AN ABSTRACT OF THE THESIS OF

Onur Acıiçmez for the degree of Master of Science in Electrical and Computer Engineering presented on May 11, 2004.

Title: Fast Hashing on Pentium SIMD Architecture

Abstract approved: Signature redacted for privacy.

Çetin Kaya Koç

The SIMD (single-instruction, multiple-data) architecture is implemented in many popular general-purpose processor families, including Intel Pentium. In this paper, we examine if any performance gains can be obtained in the implementation of the Secure Hash Algorithms (SHA-1, SHA-256, SHA-384, and SHA-512) and present the details of our optimization methods and implementation details. It turns out that while the hashing of a single message is slower, the hashing of 2 or 4 independent message streams can be made signicantly faster.

Fast Hashing on Pentium SIMD Architecture

by

Onur Acıiçmez

A Thesis

submitted to

Oregon State University

in partial fulfillment of
the requirements for the
degree of

Master of Science

Completed May 11, 2004
Commencement June 2005

TABLE OF CONTENTS

TABLE OF CONTENTS (Continued)

LIST OF FIGURES

Fast Hashing on Pentium SIMD Architecture

1. INTRODUCTION

High-speed software implementations of cryptographic hash functions are needed for many network security protocols. The performance is always an issue due to bandwidth requirements [13, 6, 7, 5], and modern hash functions are in fact designed with performance in mind, in addition to standard security requirements such as preimage resistance, second preimage resistance, and collision resistance. While hardware implementations will be expectedly faster [8], software implementations are also desired because flexibility and cost reasons [20, 21].

In this paper, we focus on how a single-instruction multiple-data (SIMD) parallel computation model can improve the software performance of the hash functions. The SIMD model speeds up the software performance by allowing the same operation to be carried out on multiple data elements in parallel. Most of the current general-purpose computers employ SIMD architectures. AltiVec extension to PowerPC [16], Intel's MMX technology SSE and SSE2 extensions [3], Sun's VIS [15] and 3DNow! of AMD [1] are examples of currently available SIMD architectures.

We are interested in obtaining high-speed and high-throughput implementations of the Secure Hash Algorithms [22, 23]. We analyzed the possibility of improving the speed or throughput of the SHA using the SIMD architecture and parallelization techniques. We chose Intel Architecture [12, 11] as the base SIMD platform since it is arguably the most widely used architecture among the ones cited above.

In chapter 2, we give an overview of single-instruction-multiple-data execution model. We briefly describe the SIMD architecture in Pentium 4 processors. Then we mention some implementation issues of SIMD in Pentium 4 under the titles code conversion, coding techniques, and coding methodologies.

In chapter 3, we characterize the Secure Hash Algorithms by indicating their properties and highlighting their importance in cryptography. This chapter also gives detailed description of SHA and analyze the SIMD compatibility of these algorithms.

We present our implementation details in chapter 4, with the results of our experiments. We express the optimization techniques employed to fasten our implementation. At the end of this chapter, we give the figures of performance results and relative performance gains obtained using SIMD instructions.

2. SINGLE INSTRUCTION MULTIPLE DATA (SIMD)

2.1. Overview

Single-instruction multiple-data execution model allows several data elements to be processed at the same time. The conventional scalar execution model, which is also known as single-instruction single-data (SISD), deals only with one pair of data at a time. Execution of an SIMD operation is illustrated in Figure 2.1.

The programs using SIMD instructions can run much faster than their scalar counterparts. However SIMD enabled programs are harder to design and implement. In order to perform parallel SIMD operations, the program must do:

1 . Load multiple data values into SIMD registers.

2 . Perform the SIMD operation on these registers.

3 . If required, load the results to memory.

4 . If more data has to be processed, repeat the steps.

SIMD instructions have the potential to speed-up the software, however there are mainly 2 problems with SIMD model:

1 . If the data layout does not match the SIMD requirements, SIMD instructions may not be used or data rearrangement code is necessary

2 . In case of unaligned data the performance will suffer dramatically.

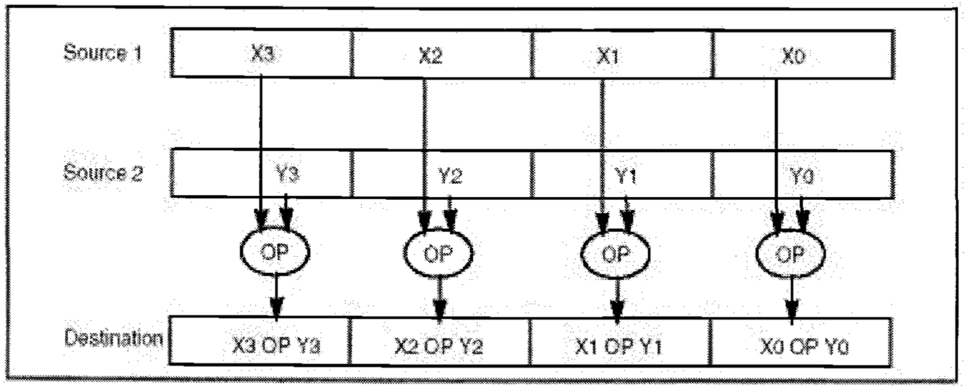

FIGURE 2.1. SIMD execution model

2.2. Intel's SIMD Architecture

Intel has introduced three extensions into IA-32 architecture to allow IA-32 processors to perform SIMD operations since the production of Pentium II and Pentium with Intel MMX technology processor families. These extensions are MMX technology, SSE extensions, and SSE2 extensions. They provide a group of SIMD instructions that operate on packed integer and/or packed floating point data elements contained in the 64-bit MMX or the 128-bit XMM registers.

Intel introduced MMX Technology in Pentium II and Pentium with MMX Technology processor families. MMX instructions use 64-bit MMX registers and perform SIMD operations on packet byte, word, or doubleword integers located in those registers.

The SSE SIMD integer instructions are the extension of MMX technology. They were introduced in Pentium III processors. These instructions use 128-bit XMM registers in addition to MMX registers and they operate on packed single-precision floating point values contained in the XMM registers and on packed integers contained in the MMX registers.

The latest SIMD extensions of Intel, SSE2, were introduced in the Pentium 4 and Intel Xeon processors. These instructions use both MMX and XMM registers and perform operations on packed double-precision floating-point values and on packed integers. The SSE2 SIMD integer instructions extend IA-32 SIMD operations by adding new 128-bit SIMD integer operations and extending all the 64 bit-SIMD integer operations introduced in the MMX technology and SSE to operate on data contained in the 128-bit XMM registers

The MMX Technology, SSE extensions, and SSE2 extensions provide a rich set of SIMD operations that operates on both integer and floating-point data arrays and on streaming integers and floating point data. These operations can greatly increase the performance of applications running on the IA-32 processors.

In this paper, we are interested in SIMD operations that can be performed on integers. As most of the other cryptographic algorithms, SHA uses integer data and performs operations on integers.

2.3. Implementation Issues of SIMD in Pentium 4

It is better to start by using one of the SIMD data types and adjusting a loop count or changing the application to execute fewer instructions to convert an application to use SIMD instructions. Following subsections give a guideline for converting a conventional code to SIMD-based program and also discuss some coding techniques and methodologies.

2.3.1. Code Convertion to SIMD Programming

Figure 2.2 provides a flowchart for the process of converting a code to SIMD programming. We have to identify the code that benefits by using SIMD technologies. Generally, a good candidate is a code that contains small-sized repetitive loops operate on sequential arrays. Being repetitive of these loops incurs costly application processing time. These codes have potential for increased performance when they are converted to use one of the SIMD technologies.

2.3.2. Coding Techniques

The process of transforming sequentially-executing, or scalar, program into a program that can execute in parallel is called vectorization. A code has to be vectorized to take advantage of the SIMD architecture. However, the memory accesses may have dependencies which can prevent parallel execution, thus vectorization. To apply vectorization to a loop, we have to determine if such dependencies exist. If there is no dependency that would limit parallel execution, we can re-code the loop with SIMD instruction and reduce the iteration count by the length of the SIMD operation. To use SIMD instructions efficiently, we have to handle data alignment, calculation compatibility, and data simplifying buffer lengths.

2.3.2.1. Data Alignment

The arguments of SIMD instructions must be aligned by 8 bytes when using MMX technology and 16 bytes when using SSE or SSE2 for maximum performance. An exception will occur when unaligned memory is used with the SSE or SSE2 instructions. However, there are special-purpose , slower, unaligned memory move instructions part of SSE2.

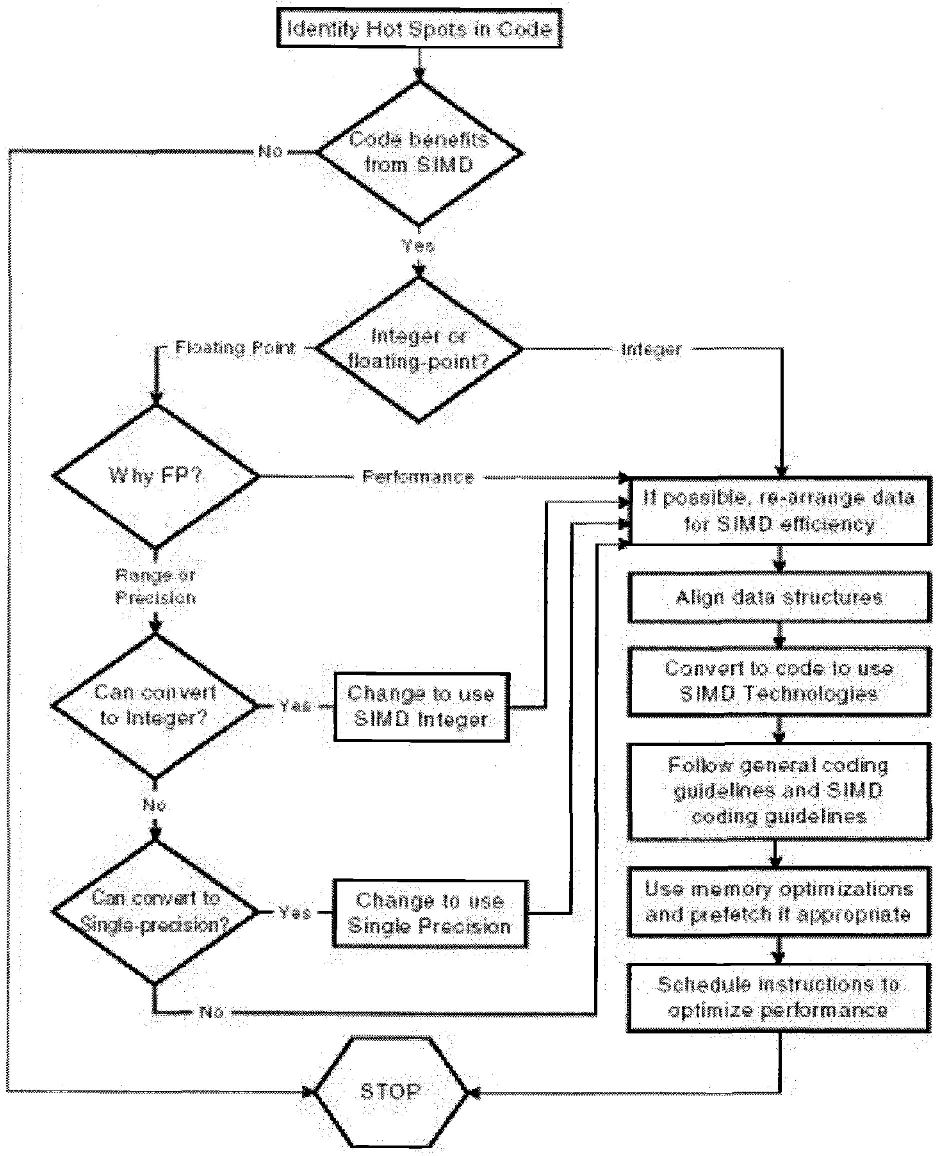
Identify Hot Spots in Code
Code benefits from SIMD
No
Yes
Integer or floating-point?
Floating Point
Integer
Why FP?
Performance
Range or Precision
If possible, re-arrange data for SIMD efficiency
Align data structures
Can convert to Integer?
Yes
Change to use SIMD Integer
Convert to code to use SIMD Technologies
No
Follow general coding guidelines and SIMD coding guidelines
Can convert to Single-precision?
Yes
Change to use Single Precision
Use memory optimizations and prefetch if appropriate
Schedule instructions to optimize performance
No
STOP

FIGURE 2.2. Guideline chart of SIMD conversion process

2.3.2.2. Compatibility of SIMD and x87 Floating Point Unit (FPU)

MMX technology and x87 FPU share their registers, and therefore, they cannot both be used at the same time. Most of the time, using 8-byte MMX technology registers can be avoided by using 16-byte SSE/SSE2 registers. 16-byte registers are not shared with x87 FPU, and also they can handle twice as much data at a time.

MMX, the floating-point SIMD instructions and the x87 FPU operate on the same registers. Therefore, extra attention has to be paid when using SIMD and x87 FPU instructions simultaneously. However, it is not an issue for this research, since we need to use only integer operations to implement SHA.

2.3.2.3. Data Simplifying Buffer Lengths

It is beneficial to use buffers that are multiples of the SIMD data size being used. An easy way to satisfy this requirement is padding or adding extra, unused bytes at the end of buffers Using multiples of the SIMD data size avoids the special case of dealing with an odd amount of remaining variables.

2.3.2.4. Integer SIMD in Pentium 4

MMX, SSE, and SSE2 technologies together offer 8- and 16-byte integer SIMD instructions that operate on 1-, 2-, 4-, 8-, and 16-byte integers.

Integer SIMD architecture in Pentium 4 can be used to do many operations including addition, subtraction, shifts, rotates, bitwise logical operations, and comparisons.

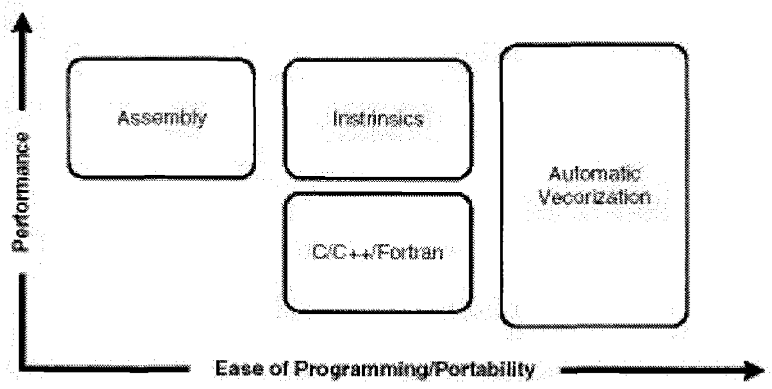

FIGURE 2.3. Coding methodologies

2.3.3. Coding Methodologies

Programming directly in assembly language for a target platform may produce a great performance gain, however, assembly code is not portable and costly to write and maintain. Performance objectives can also be met by taking advantage of different SIMD technologies using high-level languages instead of assembly. The new C/C++ language extensions designed specifically for Intel's SIMD technologies make this possible. Figure 2.3 illustrates the trade-offs between different methodologies. The following subsections discuss each methodology.

2.3.3.1. Assembly

Key parts of a code can be implemented directly in assembly language using an assembler or by using inlined assembly in C/C++ code. This model offers the greatest performance, but the lack of portability is the main disadvantage.

2.3.3.2. Intrinsics

They provide the access to the ISA functionality using C/C++ style coding. Intel defined three sets of intrinsic functions implemented in Intel C++ compiler to support MMX, SSE, and SSE2. Four new C data types, representing 64- and 128-bit objects, are used as the operands for these functions. These intrinsics are portable among all Intel processors. The performance that can be obtained using intrinsics is close to the levels achievable with assembly.

2.3.3.3. C++ Classes

Some compilers, i.e. Intel C++ compiler, provide a set of C++ classes to support MMX, SSE, and SSE2. They offer both a higher-level abstraction and more flexibility for programming and allow developers to write more natural C++ code without worrying about which intrinsic or assembly language instruction to use for a given operation.

2.3.3.4. Automatic Vectorization

Most of the current compilers provide optimization mechanisms which vectorize appropriate simple loops automatically. Only certain types of loops can be automatically vectorized, and in most cases user interaction with the compiler is need to fully enable this option.

3. SECURE HASH ALGORITHM (SHA)

3.1. Overview of Hash Functions

There are two tools needed almost for any cryptographic system and protocol: one-way functions and random number generators. This chapter covers a detailed description of a particular kind of one-way functions, namely hash functions. The use of hash functions in cryptology is vital. They are used especially for authentication, integrity, and non-repudiation.

The most famous hash functions are the MD-family: MD4, MD5, and SHA; but there are a large number of other hash functions. In this work, we concentrate on SHA because of its greater importance due to being an American Standard.

Cryptographic mechanisms need functions that can be computed one way but not the inverse: one-way functions. Cryptography needs these functions because of the main property of being hard to compute the inverse of the function. The theory of these functions involves mathematical issues which are not fully solved or known. Most of these functions, which are currently employed in cryptography, are based on "hard problems" such as factorization and discrete logarithm. As long as these problems stay "hard", meaning that they cannot reduce to simple problems, and as long as quantum computers stay being a "theorytical issue", these one-way functions will be a key element in secure systems.

3.1.1. One-Way Functions

A function f is a one-way function if it is easy to compute $f(x) = y$ for any given x in the domain of f, but it is computationally infeasible to compute $f^{-1}(y) = x$ for a given y. The theory behind the one-way functions is based on hard-problems like factorization and discrete logarithm. An example of a one-way function is the calculation of square roots in modulo a composite.

$$f(x) = x^2 \mod \text{n} \tag{3.1}$$

where n = pq, p and q are unknown large primes. It is very easy to compute f(x), however it is very hard to find x when the value of (x^2 mod n) is given.

3.1.2. Hash Functions

A hash function is a one-way function, which is usually shown as $h(x) = y$. The input x is a variable length string, while the output y is almost always of a fixed size. The result of the hash function is called message digest.

Hash functions tend to reduce the input we give. Therefore, there is always a possibility to find two input values with the same output. A function has to have the following properties to be used as a hash function.

- Preimage resistance: it must be computationally infeasable to find x using the resulting digest y, which implies that a hash function must be a one-way function.

- Weak collision resistance: given two values x and its image y, then it must be computationally infeasable to find an $x' \neq x$ where h(x) = h(x').

- Collision resistance: it must be computationally infeasable to find any x and x' with $x \neq x'$ where h(x) = h(x').

There are two main types of hash functions: keyed and unkeyed. Keyed hash functions use a second message, called key, as an input with the original input message to calculate the message digest. Only the message to be hashed is the input to unkeyed hash functions. Example applications of the keyed and unkeyed hash functions are the message authentication codes (MACs) and the modification detection codes (MDC), respectively. Both types of hash functions are widely used in cryptography. SHA is a member of MDC set, which is a subset of keyed hash functions.

3.1.3. Applications of Hash Functions

Most typical use of hash functions is digital signatures. Given a message, it is possible to sign itself. However it is prefered to sign the digest of it instead of the whole message, because of the heavy computations public-key systems require. Current digital signature schemes use hash functions to obtain the digest of the message. The other uses of hash functions are as follows:

- Message Authentication: Although digital signatures can be used for message authentication , they are usually slow and have key distribution problem. When two parties need to communicate, it is better to use a symmetric key system and send not only the message but also the message digest with it. For message authentication purposes, keyed hash functions should be used.

- Password Validation: In the earlier schemes, the login name was encrypted using the password as the key. The login name and cipher pairs were kept together by the system and the password validation was performed by either encrypting the login name or decrypting the cipher. In the latter schemes, a hash function is used instead of an encryption algorithm. For example, in Unix systems only the hash of the password is stored with the user name and when user logs in, the entered password is hashed and compared to the stored one

- Software Protection: Hash functions can be used to protect software against modification and viruses. A system or a user can compute a digest of the software and store it in a safe place. Any modification can be realized by recomputing the digest and comparing to the stored one.

- Computing Keys: Hash functions can be used to compute fixed size keys from longer text like passphrases. Longer passphrases have much more entropy than passwords, but they must be compressed to a fixed size. Hash functions can also be used to produce a number of different keys from a master key.

- Generating Pseudorandom Numbers: Pseudorandom numbers can be generated using a hash function and a proper seed.

3.1.4. Modification Detection Codes (MDC)

The hash functions in this category can be further classified according to the properties they have:

- One way hash function: These functions have preimage resistance and weak collision resistance properties. Every one-way function cannot be used as a one-way hash function, because the latter property brings supplementary restrictions.

- Collision resistant hash function: These functions have also the third property, collision resistance.

Another classification of these functions depending on their structures can be following:

- Based on block cyphers: These hash functions use an existing block cypher to obtain message digest.
- Customized hash functions: These functions are created specifically to be used in hashing purposes.
- Based on modular arithmetic: They use existing modular arithmetic and structures to reduce the message.

3.1.5. MD Family Hash Functions

These are most widely used, therefore most important, hash function in MDCs. The computation of hash functions in this family is done very similar to each other:

- Some extra bits are padded to the message depending on its size.
- The padded message divided into fixed sized blocks
- A set of constants is applied to add dispersion to the message.
- A set of rounds are iteratively applied to the blocks.
- The result of the round is applied as input for the next round

Some of the functions in this family are MD4, MD5, and SHA. MD4 has 3 rounds while MD5 has 4 rounds of computation. They create a message digest of 128 bits. MD4 has already been broken, and therefore should not be used. And MD5 should not be considered for future applications. Both of these hash functions does not meet recent cryptographic requirements. The new generation in this family of hash functions is SHA.

3.2. Description of SHA Algorithms

The SHA is an iterative hash function that processes a message to produce a message digest. There are four different versions of SHA, namely SHA-1, SHA-256, SHA-384, and SHA-512 that differ in the number of bits of the message digest values. They further differ in terms of word and block sizes. SHA-1 and SHA-256 use message blocks of 512 bits represented as a sequence of sixteen 32-bit words. These two hash computation algorithms perform operations on 32-bit words. On the other hand, the operations are performed on 64-bit words in SHA-384 and SHA-512. These algorithms use 1024-bit message blocks, which are represented as a sequence of sixteen 64-bit words.

Each algorithm has two stages: preprocessing and hash computation. Preprocessing involves padding a message, setting initialization values, and parsing the padded message into m-bit blocks where m is 512 for SHA-1 and SHA-256, and 1024 for SHA-384 and SHA-512. A message schedule is generated from the padded message and the hash computation uses this schedule to iteratively generate a number of intermediate hash values. The final hash value becomes the message digest.

Each of four SHA algorithms uses a different compression function as its basic building block, as shown in Figure 3.1. SHA-384 is mainly the same algorithm as SHA-512. There are only two differences: the number of bits in the output message digest and the initial hash value. SHA-256 has 64 rounds of com-

Algorithm	Compression Function
$SHA-1$	from round 17 onwards: $W_t = (W_{t-3} \oplus W_{t-8} \oplus W_{t-14} \oplus W_{t-16})^{\Leftarrow 1}$
	$T = A^{\Leftarrow 5} + f(B,C,D) + E + K_t + W_t$
	$E = D; D = C; C = B^{\Leftarrow 30}; B = A; A = T;$
$SHA-256$	from round 17 onwards: $W_t = \sigma_1(W_{t-2}) + W_{t-7} + \sigma_0(W_{t-15}) + W_{t-16}$
	$T_1 = H + \sum_1(E) + Ch(E,F,G) + W_t + K_t$
	$T_2 = \sum_0(A) + Maj(A,B,C)$
	$H = G; G = F; F = E; E = D + T_1; D = C; C = B; B = A; A = T_1 + T_2$
$SHA-512$	from round 17 onwards: $W_t = \sigma_1(W_{t-2}) + W_{t-7} + \sigma_0(W_{t-15}) + W_{t-16}$
	$T_1 = H + \sum_1(E) + Ch(E,F,G) + W_t + K_t$
	$T_2 = \sum_0(A) + Maj(A,B,C)$
	$H = G; G = F; F = E; E = D + T_1; D = C; C = B; B = A; A = T_1 + T_2$

FIGURE 3.1. Definition of the compression function used in SHA algorithms. Additions are modulo 2^{32} in SHA-1 and SHA-256, and modulo 2^{64} in SHA-512. K is a constant, W_t is the message schedule, and $A, B, \ldots, H$ are the chaining variables. $x^{\Leftarrow s}$ indicates the rotation of x to the left by s bits. $f()$ is one of the boolean functions given in Figure 3.2.

putation while the others have 80 rounds. At each round the compression function is applied on a certain number of chaining variables.

Figure 3.2 shows the functions employed in the compression rounds. The boolean functions are given in [23], but they are not efficient enough to use in our implementation. We will describe more efficient versions of these functions with the implementation details later in this paper.

SHA-256 Functions	
$\sum_0(x)$	$x^{\Longrightarrow 2} \oplus x^{\Longrightarrow 13} \oplus x^{\Longrightarrow 22}$
$\sum_1(x)$	$x^{\Longrightarrow 6} \oplus x^{\Longrightarrow 11} \oplus x^{\Longrightarrow 25}$
$\sigma_0(x)$	$x^{\Longrightarrow 7} \oplus x^{\Longrightarrow 18} \oplus x^{\rightarrow 3}$
$\sigma_1(x)$	$x^{\Longrightarrow 17} \oplus x^{\Longrightarrow 19} \oplus x^{\rightarrow 10}$

SHA-512 Functions	
$\sum_0(x)$	$x^{\Longrightarrow 28} \oplus x^{\Longrightarrow 34} \oplus x^{\Longrightarrow 39}$
$\sum_1(x)$	$x^{\Longrightarrow 14} \oplus x^{\Longrightarrow 18} \oplus x^{\Longrightarrow 41}$
$\sigma_0(x)$	$x^{\Longrightarrow 1} \oplus x^{\Longrightarrow 8} \oplus x^{\rightarrow 7}$
$\sigma_1(x)$	$x^{\Longrightarrow 19} \oplus x^{\Longrightarrow 61} \oplus x^{\rightarrow 6}$

Boolean Functions	
$Ch(x, y, z)$	$(x \wedge y) \oplus (\neg x \wedge z)$
$Parity(x, y, z)$	$x \oplus y \oplus z$
$Maj(x, y, z)$	$(x \wedge y) \oplus (x \wedge z) \oplus (y \wedge z)$

FIGURE 3.2. Functions used in SHA. $x^{\Longrightarrow s}$ represents a rotation to the right, $x^{\rightarrow s}$ represents a right shift both by s bits.

The implementation of the compression functions requires a limited number of simple operations:

- addition modulo 2^{32} or 2^{64}
- logical operations (and, or, xor, not) of 32-bit or 64-bit quantities
- shifts and rotations of 32-bit or 64-bit quantities
- load/store operations

In this paper we will mainly focus on these operations, because the performance of the SHA algorithm depends on them. The time spent on the compression functions is 88 % of the SHA computation time [13]. Therefore we will focus on the computation of compression functions and we will omit the details about the preprocessing phase.

3.3. SIMD Compatibility of SHA Algorithms

Our goal in this paper is to present a fast implementation of SHA algorithms on Intel's Pentium 4 processor and analyze the effects of Intel's SIMD instructions on the performance of the SHA. In this section, we will investigate the possible applications of the SIMD techniques to the algorithms and point out the appropriate parts of each algorithm, where SIMD instructions can be used.

The SIMD model speeds up the software performance by allowing the same operation to be carried out on multiple data elements in parallel. Therefore the programs using SIMD instructions can run much faster than their scalar counterparts. Intel has introduced three extensions into IA-32 architecture to allow IA-32 processors to perform SIMD operations. These are MMX technology, SSE, and

SSE2 extensions. The latest SIMD extension of Intel, SSE2, was introduced in the Pentium 4 and Intel Xeon processors. They provide a group of SIMD instructions that operate on packed integer and/or floating point data elements contained in the 64-bit MMX or the 128-bit XMM registers. In this paper, we are interested in operations that can be performed on integers. As most of the other cryptographic algorithms, the SHA uses integer data and performs operations on integers.

Intel's SIMD architecture provides appropriate instructions for each operation mentioned above. Thus SHA algorithms are fully SIMD compatible in terms of its operations. The only operation that requires more than one instruction to implement is rotation. This operation will be discussed in detail later.

We are interested in two types of parallelism that can be achieved using SIMD instructions: *parallelism within a thread* and *thread-level parallelism*. We use the word *thread* to mean the ensemble of all the operations to hash one message. Hashing more than one message simultaneously is referred as thread-level parallelism. The purpose of parallelism within a thread is to speed up hashing a single message by performing SIMD compatible operations on several data elements at the same time.

In order to perform the same operation on different data simultaneously, the values that the operation uses has to be known in advance. The level of parallelism depends on how early the values to be used are known. The level of parallelism is the number of operations that can be executed together.

Thread-Level Parallelism: We need appropriate SIMD instructions for each operation used in SHA algorithms to implement thread-level parallelism. Fortunately, Intel's SIMD architecture contains all the required SIMD instructions to perform all operations of SHA algorithms.

SHA-1 and SHA-256 perform operations on 32-bit words. 64-bit MMX registers can store two 32-bit words, so we can hash 2 independent messages simultaneously using one of these two algorithms. Moreover, we can also hash 4 independent messages at the same time if 128-bit XMM registers are used.

SHA-384 and SHA-512 perform operations on 64-bit words. If MMX registers are used, no parallelism can be achieved since an MMX register can only hold one 64-bit word. However we can obtain a high performance gain by using MMX registers instead of 32-bit general purpose registers. We can hash two messages in parallel by using XMM registers.

In both cases more messages can be hashed using XMM registers, which is expected due to the size of the registers.

Parallelism within a Thread: To speed up hashing, we combine same operations of different rounds and use SIMD instructions to perform these operations at a time. In order to combine the same operation of two consecutive rounds, we must know the values that will be used in the next round while we are processing the previous rounds. If we know these values in advance, we can successfully convert these operations into one SIMD instruction. We need to analyze each operation of SHA algorithms to determine whether or not this operation is SIMD compatible.

The rest of this section introduces the results of our analysis. The following subsections give the SIMD compatible operations of each SHA algorithm.

3.3.1. SHA-1

There are five main parts of the algorithm that SIMD instructions can be used. Most important of them is message scheduling.

$$W_t = (W_{t-3} \oplus W_{t-8} \oplus W_{t-14} \oplus W_{t-16})^{\Leftarrow 1}$$

To be able to compute W_t, we first have to compute W_{t-3}. Therefore W_t of only three consecutive rounds can be computed simultaneously. However the computation of

$$W_{t-8} \oplus W_{t-14} \oplus W_{t-16}$$

can be executed 8 at a time. On the other hand, the maximum number of parallel operations that can be executed is 4 due to the restrictions of the architecture. If MMX registers or XMM registers are used, we can perform two 32-bit operations or four 32-bit operations, respectively. So the maximum level of parallelism that can be reached is 4.

Another part of the algorithm we can apply SIMD instructions is the summation: $E + W_t + K_t$. After completing message scheduling, we can perform the addition $W_t + K_t$ of all rounds at the same time. But again we are restricted by 4 operations at a time. The first addition requires the value of E, which is determined by $B^{\Leftarrow 30}$ operation executed three rounds before. Thus, at most three summations can be performed together simultaneously.

Other SIMD compatible operations are the computations of $f_t(B, C, D)$ and $B^{\Leftarrow 30}$. The values of B and D of a round are the same as the values of A and C of the previous round, respectively. The value of C used to calculate $f_t()$ is the result of $B^{\Leftarrow 30}$ operation of the previous round. This gives us the opportunity to use SIMD instructions for calculations of $f_t(B, C, D)$ and $B^{\Leftarrow 30}$.

If we want to compute two $f_t(B,C,D)$ operations in parallel, we first need to compute two $B^{\Leftarrow 30}$ operations in parallel. If SIMD instructions are not used to calculate $B^{\Leftarrow 30}$, we can still fasten the computation of $f_t(B,C,D)$ by using SIMD for the parts that only B and D are involved. In this case, it is more efficient to use the Boolean equation as

$$\mathrm{Maj}(X,Y,Z) = [Y \wedge (X \vee Z)] \vee (X \wedge Z) \ .$$

Operation	LoP
$W_{t-8} \oplus W_{t-14} \oplus W_{t-16}$	4
$(W_{t-3} \oplus W_{t-8} \oplus W_{t-14} \oplus W_{t-16})^{\Leftarrow 1}$	3
$W_t + K_t$	4
$E + W_t + K_t$	3
$f_t(B,C,D)$	2
$B^{\Leftarrow 30}$	2

FIGURE 3.3. SIMD compatible operations of SHA-1 and the levels of parallelism (LoP).

3.3.2. SHA-256

Because SHA-256 uses 32-bit words, we can perform 2 operations in one SIMD instruction if MMX registers are used, or we can perform 4 operations if we use XMM registers. The most important part of SHA-256 that SIMD instructions can be successfully mounted is message scheduling.

$$W_t = \sigma_1(W_{t-2}) + W_{t-7} + \sigma_0(W_{t-15}) + W_{t-16} \ .$$

To be able to compute W_t, we first have to compute W_{t-2}. Because of it, we can just compute W_t of two consecutive rounds simultaneously. However, the computation of

$$W_{t-7} + \sigma_0(W_{t-15}) + W_{t-16} \ .$$

can be executed 4 at a time using XMM registers.

SIMD instructions can also be used in the summation $H + W_t + K_t$. After completing message scheduling, we can perform the addition $W_t + K_t$ of all rounds at the same time. But we are restricted by 4 operations at a time. The first addition requires the value of H, which is determined by $D+T_1$ operation executed four rounds before. Therefore, four summations can be performed together simultaneously.

Operation	LoP
$W_{t-7} + \sigma_0(W_{t-15}) + W_{t-16}$	4
$\sigma_1(W_{t-2}) + W_{t-7} + \sigma_0(W_{t-15}) + W_{t-16}$	2
$W_t + K_t$	4
$H + W_t + K_t$	4

FIGURE 3.4. SIMD compatible operations of SHA-256 and the levels of parallelism (LoP).

3.3.3. SHA-384 and SHA-512

These algorithms use 64-bit words, so maximum level of parallelism that can be achieved is two when XMM registers are used. Because the structures of these algorithms and SHA-256 are same, the SIMD compatible parts of these three algorithms are also same. The only difference is the maximum level of parallelism that can be achieved, which is only 2 for SHA-384 and SHA-512.

Operation	LoP
$W_{t-7} + \sigma_0(W_{t-15}) + W_{t-16}$	2
$\sigma_1(W_{t-2}) + W_{t-7} + \sigma_0(W_{t-15}) + W_{t-16}$	2
$W_t + K_t$	2
$H + W_t + K_t$	2

FIGURE 3.5. SIMD compatible operations of SHA-384 and SHA-512 and the LoP.

4. IMPLEMENTATION DETAILS AND RESULTS

In this section we give the details of our implementations and report the performance results. First we present scalar implementations of all four SHA algorithms and then use the performance of these implementations as a base to analyze the effect of SIMD instructions on the SHA.

4.1. Optimization Techniques

Here we explain the optimization techniques used in our implementations:

4.1.1. Loop Unrolling and Renaming Registers

Unrolling the loops eliminates the loop overhead and index calculations. It also allows us to rename the registers instead of shifting them in each iteration. The main computation gain can be reached by eliminating the shifting phase of state variables. We have to unroll the loops by the number of state variables to obtain this gain. Full unrolling releases one register and eliminates a branch misprediction. However, increased code size causes more cache misses and page faults. Therefore there is a tradeoff between reduced computation and increased code size. Every possible combination of loop unrolling are implemented for each algorithm.

4.1.2. Redefining Boolean Functions

We reduced the number of instructions by redefining boolean functions. Figure 4.1 shows the original definition of two boolean functions with the corresponding new definitions.

Boolean Functions		
	$OriginalDefinition$	$ModifiedDefinition$
$Ch(x, y, z)$	$(x \wedge y) \oplus (\neg x \wedge z)$	$(z \oplus (x \wedge (y \oplus z)))$
$Maj(x, y, z)$	$(x \wedge y) \oplus (x \wedge z) \oplus (y \wedge z)$	$(y \wedge (x \vee z)) \vee (x \wedge z)$
$Parity(x, y, z)$	$x \oplus y \oplus z$	$SAME$

FIGURE 4.1. Redefined Boolean functions.

4.1.3. Pre-fetching Data

PREFETCH*h* instruction was introduced in Pentium 3 processors as a part of SSE instruction set. It load data from memory to a selected level of cache. Pre-fetching data reduces the effect of data transfer latencies by overlapping hash computations and data transfer.

4.1.4. Memory Alignment

SIMD parallelization suffers from unaligned memory operations. Furthermore, a good vectorization requires appropriate data arrangement in memory. Therefore we paid extra attention to memory alignment and arrangement.

4.1.5. Common Optimization Techniques

We reduced data dependencies as much as possible by achieving a good instruction scheduling. We used simple instructions and reduced memory accesses by avoiding unnecessary work. However we had to use a significant number of memory accesses, because of the very limited number of registers in Pentium 4. We try to keep the frequently used values in registers.

4.1.6. Other Important Issues

There are two more issues we want to stress:

4.1.6.1. Rotations

The implementations were coded in the assembly language because we would like to have full control over the instructions used. The Rotation operation, which is heavily used in SHA, is a good example of this advantage. We give the definition of rotation operation in C++ below.

$$\text{define Rotation(x, s) } ((x \ll s)|(x \gg (32 - s))$$

This requires at least 4 instructions: 1 move, 2 shifts and 1 OR. However we can implement this operation just using one instruction, **rol**, and save at least 3 instructions. Although there are always other options we can use, i.e., using Intel C++ compiler and intrinsics [9], the use of assembly language gives use the best control over the program.

4.1.6.2. Endianness

Because Pentium-4 is a little-endian machine, we need to swap the bytes of the message to feed the compression function. Intel's instruction set has **bswap** instruction which swaps the data in a 32-bit register. But the unavailability of such instruction for MMX or XMM registers degrades the performance of the SIMD-based implementations.

4.2. Performance Results

The compression functions were coded using inline assembly language and the programs were compiled and executed on the Windows XP operating system. We used 2.4 GHz Pentium-4 processor with 256 MB of main memory to test the programs and obtain the performance results. We used large files of approximately 400 MB in order to obtain reliable timing results by minimizing the effect of one-time operations like message padding or file opening.

The scalar implementation, which is coded first, use only 32-bit registers and employs the corresponding instructions that operate on those registers. We considered all possible optimizations [2, 14, 24, 4, 10] mentioned above and applied different combinations of them to obtain the best results. We coded several versions of the algorithm and determined the best one. The performance of each algorithm is shown in Figure 4.2.

	From Main Memory	From Hard Disk
SHA-1	87.12 MB/s	67.08 MB/s
SHA-256	48.88 MB/s	37.64 MB/s
SHA-512	38.26 MB/s	30.60 MB/s

FIGURE 4.2. Performance of the scalar implementations in terms of bandwidth.

Two different timings are given for each case of hashing a message from main memory and hashing a file from hard disk. The performance for the second case depends on the architecture of the attached disk system. Therefore, the performance of the first case is much reliable and we will use the results of hashing from main memory in our analysis.

Figure 4.3 shows the comparison of our implementation to some open source implementations [17–19]. We compiled these codes using Intel C++ compiler with speed optimizations. The same configuration, mentioned above, is used to obtain timing results. As can be seen in the figure, our implementation is very efficient even without SIMD architecture support.

	Raw Performances (MB/s)			Relative Performances		
	SHA-1	SHA-256	SHA-512	SHA-1	SHA-256	SHA-512
Gladman	76.66	43.52	24.33	0.88	0.89	0.64
Devine	-	43.19	-	-	0.88	-
Crypto++ 5.1	67.45	23.90	-	0.77	0.49	-
Our Code	87.12	48.88	38.26	1.00	1.00	1.00

FIGURE 4.3. Comparison of our scalar implementation to other public implementations.

There are mainly four basic blocks (BB) in compression functions regardless of the SHA version:

- **BB1:** Retrieving and swapping message block
- **BB2:** Calculating message schedule
- **BB3:** Main compression rounds
- **BB4:** Retrieving and updating state variables

The most time consuming basic block is the third one, since it includes the main compression rounds. Figure 4.4 shows the percentage of time spent on each block.

We first present the results of SHA-1 and SHA-256. Their results are similar because they both use 32-bit words.

Basic	Algorithm		
Block	SHA-1	SHA-256	SHA-512
BB1	5%	3%	3%
BB2	14%	14%	27%
BB3	77%	80%	69%
BB4	4%	3%	1%

FIGURE 4.4. Percentage of time spent on each basic block.

We employed both MMX and XMM registers and their appropriate operations. We packed two 32-bit words in an MMX register and performed the same operation on them simultaneously. This reduces two 32-bit instructions into one 64-bit instruction. Then we packed four words in an XMM register and reduced four 32-bit instructions into one 128-bit instruction.

We realized that neither of them yields a better performance for hashing a message. The main reason is higher instruction latency and throughput SIMD instructions have. Figure 4.5 shows the approximate latency and throughput values of the Pentium 4 instructions [9]. As can be seen, the performance of the operations is proportional to the size of the operation. 128-bit operations take four times, 64-bit operations take twice as much time as 32-bit operations. Therefore we can confidently say that reducing 32-bit instructions into higher-bit instructions does not give any performance gain.

The throughputs of these instructions are same regardless of the size of the operation. However we performed some experiments to find out which SIMD instructions are faster. These experiments show that while SIMD move and shift instructions have higher throughput than their 32-bit counterparts, addition has a lower throughput.

Instructions	Latency	Throughput
Addition, subtraction, increment, decrement, logic(AND, OR, XOR), compare, test, jump, memory move, call, return	0.5	0.5
Push, pop, rotate, shift, SIMD memory moves, 64-bit technology operations	1	1
128-bit SIMD integer operations	2	2

FIGURE 4.5. Approximate instruction performance on the Pentium 4 processor.

Another reason is the lack of rotation and byte swap instructions in Intel's SIMD architecture. We have to use 4 different instructions for one rotation. Swapping bytes is much more costly in terms of number of required instructions. Also there is no immediate loading instructions for SIMD registers. First we need to put the immediate value into a 32-bit register and load this register to the target SIMD register. Therefore we have to use more than one instruction to load any SIMD register with an immediate value, which degrades the performance.

Embedding SIMD operation into a scalar implementation brings additional overhead due to the load/store operations between SIMD and 32-bit registers. A good approach is to implement a large portion of an algorithm using only SIMD architecture. However, the largest portion of SHA, the third basic block, which is the bottleneck of the hash function, has a sequential structure due to the nature of a hash algorithm.

We cannot obtain faster implementations of SHA-1 and SHA-256 using Intel's SIMD architecture because of the slow SIMD instructions and the lack of some operations, however, we believe that a performance gain can be obtained by applying the presented ideas on a different platform, which has a faster SIMD architecture.

SHA-512 uses 64-bit words and performs operations on these 64-bit values. Therefore SHA-512 favors 64-bit architectures. Instead of packing two 32-bit words in an MMX register, we use the whole register to perform 64-bit operations. Using this idea, we yield much higher performance of 55.71 MB/s with a speed up of 1.46.

An advantage, SIMD technique brings us, is the capability of performing more than one hashing simultaneously. We can hash 4 different messages at the same time if we use 128-bit register and a hashing algorithm with 32-bit word size. If the word size of the algorithm is 64-bit, as in the case of SHA-512, we can perform 2 simultaneous hashing with the same registers.

Number of		SHA-1		SHA-256	
Streams	Architecture	Throughput	Speedup	Throughput	Speedup
1	32-bit	87.12 MB/s	1.000	48.88 MB/s	1.000
2	64-bit (MMX)	128.46 MB/s	1.474	73.22 MB/s	1.498
4	128-bit (XMM)	143.09 MB/s	1.642	83.81 MB/s	1.715

FIGURE 4.6. Performance results of simultaneous hashing for SHA-1 and SHA-256.

Again we used the fastest scalar implementations of SHA-1 and SHA-256 and exchanged 32-bit instructions with their SIMD counterparts. We implemented both hashing 2 messages using MMX registers and 64-bit operations and 4 messages using XMM registers and 128-bit operations. We paid attention to give the input messages of the same size while measuring the performance in order to get more accurate results. SHA-1 takes 35.6% more time to hash two messages simultaneously. However, the throughput of the algorithm increases 47.4% since the size of the input doubles. Figure 4.6 and Figure 4.7 show all the results and the speedup obtained by this technique.

Number of		SHA-384 & SHA-512	
Streams	Architecture	Throughput	Speedup
1	32-bit	38.26 MB/s	1.000
1	64-bit (MMX)	55.71 MB/s	1.456
2	128-bit (XMM)	88.02 MB/s	2.301

FIGURE 4.7. Performance results of simultaneous hashing for SHA-384 and SHA-512.

Figure 4.8 shows the overall performance results we obtained with and without using SIMD architecture.

Because Pentium-4 is a very complex processor system, its behaviour and performance is difficult to predict accurately. It is often infeasible to completely understand how it works and processes data without using advanced performance analyzers and profiling tools. We believe that the performance of our implementations can be improved further using high-end performance and profiling software.

	SHA-1	SHA-256	SHA-512
Gladman	0.88	0.89	0.64
Devine	-	0.88	-
Crypto++ 5.1	0.77	0.49	-
Our Code (1 stream)	1.00	1.00	1.00
Our Code (2, 2, 1 streams)	1.47	1.50	1.46
Our Code (4, 4, 2 streams)	1.64	1.72	2.30

FIGURE 4.8. The relative performances of the implementations.

5. CONCLUSION

We demonstrated that it is possible to obtain high-throughput implementations of SHA by employing SIMD techniques if multiple independent streams of messages are to be hashed. In addition, we derived a number of guidelines applicable to implementations of SHA on current processor families other than Intel Pentium. The final conclusion of this study is that while the Pentium SIMD instructions are slow for obtaining a faster implementation of the SHA of a single stream of data, faster implementation of SIMD instructions will result in better performance, and that we can still obtain faster SHA by hashing two or four independent streams of message on the Pentium SIMD architecture, obtaining a speedup of 1.474 to 2.301.

BIBLIOGRAPHY

[1] AMD. *AMD Extensions to the 3DNow! and MMX Instruction Sets*, March 2000.

[2] D. Anderson and T. Shanley. *Pentium Processor System Architecture.* Addison-Wesley, 1995.

[3] D. Bistry, C. Delong, and M. Gutman. *The Complete Guide to MMX Technology.* McGraw-Hill, 1997.

[4] R. Booth. *Inner Loops: A Sourcebook for Fast 32-bit Software Development.* Addison-Wesley, 1997.

[5] A. Bosselaers. Even faster hashing on the Pentium. In *Rump Session, EUROCRYPT 97*, 1997.

[6] A. Bosselaers, R. Govaerts, and J. Vandewalle. Fast hashing on the Pentium. In N. Koblitz, editor, *Advances in Cryptology - CRYPTO 96*, pages 298–312. Springer Verlag, LNCS Nr. 1109, 1996.

[7] A. Bosselaers, R. Govaerts, and J. Vandewalle. SHA: A design for parallel architectures. In W. Fumy, editor, *Advances in Cryptology - EUROCRYPT 97*, pages 348–362. Springer Verlag, LNCS Nr. 1233, 1997.

[8] R. Buchty. *Cryptonite - A Programmable Crypto Processor Architecture For High-Bandwidth Applications.* PhD thesis, Technische Universität München, December 2002.

[9] R. Gerber. *The Software Optimization Cookbook.* Intel Press, 2002.

[10] Intel Corporation. *Desktop Performance and Optimization for Intel Pentium 4 Processor*, February 2001.

[11] Intel Corporation. *IA-32 Intel Architecture Optimization*, 2003.

[12] Intel Corporation. *IA-32 Intel Architecture Software Developer's Manual, Volume 1, 2, and 3*, 2003.

[13] K. McCurley. A fast portable implementation of the secure hash algorithm, III. Technical Report SAND93-2591, Sandia National Laboratories, 1994.

[14] H.-P. Messmer. *The Indispensable Pentium Book.* Addison-Wesley, 1995.

[15] Sun Microsystems. VIS Instruction Set. http://www.sun.com/processors/vis.

[16] Motorola. AltiVec Technology. http://www.simdtech.org/altivec.

[17] B. Gladman. SHA1, SHA2, HMAC and Key Derivation in C, January 2004. http://fp.gladman.plus.com/cryptography_technology/sha/index.htm.

[18] C. Devine. SHA-256 Source Code, 2003. http://www.cr0.net:8040/code/crypto/sha256/.

[19] W. Dai. Crypto++ Library 5.1, 2003. http://www.eskimo.com/ weidai/cryptlib.html.

[20] E. Nahum, S. O'Malley, H. Orman, and R. Schroeppel. Towards high performance cryptographic software. Technical Report 95-04, Department of Computer Science, University of Arizona, March 1995.

[21] E. Nahum, D. Yates, S. O'Malley, H. Orman, and R. Schroeppel. Parallelized network security protocols. In *Internet Society Symposium on Network and Distributed System Security*, pages 145–154, San Diego, California, February 1996. IEEE Computer Society Press.

[22] National Institute for Standards and Technology. Specifications for the secure hash standard. FIPS Publication 180-1, April 2002.

[23] National Institute for Standards and Technology. Specifications for the secure hash standard. FIPS Publication 180-2, August 2002.

[24] K. R. Wadleigh and I. L. Crawford. *Software Optimization for High Performance Computing.* Prentice-Hall, 2000.

APPENDICES

Performance Metrics of Our Code

VTune[1] is a performance tuning environment developed by Intel. It monitors the performance of all active software, identifies hotspots of a program, examines each instruction and uncovers problems at machine level code. Information collected with VTune includes instruction distribution, branch prediction rate, instruction/micro-op executed per cycle, and some performance impact values. These performance impact values indicate the effect of coding pitfalls on the performance of the program. The higher values of these indicators pinpoint the part of the code which needs optimizing. We can probably say that the values below 1 are not worthy of attention.

[1]Intel Corporation, URL: `http://www.intel.com/software/products/vtune`

SHA-1	scalar	MMX	XMM
Clockticks per Inst. Retired (CPI)	0.815834499	0.855176209	1.537516440
Cycles per Retired Micro-op	0.805213576	0.748647956	0.571526585
64-bit MMX(TM) Instructions	0	50.96443191	0
128-bit MMX(TM) Instructions	0	0	75.76040523
Branch Prediction Rate	98.94263022	98.96110742	98.88476093
Branch Mispredict Performance Impact	0.033615766	0.024292926	0.014051956
Split Loads Performance Impact	0	1.133185232	1.240289630
Split Stores Performance Impact	6.30595E-06	3.37866E-06	1.85101E-06
64K Aliasing Conflict Performance Impact	0.009252035	0.908163234	0.518454523
Trace Cache Miss Performance Impact	0.011808933	0.054913404	0.038886455

SHA-256	scalar	MMX	XMM
Clockticks per Inst. Retired (CPI)	0.723946089	0.752008262	1.225010585
Cycles per Retired Micro-op	0.718887409	0.686999191	0.599415492
64-bit MMX(TM) Instructions	0	53.32294728	0
128-bit MMX(TM) Instructions	0	0	66.00791075
Branch Prediction Rate	97.68053657	98.23620053	98.22454836
Branch Mispredict Performance Impact	1.741436524	0.827699474	0.509243529
Split Loads Performance Impact	0	0.610750101	0.743397310
Split Stores Performance Impact	1.63E-06	1.82E-06	1.20E-06
64K Aliasing Conflict Performance Impact	0.075797952	0.732646357	0.806864142
Trace Cache Miss Performance Impact	0.027144073	0.013025083	0.023640639

SHA-512	scalar	MMX	XMM
Clockticks per Inst. Retired (CPI)	0.537688353	0.750974844	1.213004122
Cycles per Retired Micro-op	0.446157025	0.688945017	0.592610353
64-bit MMX(TM) Instructions	0	53.05749799	0
128-bit MMX(TM) Instructions	0	0	57.27867009
Branch Prediction Rate	98.43921139	98.65968723	98.20767492
Branch Mispredict Performance Impact	0.637789356	0.706421601	0.529223701
Split Loads Performance Impact	6.10E-07	0.484165685	0.385366994
Split Stores Performance Impact	2.03E-06	0.806640353	2.51E-06
64K Aliasing Conflict Performance Impact	0.244594487	0.583929872	0.460278587
Trace Cache Miss Performance Impact	0.039623983	0.014963164	0.028386361