1928 | 389 | 170 | 2487 | 531 | 8.30 | 1.82 | 2.34 | 8567 | 248 | | | |
| SHA-1 | 3 | 2977 | 684 | 376 | 4037 | 559 | 8.73 | 1.78 | 2.41 | 13175 | 340 | | | |
| SHA-256 | 1 | 2491 | 609 | 418 | 3518 | 1519 | 23.73 | 1.64 | 2.32 | 10202 | 144 | | | |
| SHA-256(*) | 2 | 4385 | 689 | 418 | 5492 | 1318 | 20.59 | 1.66 | 2.08 | 6705 | 1232 | | | |
| SHA-256(*) | 3 | 6921 | 1352 | 852 | 9125 | 1417 | 22.14 | 1.63 | 2.15 | 14142 | 1816 | | | |
| SHA-512(*) | 1 | 8828 | 1522 | 1186 | 11536 | 5143 | 40.18 | 1.72 | 2.24 | 7203 | 1496 | | | |
| Whirlpool(*) | 1 | 3328 | 1634 | 384 | 5346 | 2337 | 36.52 | 1.42 | 2.29 | 2456 | 8496 | | | |

(*) only partially loop-unrolled in a message block to reduce code size within the first level cache (16KB). All other implementations are fully loop-unrolled.

of the designers' C implementation [BR00]. Very interestingly, this performance is almost the same as that of SHA-512; Whirlpool is slightly (within 10%) faster than SHA-512, but this difference looks within "a margin of implementation".

## 6 Conclusions

This paper showed a performance analysis and speeding-up method of dedicated hash functions on the Pentium III processor. A further improvement of software performance is ongoing.

An overall performance of a processor can be achieved by (1) high instruction parallelism, (2) high SIMD parallelism and (3) high clock frequency. The index "cycle/byte" is the most common performance measure of cryptographic algorithms, which direct reflects (1) and (2), but not (3). So we should note that this measure is appropriate to compare software performance of given cryptographic algorithms on a fixed target processor, but not necessarily appropriate to evaluate hardware performance of given processors on a fixed cryptographic algorithm.

High processor performance has been achieved by improving (1) and (3) in the past, but in recent processors this seems to be rapidly shifting to (2) and (3). This means that use of parallel execution of multiple blocks will be much more important and have a practical impact in near future.

## Acknowledgments

We would like to thank Antoon Bosselaers for his careful reading and many valuable remarks.

## References

[BR00] P. Barreto, V. Rijmen, "The Whirlpool hashing function," *First open NESSIE Workshop record,* Leuven, 13-14 November 2000. The document is available at http://www.cryptonessie.org/workshop/submissions/whirlpool.zip.

[BGV96] A. Bosselaers, R. Govaerts, J. Vandewalle, "Fast hashing on Pentium," *Advances in Cryptology, Proceedings Crypto '96, LNCS 1109,* N. Koblitz, Ed., Springer-Verlag, 1996, pp. 298-312.

[BGV97] A. Bosselaers, R. Govaerts and J. Vandewalle, "SHA: A design for parallel architectures?," *Advances in Cryptology, Proceedings Eurocrypt'97, LNCS 1233,* W. Fumy, Ed., Springer-Verlag, 1997, pp. 348-362.

[Bos97] A. Bosselaers, "Even faster hashing on the Pentium," presented at the rump session of Eurocrypt'97. Available at http://www.esat.kuleuven.ac.be/cosicart /pdf/AB-9701.pdf.

[Dob98] H. Dobbertin, "Cryptanalysis of MD4," *J. Cryptology,* Vol. 11, pp. 253-271, 1998.

[Dob96] H. Dobbertin, "The status of MD5 after a recent attack," *Cryptobytes,* Vol. 2, No. 2, pp. 1-6, 1996. Available at ftp://ftp.rsasecurity.com/pub/cryptobytes/ crypto2n2.pdf

[DBP96] H. Dobbertin, A. Bosselaers, B. Preneel, "RIPEMD-160, a strengthened version of RIPEMD," *Fast Software Encryption, LNCS 1039,* D. Gollmann, Ed., Springer-Verlag, 1996, pp. 71-82. The final version is available at http:// www.esat.kuleuven.ac.be/cosicart/pdf/AB-9601.pdf

[FIP95] Federal Information Processing Standards (FIPS) Publication 180-1, *Secure Hash Standard (SHS),* U.S. DoC/NIST, April 17, 1995.

[FIP01] Draft Federal Information Processing Standards (FIPS) Publication 180-2, *Secure Hash Standard (SHS),* U.S. DoC/NIST, May 30, 2001.

[Fog00] Agner Fog, How to Optimize for the Pentium Microprocessors, 03 July 2000. Available at http://www.agner.org/assem/

[Int01] Intel, *Intel Architecture Optimization. Reference Manual,* 1999. Order Number 245127-001. Available at http://www.intel.com/design/pentiumIII/manuals/

[Int02] Intel, *Intel Architecture Optimization Manual,* 1997. Order Number 242816-003. Available at http://www.intel.com/design/pentium/manuals/

[Int03] Intel, *Intel Architecture Software Developer's Manual,* 2001.
Volume 1 Basic Architecture (Order Number 245470)
Volume 2 Instruction Set Reference (Order Number 245471)
Volume 3 System Programming Guide (Order Number 245472)
Available at http://www.intel.com/design/pentiumIII/manuals/

[ISO97] ISO/IEC 10118-3, *"Information technology - Security techniques - Hash-functions -Part 3: Dedicated hash-functions,"* IS 10118, 1997.

[Lip98] H. Lipmaa, "IDEA, A Cipher for Multimedia Architectures?," *Selected Areas in Cryptography '98, LNCS 1556,* Henk Meijer, Eds., Springer-Verlag, 1998, pages 248–263. Available at http://www.tcs.hut.fi/ helger/papers/lip98/.

[PRB98] B. Preneel, V. Rijmen, A. Bosselaers, "Recent developments in the design of conventional cryptographic algorithms," *Computer Security and Industrial Cryptography, State of the Art and Evolution, LNCS 1528,* B. Preneel, V. Rijmen, Eds., Springer-Verlag, 1998, pp. 106-131.

[R90] R.L. Rivest, "The MD4 message digest algorithm," *Advances in Cryptology, Proceedings Crypto '90, LNCS 537,* S. Vanstone, Ed., Springer-Verlag, 1991, pp. 303-311.

[R492] R.L. Rivest, "The MD4 message-digest algorithm," *Request for comments (RFC) 1320,* Internet Activities Board, Internet Privacy Task Force, April 1992.

[R592] R.L. Rivest, "The MD5 message-digest algorithm," *Request for comments (RFC) 1321,* Internet Activities Board, Internet Privacy Task Force, April 1992.

[Rob96] M. Robshaw, "On recent results for MD2, MD4 and MD5," *RSA laboratories' Bulletin,* No. 4, November 1996. Available at ftp://ftp.rsasecurity.com/pub/pdfs/bulletn4.pdf