Fast Hashing on the Pentium

Antoon Bosselaers, René Govaerts and Joos Vandewalle

Katholieke Universiteit Leuven, Dept. Electrical Engineering-ESAT
Kardinaal Mercierlaan 94, B-3001 Heverlee, Belgium

`antoon.bosselaers@esat.kuleuven.ac.be`

25 May 1996

Abstract. With the advent of the Pentium processor parallelization finally became available to Intel based computer systems. One of the design principles of the MD4-family of hash functions (MD4, MD5, SHA-1, RIPEMD-160) is to be fast on the 32-bit Intel processors. This paper shows that carefully coded implementations of these hash functions are able to exploit the Pentium's superscalar architecture to its maximum effect: the performance with respect to execution on a non-parallel architecture increases by about 60%. This is an important result in view of the recent claims on the limited data bandwidth of these hash functions. Moreover, it is conjectured that these implementations are very close to optimal. It will also be shown that the performance penalty incurred by non-cached data and endianness conversion is limited, and in the order of 10% of running time.

Key words. Cryptographic hash functions, Parallel implementation, Software performance, Pentium processor.

1 Introduction

A cryptographic hash function h maps bitstrings of arbitrary finite length into strings of fixed length. Given h and an input x, computing $h(x)$ must be easy. A *one-way hash function* must provide both *preimage resistance* and *second preimage resistance*, i.e., it must be computationally infeasible to find, respectively, any input which hashes to any pre-specified output, and any second input which has the same output as any specified input. For an *ideal* one-way hash function with m-bit result, finding a preimage or a second preimage requires about 2^m operations. A *collision resistant hash function* is a one-way hash function that provides the additional property of *collision resistance*, i.e., it must be computationally infeasible to find two distinct inputs that hash to the same result. For an *ideal* collision resistant hash function with m-bit result, no attack finding a collision requires less work than a birthday or square root attack of about $2^{m/2}$ operations [Pre94].

The most popular hash functions, currently used in a wide variety of applications, are the custom designed iterative hash functions from the MD4-family. MD4 was introduced in 1990 by R. Rivest [Riv92a]. One of the design principles was to be fast on 32-bit machines in general, and on the Intel x86 family in particular. The latter is more or less a must, because of the pervasiveness of the x86 processor family. Or, as P. Rogaway and D. Coppersmith put it, by doing well on these 'difficult-to-optimize-for vehicles' [RoCo94], one expects to do well on any modern 32-bit processor.

Since the introduction of MD4, and as a result of developments in cryptanalysis (see [Rob95] for an overview, and [Dob96a,Dob96b] for the most recent results) a whole family of MD4-like hash functions has been developed. All these descendants aim at strengthening their ancestors, taking into account the existing attacks at the moment of their introduction: MD5 ('91, [Riv92b]), SHA-1 ('94, [FIPS180-1]), RIPEMD ('92, [RIPE95]), RIPEMD-128 and RIPEMD-160 ('96, [DBP96]). Their common MD4-ancestry resulted in still fairly fast implementations on 32-bit architectures, but their increased complexity nevertheless degraded their performance.

All these hash functions have been designed with the first generation of 32-bit Intel processors in mind: the i386, introduced in October 1985, and the i486, introduced in August 1989. As expected, these hash functions could, without too much difficulty, be implemented efficiently on these processors. The advent of the Pentium processor marks the beginning of a new generation of 32-bit Intel processors. More RISC (Reduced Instruction Set Computer) aspects than ever before have been incorporated in this from origin CISC (Complex Instruction Set Computer) processor. From the outside the Pentium might look like a CISC, inside it is definitely more RISC than CISC. The processor's crucial architectural innovation is the ability to issue, under certain conditions, two instructions at once, thanks to its twin superscalar pipelines. It turns out that, although this was certainly not one of the design principles, the MD4-family fits the Pentium's superscalar architecture very nicely, boosting the performance of these hash functions to unprecedented levels. It is conjectured that our implementations are very close to optimal, and that on a Pentium architecture it will be very hard to improve on the presented performance figures. This is a significant result taking into account the importance of performant hash functions in many cryptographic applications, and the fact that some of the MD4-like hash functions will be around for some years to come.

The next section gives a comparative overview of the MD4-family members from a performance point of view. Section 3 gives an overview of the Pentium architecture, and concentrates on its superscalar features. In Section 4 it is shown how the latter can be used to improve performance of MD4-like hash functions considerably. The actual performance figures for the 6 hash functions discussed in this paper are given in Section 5. Section 6 discusses two data related topics: cacheing and string-integer transformation, and their impact on performance. Finally, Section 7 formulates the conclusions.

2 Comparative description of the MD4-family

The six members of the MD4-family are iterative hash functions operating on 32-bit words. For a full description of these hash functions we refer to the references given in the introduction. This section will only describe them as far as performance is concerned. The different compression functions take as input a 4 or 5-word chaining variable and a 64-byte message block, and map this to a new chaining variable. All operations are defined on 32-bit words. First, the 64-byte message block is converted to a block of 16 words using one of two possible string-integer conversions. Next and depending on the algorithm, 3 to 5, possibly parallel, rounds are applied. Each of these rounds consists of 16 individual steps, except for SHA-1, where rounds of 20 steps are used. Finally, the previous value of the chaining variable is added to the newly obtained value by means of a feedforward. Every round uses a particular non-linear function, and every step modifies one word of the chaining variable and possibly rotates another. Table 1 summarizes the definitions of a step function for the 6 hash functions considered.

Algorithm	Step function
MD4	$A := (A + f(B, C, D) + X_i + K)^{\lll s}$
MD5	$A := B + (A + f(B, C, D) + X_i + K)^{\lll s}$
SHA-1	from step 17 onwards: $X_i := (X_i \oplus X_{i+2} \oplus X_{i+8} \oplus X_{i+13})^{\lll 1}$ $A := A + B^{\lll 5} + f(C, D, E) + X_i + K$ $C := C^{\lll 30}$
RIPEMD	$A := (A + f(B, C, D) + X_i + K)^{\lll s}$
RIPEMD-128	$A := (A + f(B, C, D) + X_i + K)^{\lll s}$
RIPEMD-160	$A := E + (A + f(B, C, D) + X_i + K)^{\lll s}$ $C := C^{\lll 10}$

Table 1. Definition of a step for the MD4-family of hash functions. All additions are modulo 2^{32}, and $\lll s$ indicates a rotation over s bits to the left. A, B, C, D, E are chaining variable words, K and s are constants, and $f()$ is a non-linear function of three variables. X_i is a message word or, in case of SHA-1 and from step 17 onwards, an exor combination of message words.

Each step in a round uses a different message word X_i, and in each round the order in which they are used is different. SHA-1 differs in this respect from the other hash functions in that starting from step 17 (i.e., once every message word has been used once) a linear recursion is applied to the array of 16 message words: every element of the array is computed as the exor of four other elements. Any message bit is now input to at least 28 and at most 36 steps [Pre93]. The additive constants K are different per round, except for MD5, where each step has a different K. The rotation constants s are different per round and per chaining variable for MD4 and MD5, are fixed for SHA-1, and are different per round and per message word for the RIPEMD-sisters. The Boolean functions $f()$ are different for each round, but are chosen from a limited set of four, summarized in Table 2.

These descriptions lead to a number of important observations from an implementation point of view.

Multiplexer	$(x \wedge y) \vee (\overline{x} \wedge z)$	all
Majority	$(x \wedge y) \vee (x \wedge z) \vee (y \wedge z)$	MD4, RIPEMD, SHA-1
Exor	$x \oplus y \oplus z$	all
Or-Exor	$(x \vee \overline{z}) \oplus y$	MD5, RIPEMD-128, RIPEMD-160

Table 2. Boolean functions of the MD4-family. The last column indicates where they are used.

Simple instructions The compression function can be implemented with a limited number of simple instructions on 32-bit words: assignment of one word to another (`mov`), addition modulo 2^{32} of two words (`add`, `lea`), unary or binary logical operations (`not`, `and`, `or`, `xor`), and rotation of a word over a number of bits to the left (`rol`).

Small buffer size The chaining variable consists of 4 or 5 words only. Therefore, it can constantly be kept in registers during an iteration of the compression function. Performance benefits in a significant way from being able to keep intensively used words in registers all the time.

Few memory references The algorithms use no tables, and memory references are limited to message word access. In the assumption that a message block of 16 words cannot be kept in registers, less than 15% of all instructions are referring to memory for all compression functions except SHA-1. For SHA-1 this figure rises to 33%, due to the linear recursion on the array of message words. These figures do not take into account the possible additional memory references needed in case an explicit coding of the string-integer conversion is required, the impact of which will be discussed in Section 6.

Fairly compact code The code of all compression functions is fairly compact (see e.g., Table 5), and will never be larger than 8K. This means that it can be kept in the on-chip cache of most processors, leading to faster execution of the code from the second iteration onwards.

Endianness Before it can be processed the message block is converted from a 64-byte string to a 16-word block. However, two conventions are in use for this string-integer conversion: the byte with the lowest address in memory is either the first word's least significant byte (little-endian conversion) or its most significant byte (big-endian conversion). Loading data from memory into a register the processor uses one of these conversions, and the other has to be coded explicitly, causing a performance degradation. SHA-1 uses big-endian conversion, all other hash functions use little-endian conversion.

3 Pentium architecture overview

The Pentium processor is a member of the Intel x86 processor family offering several architectural improvements over its predecessors, and, at the same time, remaining fully compatible with them: code written for the i386/i486 processors will without any problem run on the Pentium, and, due to the new architectural features, faster than one would expect from the ratio of their clock frequencies. However, for maximum performance, the i386/i486 code will have to be rewritten, and in order to do so, a thorough knowledge of the Pentium's architecture and its new features is indispensable.

The Intel x86 processor family belongs to the CISC community of processors, known for their large instruction set, consisting of more than 300 machine instructions, their complex addressing schemes, and the micro-encoding of the processor instructions. The latter refers to the fact that a single processor instruction is encoded as a sequence of more elementary instructions to the instruction and execution unit in the processor. The reduction of the extensive and complex instruction set to fewer and more efficient instructions, with the ability to execute most of the instructions in one clock cycle, resulted in the RISC architecture. Other features of RISC processors are: a large number of hardware storage registers, instruction pipelining, and, more recently, a superscalar architecture, allowing parallel execution of more than one instruction in separate pipelines.

The Pentium is a typical CISC processor in that it inherited the large instruction set and the small register set of its predecessors. In this context it suffices to know that the Pentium has 7 general-purpose registers `eax`, `ebx`, `ecx`, `edx`, `esi`, `edi`, `ebp`, each 32 bits wide. But it also shares a number of characteristics with modern RISC designs, such as a pipelined approach to instruction execution, and a superscalar architecture. New Pentium features of interest to us are:

– a superscalar dual-integer execution unit;
– a split cache: two 8-Kbyte on-chip caches for data and code;
– an advanced branch prediction mechanism;
– a 64-bit external data bus interface; and
– an integrated performance-monitoring module.

Of these, the first two are the main tools for improving the performance of the hash functions in the MD4-family, and will be treated in more detail below. The third feature won't influence performance by much in our case, as our code contains only one branch (i.e., looping over 64-byte message blocks). It will nevertheless ensure that, except for the first and last iteration, the branch will be correctly predicted, and will execute in (at most) a single clock cycle. The fourth item will only concern us as far as it allows for faster cache line fills of the Pentium's two internal caches. The last, largely undocumented feature [Mat94] will allow us to monitor the extent of our improvements.

The Pentium processor allows to execute two instructions in parallel through two five-stage pipelines, called the U pipe and the V pipe. The processor always issues the first instruction of the pair to the U pipe. The second instruction of the pair is issued to the V pipe only if the instruction satisfies a number of constraints, called the *pairing rules*. An important feature of the Pentium is that this instruction pairing is carried out automatically and independently. Neither software control instructions nor specific dual instructions are required in order to use the superscalar architecture. Instruction pairing and parallel execution is completely transparent to the programmer. However, it is obvious that the sequence of the instructions plays a significant part in improving the performance.

Slight changes in the code sequence, e.g., to avoid dependency between consecutive instructions, can produce substantial improvements in performance. That is where the pairing rules come into play, an understanding of which is crucial for optimization purposes.

Pairing Rules

Rule 1 Both instructions in the pair must be `simple`. `Simple` instructions are entirely hardwired, and therefore require no microcode support. In this way, they can normally execute in a single clock cycle. These instructions include register-to-register and immediate-to-register ALU operations (any arithmetic or logical instruction, such as `add`, `and`, `or`, `xor`, `rol`); `movs`, `inc`, `dec`, `push`, `pop`, `lea`, and `nop`. The `near` conditional and unconditional branches `jcc`, `jmp`, and `call` can only be paired if they occur as the *second* instruction in a pair. In addition, all the ALU memory-to-register and register-to-memory instructions are considered `simple`, even though they require respectively two and three clock cycles. From the shift and rotate family only a shift/rotate by 1 position and a shift by an immediate value are pairable, and even then only as the *first* instruction in a pair.

Rule 2 There must be no data dependencies between the two instructions. A destination of the U pipe instruction cannot be used as a source or destination of the V pipe instruction.

Rule 3 Neither instruction in a pair may contain both a displacement and an immediate value.

Rule 4 Instructions with prefixes can only occur in the U pipe.

These pairing rules are summarized in Algorithm 1. For a detailed explanation of these pairing rules we refer to [Int93a].

The last rule is important for our purposes as it implies that 32-bit instructions can only be paired in native protected mode. In real, virtual 8086, and 80286-compatible mode the default size of operands and addresses is 16-bit, while in native protected mode the default is 32-bit. A prefix is used to change the default value, i.e., to execute a 16-bit instruction in native protected mode, or a 32-bit instruction in real, virtual 8086, or 80286-compatible mode. This is important, as under DOS or Windows still many (most?) applications run in 16-bit mode, and hence pairing of 32-bit instructions is not possible. Moreover, each prefix incurs a penalty of an additional clock cycle. The MD4-like hash functions consist nearly exclusively of 32-bit operations, nearly all of which can be implemented by means of 32-bit instructions executing in a single clock cycle. Running such an implementation in 16-bit mode will result in serious performance degradation: pairing is impossible, and nearly every instruction takes twice as long due to the additional prefix cycle. A 32-bit implementation of e.g., SHA-1 runs *three* times slower in real mode than in native protected mode. A 16-bit implementation probably won't do much better due to the increase of instructions to more than twice the amount of a 32-bit implementation.

The meaning of the instructions mentioned in Rule 1 is straightforward, except perhaps for `lea`, which performs memory addressing calculations and has

```
if (     ($I1$ = simple)
     and ($I1 \neq$ jump)
     and ($I1 \not\ni$ displacement-immediate)
     and ($I2$ = simple)
     and ($I2 \neq$ shift/rotate)
     and ($I2 \not\ni$ displacement-immediate)
     and ($I2 \not\ni$ prefix)
     and (destination of $I1 \neq$ source of $I2$)
     and (destination of $I1 \neq$ destination of $I2$)
) then {
    issue $I1$ to U pipe
    issue $I2$ to V pipe
} else
    issue $I1$ to U pipe
```

Algorithm 1.. Pentium's algorithm to determine whether the consecutive instructions $I1$ and $I2$ can be paired.

the interesting feature that it can be used as a super-`add` instruction (see e.g., [Abr94]). The intended use of `lea` is to calculate the offset of a particular memory location by adding a base address, an index value, and a fixed displacement and storing the result in a specified register. Base address, index, and result can be any general-purpose 32-bit register, and the displacement can be any 32-bit constant. This means that `lea` can be used to add any two general-purpose registers and any constant and store the result in any register, all in one instruction taking, in principle, no more time than adding just 2 of them by means of `add`. However, there is one important difference with `add`: the two general-purpose registers to be added by means of `lea` must have received their value at least one cycle in advance of the `lea` instruction. This is a consequence of the fact that the value of a register needed in memory addressing calculations has to be set far enough ahead for the Pentium to perform the addressing calculations before the instruction needs the address. Otherwise a so-called address generation interlock (AGI) is generated causing the pipeline to stall until the value becomes available and the addressing calculations have been performed.

Another source of pipeline stalls are references to (slow) memory. Their impact on performance can be reduced by the use of cache memory. The Pentium has split instruction and data caches (a Harvard architecture), in contrast to the i486, that had a unified cache for both code and data (a Princeton architecture). This eliminates interference between data and instruction references, and allows for simultaneous data and instruction fetches. Each is an 8-Kbyte, so-called two-way set-associative cache with 32-byte lines. In such a cache data is moved to and from cache in units of 32 bytes (a line), lines are grouped in sets of 2 lines each (two-way), and a block of data can be placed in either of the 2 lines within a set (set-associative). The advantage of a two-way set-associative cache over a direct-mapped cache of the same size (in which data can be placed in one location only) is that it decreases the miss rate by a factor of 2, and therefore increases system performance. Cache lines are filled or written back in burst mode, a special bus mode in which a complete 32-byte cache line is transferred in 4 bus cycles (1 for each 64-bit quad word, benefiting from the Pentium's 64-bit external data bus).

4 Pairing in MD4-like hash functions

The pairing rules that concern us most are rules 1 and 2. Rule 1 tells us which instructions can be paired, while rule 2 states under which conditions this can happen. A comparison of the `simple` instructions of rule 1 with the simple instructions used in the MD4-family of hash functions and listed in Section 2 learns us that both sets of 'simple' instructions overlap nearly completely, but for two exceptions: one's complement and rotation over more than 1 bit position.

The former exception is not such a problem. The `not` instruction only appears in the boolean functions $(x \wedge y) \vee (\overline{x} \wedge z)$ and $(x \vee \overline{z}) \oplus y$. An implementation will substitute the first expression by the equivalent, but more efficient expression $((y \oplus z) \wedge x) \oplus z$ [NMVR95, Appendix 3]. The second expression is already optimal from a performance point of view, and here $\overline{z}$ can be substituted by the pairable expression $z \oplus \mathtt{FFFFFFFF}_\mathtt{x}$. However, the fact that a rotation over more than 1 bit position cannot be paired with another instruction is very unfortunate. Table 1 indicates that the step function of each MD4-family member contains at least one such rotation, SHA-1 and RIPEMD-160 contain even two of them. Of course, a rotation over n bit positions could be replaced by n rotations over 1 bit. However, both instructions last 1 cycle, so that it only pays off for $n = 2$, and even then only if they can be paired with other instructions: a rotation over 1 bit can only be the first instruction of a pair. This strategy is only applicable in case of the SHA-1 instruction $C^{\lll 30}$, that could be replaced by two rotations over 1 bit position in the opposite direction.

Except for these rotations all instructions of the MD4-like compression functions can in principle be paired. However, rule 2 tells us that for it to be possible in practice there should be no data dependencies between two consecutive instructions. A straightforward implementation of a step will result in practically no pairing due to data dependency between each instruction and the next, as illustrated in Table 3. Here a step of MD5's first round is implemented in three ways: straightforward, optimized for maximal pairing, and optimized using `lea` as super-`add`. The problem with the first approach is that for 7 out of the 9 possible pairs the destination of both instructions in the pair is the same, for 1 pair the destination of the first instruction is the source of the second, and only the last instruction of a step can be paired with the first instruction of the next step. Or, without recoding, only 1 out of 9 instructions is executed in the V pipe. And this example is by no means specially selected, but it is the illustration of the typical situation for all straightforward implemented step expressions. By rearranging the instructions it turns out that *all* simple instructions can be paired, resulting in a gain of 50% (from 9 cycles for the straightforward implementation to 6 cycles for the optimized case). The V pipe use increased to 4 out of 9 instructions, or 44%. Once again, these figures are typical for every round of each MD4-family member: all their simple instructions can be paired, except for those rounds that use the multiplexer $(x \wedge z) \vee (y \wedge \overline{z})$. The latter is the case for the second round of MD5, and the fourth left line round of RIPEMD-128 and RIPEMD-160, as well as in the corresponding right line rounds of the latter two hash functions.

$$A := B + (A + ((B \wedge C) \vee (\overline{B} \wedge D)) + X_i + K)^{\lll s}$$

Instructions	Cycles	Instructions	Cycles	Instructions	Cycles
⋮		⋮		⋮	
`add ebx,ecx`	1	`add ebx,ecx`	1	`add ebx,ecx`	1
`mov edi,ecx`	paired	`xor edi,edx`	paired	`xor edi,edx`	paired
`xor edi,edx`	1	`add eax,X[esi]`	2	`add eax,X[esi]`	2
`and edi,ebx`	1	`and edi,ebx`	paired	`and edi,ebx`	paired
`xor edi,edx`	1	`add eax,K`	1	`xor edi,edx`	1
`add eax,edi`	1	`xor edi,edx`	paired	`lea eax,`	
`add eax,X[esi]`	2	`add eax,edi`	1	`[eax+edi+K]`	1+AGI
`add eax,K`	1	`mov edi,ebx`	paired	`mov edi,ebx`	paired
`rol eax,s`	1	`rol eax,s`	1	`rol eax,s`	1
`add eax,ebx`	1	`add eax,ebx`	1	`add eax,ebx`	1
`mov edi,ebx`	paired	`xor edi,ecx`	paired	`xor edi,ecx`	paired
⋮		⋮		⋮	

Cycles per instr.	1.00	Cycles per instr.	0.67	Cycles per instr.	0.88
V pipe use	11%	V pipe use	44%	V pipe use	38%
Paired simple instr.	25%	Paired simple instr.	100%	Paired simple instr.	86%

Table 3. Implementation of a round 1 step of MD5 on a Pentium processor. The chaining variable A, B, C, D is stored in registers `eax` through `edx`. The optimized expression $((C \oplus D) \wedge B) \oplus D$ for the multiplexer is used. The left column shows the straightforward implementation. In the middle column the instructions are rearranged in such a way that all pairable instructions are paired. The right column illustrates the use of the super-`add` instruction `lea`. Memory read access is limited to 2 cycles if the data resides in the on-chip cache.

The `lea` instruction can, with reference to Table 1, be used to add the constant K and two out of A, $f()$, and X_i (and in the case of SHA-1 also $B^{\lll 5}$). The case $A + f() + K$ is illustrated in the right column of Table 3. The fact that `lea eax,[eax+edi+K]` replaces `add eax,K` and `add eax,edi` means that the resulting code is potentially faster, provided both AGIs and data dependencies can be avoided. In order to avoid an AGI both A and $f()$ have to be ready 1 cycle in advance of `lea`. This 1-cycle gap (or 2 paired instructions) can only be filled with instructions of the next step, as instructions of the current step involve as a destination either the register for A or for $f()$. But bringing instructions forwards from the next step introduces data dependencies in that step. Therefore, in general the use of `lea` does not result in faster code, in case of the example even in slower one (7 cycles compared to 6 cycles using `add`s). There is, however, one exception: SHA-1. The rotations in SHA-1 are, in contrast to the other hash functions, confined to individual chaining variables ($B^{\lll 5}$, $C^{\lll 30}$). This allows for greater flexibility in moving instructions between steps, so that both AGIs and data dependencies can be avoided.

This discussion of the pairing rules allows us to formulate a number of general guidelines that help us pairing as many instructions in a step as possible:

1. Sometimes it pays off to substitute non-pairable instructions by 1 or more simple instructions with the same effect. Examples are one's complement and rotation over more than 1 bit position.
2. It might be necessary to move instructions from one step to the previous or next one. An example is the `mov edi,ebx` instruction in Table 3, which is the first instruction of the next step, and has been moved forwards by two instructions for pairing purposes. In this respect it is important to mention that Pentium instructions can never be executed out of order, and therefore it is up to the programmer to properly change the order of execution by rearranging the instructions.
3. Pairing of two instructions where the source of the first instruction is the destination of the second instruction, is no problem. An example is the pair `add eax,edi` and `mov edi,ebx` of Table 3, that is executed in a single clock cycle.
4. Sometimes it is useful to use two different auxiliary registers in two consecutive steps. This creates more data-independent instructions, and by moving them from one step to another, more room for pairing simple instructions. Of course, the fact that the Pentium has only 7 general purpose registers, up to 5 of which are used for storing the chaining variable, restricts the options available to us to a bare minimum. Putting some register contents temporarily on stack sometimes pays off.
5. Pairing a 1-cycle instruction with a 2 or 3-cycle instruction saves only 1 out of 3 or 4 cycles. However, most of the time it is the only possibility. Only the feedforward of the chaining variable offers us the opportunity to pair 2 or 3-cycle instructions with each other. But even here the gain is only partial: pairing two simple 3-cycle instructions (so-called read-modify-write instructions) results in a 2-cycle penalty, since the write accesses of both instructions must be completed one after the other (the read-modify part is executed in parallel).
6. The use of `lea` as a super-`add` instruction only pays off if AGIs and/or additional data dependencies can be avoided.

5 Performance figures

The entire MD4-family has been implemented on the Pentium in Assembly according to the guidelines of the previous section. Analysis of the code, as well as the use of the built-in monitoring capabilities of the Pentium resulted in the figures of Tables 4 and 5. These figures refer to performance of the hash function's basic building block: the compression function. The figures for hashing a message of any length can be easily derived from these figures by taking into account the additional iteration due to the padding block. All cycle and speed related data are in the assumption that both code and data reside in the Pentium's on-chip caches. For the code and local data this is true after the first iteration. In the next section we will argue that also for the message block being hashed this is a realistic assumption in many applications. Moreover, the overhead for reading from secondary cache or main memory is relatively small, and does not depend on the number of memory references, but on the message block size (determining the number of data cache line fills; 2 for all hash functions concerned) and on the time between the first two references to data located in the same cache line (different for each hash function). Of course, it is also assumed that the data is aligned on a 4-byte boundary. Every misaligned access in the data cache costs an extra 3 cycles.

The implementations pair nearly all available `simple` instructions, except for a few instructions in the already mentioned multiplexer $(x \wedge z) \vee (y \wedge \overline{z})$ of MD5, RIPEMD-128, and RIPEMD-160. This results in a high percentage usage of the V pipe. In case of SHA-1 the higher percentage of pairable instructions in the linear recursion is compensated for by the lower percentage in the rest of the code: 2 non-pairable rotates per step, compared to 1 rotate for all other hash functions except RIPEMD-160. In addition to a higher percentage of non-pairable rotates, RIPEMD-160 suffers from the above mentioned multiplexer. An important criterion for judging the quality and speed of an implementation is the number of cycles per instruction (CPI), i.e., the number of cycles it takes, on average, for an instruction to execute. The minimal CPI is 1 for a non-paired instruction and 0.5 for a paired one. For all hash functions the average CPI is about 0.70. Without dual-integer execution this would be about 1.13, or a gain of a factor 1.6. The reason that a dual-pipeline superscalar architecture does not result in a speed-up of a factor 2 is twofold: not all instructions can be executed in parallel (e.g., rotates), and some 1-cycle instructions are paired with 2 or 3-cycle instructions. To gauge the impact of the latter on the overall performance, we calculated the theoretical minimum CPI for our code of each hash function, i.e., the CPI in case each pairable instructions of our implementation was paired, and all 2 and 3-cycle instructions were paired with each other. This is no minimum in absolute sense, but only with respect to our code (another implementation could have a lower minimum). It turns out that the actual CPIs of Table 4 are within 90% of this theoretical minimum for all hash functions concerned. This figure also relates to the number of memory references, as it are precisely those references that take 2 or 3 cycles (in our case mostly 2). SHA-1's linear recursion is in this respect both a curse and a blessing: it involves mainly memory references, but they can be paired with each other. That is why SHA-1, despite its 36% of instructions that refer to memory, can keep up with the rest.

Algorithm	MD4	MD5	SHA-1	RMD	RMD-128	RMD-160
Instructions	397	573	1247	795	985	1566
% V pipe use	43.32	41.19	42.82	43.65	41.73	37.55
% Paired `simple` instr.	98.57	92.73	99.72	99.57	95.92	94.38
% Memory ref.'s	14.61	12.91	35.85	14.72	15.13	11.88
Cycles	275	403	943	556	718	1153
Cycles per instr.	0.69	0.70	0.76	0.70	0.73	0.74
Speed-up factor	1.63	1.59	1.64	1.62	1.57	1.51

Table 4. Performance figures on a Pentium for our implementation of the compression function of the 6 members of the MD4 hash function family. Both code and data are assumed to reside in the on-chip caches. All figures are independent of the processor's clock speed. The speed-up factor is with respect to a (hypothetical) execution of the same code on a non-parallel architecture under otherwise unchanged conditions.

The bandwidth figures of Table 5, obtained from actual timings, correspond exactly with the cycle figures of Table 4, if one allows for a few cycles overhead. A portable C implementation is, on average, twice as slow. The first iteration of a compression function takes longer because of code and data cache fills, and has been excluded from the timings. Compared to the MD5 figures in [Tou95], our C version is 28% faster, and our Assembly implementation is faster by almost a factor 2.5.

Algorithm	Size (bytes)	Speed (Mbit/s) Portable C	x86 Assembly	Factor Assembly-C
MD4	1092	81.5	166.8	2.04
MD5	1611	59.7	113.7	1.90
SHA-1	5157	21.2	48.7	2.30
RIPEMD	2122	44.0	82.7	1.88
RIPEMD-128	2716	35.6	64.0	1.80
RIPEMD-160	4280	19.3	39.9	2.07

Table 5. Code size and hashing speeds of the different compression functions on a 90 MHz Pentium both for our Assembly implementations and a corresponding portable C implementation (Watcom C 10.0). The code size only refers to the Assembly implementations. Again both code and data are assumed to reside in the on-chip caches. The figures are independent of the buffer size as long as it, together with the local data, fits in the 8-Kbyte on-chip cache.

6 Effects of data cacheing and representation

The data to be hashed will in many applications reside in the on-chip cache. An example is file hashing, where multiples of the block size are read in a buffer, and hashed buffer by buffer. If the buffer size is smaller than the on-chip cache (8 Kbyte in our case), the reading from disk will already put the data in cache. Another example is when a piece of data is subjected to more operations than just hashing, such as compression or encryption. Only one of these operations will have to bear the overhead of reading the data from main memory.

But even if the data is read from the much larger (e.g., 256 Kbyte) secondary cache or from main memory, the overhead is limited, due to the architecture of the on-chip (L1) cache (see Section 3). Data is read into the L1 cache 32 bytes at a time, so that a memory reference to a message word causes 7 other message words to be read as well. This is a perfect example of the spatial locality principle: chances are high that in the near future a word will be accessed close to the one just accessed. Hence, only two 32-byte cache line fills are needed to read a 64-byte message block into the on-chip cache. One such read causes the pipeline to stall for 7 cycles if read from secondary (L2) cache, and 20 cycles if read from main memory. A second reference to data residing in the same cache line will not result in additional delays, provided the entire cache line has been filled at the moment of the second reference (program execution and cache line filling are partially running in parallel). Unfortunately, this is not the case for our code, but one can make sure that the first two references are as far apart as possible. The resulting additional delays are for all implementations on average 5 cycles for an L2 access, and 10 cycles for an access to main memory. For SHA-1 these figures are slightly higher due to the explicit big-endian conversion, in which memory references are closely packed together: 8 cycles (L2) and 15 cycles (main memory), respectively. The cycle figures related to cacheing effects apply strictly speaking only to the particular configuration used for these measurements (90 MHz Pentium), and might be different for configurations using a Pentium with a different clock speed or different types of L2 and main memory. In particular it is expected that for faster Pentiums the access times to memory will increase, as it is a well known fact that memory performance has a hard time keeping up with that of processors.

A final issue is the conversion from little-endian to big-endian representation. This conversion can be efficiently implemented using the `bswap` instruction. This instruction is listed to take 1 cycle [Int93b], but always has the `0F`$_{\mathtt{x}}$-prefix. Each prefix requires 1 additional cycle, so that `bswap` takes in effect always 2 cycles [Gul95]. Using `bswap` the penalty incurred by endianness conversion is limited to 48 cycles, including data copying. On a Pentium only SHA-1 is affected by this conversion, but Table 6 lists the effect on all hash functions. The figure of 11% for MD5 should be compared to the 33% figure reported in [Tou95]. Cacheing effects hardly influence the time spent on endianness conversion: the local data buffer storing the converted data will, after the first iteration, reside in the on-chip cache, and the extra time needed for reading from secondary cache or main memory will be almost the same, whether an explicit conversion is required or not. Only the location of these time consuming memory references in the code will be different: during round 1 of the compression function in case no conversion is required, and during the conversion itself otherwise. However, as such the extra reading time is no part of the conversion.

Algorithm	MD4	MD5	SHA-1	RMD	RMD-128	RMD-160
Data from L2 cache	8.7	5.5	3.3	4.3	3.7	2.1
Data from main memory	21.8	14.4	7.5	10.8	8.7	5.3
Endianness conversion	14.9	10.6	5.1	7.9	6.3	4.0

Table 6. Percentage of performance degradation when data is read from secondary cache or from main memory, as well as the percentage of time spent on (hypothetical) endianness conversion. Only SHA-1 actually executes the latter.

7 Conclusion

Efficient and optimal implementations of all MD4-like hash functions on a Pentium processor have been presented. The increase in performance with respect to an equally fast non-parallel architecture is in the order of 60%. It has also been shown that the impact on performance from processing non-cached data as well as from endianness conversion is relatively small. In addition, a number of implementation guidelines have been derived, that are also applicable to implementations of other cryptographic primitives.

References

[Abr94] M. Abrash, *Zen of code optimization,* Coriolis Group Books, 1994.

[Dob96a] H. Dobbertin, “Cryptanalysis of MD4,” *Fast Software Encryption, LNCS 1039*, D. Gollmann, Ed., Springer-Verlag, 1996, pp. 53–69.

[Dob96b] H. Dobbertin, “Cryptanalysis of MD5 compress,” presented at the rump session of Eurocrypt’96.

[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. A corrected version and a C reference implementation can be found in the directory `/pub/COSIC/bosselae/ripemd/` at ftp site `ftp.esat.kuleuven.ac.be`.

[FIPS180-1] FIPS 180-1, *Secure hash standard,* NIST, US Department of Commerce, Washington D.C., April 1995.

[Gul95] P. Gulutzan, “Making programs go faster,” *Dr. Dobb’s Journal*, Vol. 20, No. 1, January 1995, pp. 133–135.

[Int93a] *Pentium processor user’s manual, Volume 1: Pentium processor’s data book,* Intel Corp., Mt. Prospect, Ill., 1993.

[Int93b] *Pentium processor user’s manual, Volume 3: Architecture and programming manual,* Intel Corp., Mt. Prospect, Ill., 1993.

[Mat94] T. Mathisen, "Pentium Secrets," *Byte*, Vol. 19, No. 7, July 1994, pp. 191–192.

[NMVR95] D. Naccache, D. M'Raïhi, S. Vaudenay, D. Raphaeli, "Can DSA be improved? Complexity trade-offs with the Digital Signature Standard," *Advances in Cryptology, Proceedings Eurocrypt'94, LNCS 950*, A. De Santis, Ed., Springer-Verlag, 1995, pp. 77–85.

[Pre93] B. Preneel, *Analysis and design of cryptographic hash functions,* Ph.D. thesis, K.U.Leuven, February 1993.

[Pre94] B. Preneel, "Cryptographic hash functions," *European Transactions on Telecommunications*, Vol. 5, No. 4, 1994, pp. 431–448.

[Riv92a] R.L. Rivest, "The MD4 message-digest algorithm," *Request for Comments (RFC) 1320*, Internet Activities Board, Internet Privacy Task Force, April 1992.

[Riv92b] R.L. Rivest, "The MD5 message-digest algorithm," *Request for Comments (RFC) 1321*, Internet Activities Board, Internet Privacy Task Force, April 1992.

[Rob95] M. Robshaw, "MD2, MD4, MD5, SHA, and other hash functions," Technical Report TR-101, Version 4.0, RSA Laboratories, July 1995.

[RoCo94] P. Rogaway and D. Coppersmith, "A software-optimized encryption algorithm," *Fast Software Encryption, LNCS 809*, R. Anderson, Ed., Springer-Verlag, 1994, pp. 56–63.

[RIPE95] RIPE, *Integrity Primitives for Secure Information Systems. Final Report of RACE Integrity Primitives Evaluation (RIPE-RACE 1040), LNCS 1007*, A. Bosselaers and B. Preneel, Eds., Springer-Verlag, 1995.

[Tou95] J. Touch, "Performance analysis of MD5," *Proceedings of ACM SIGCOMM'95, Comp. Comm. Review,* Vol. 25, No. 4, 1995, pp. 77-86.