Parallelizing MD5 on Intel Architectures

Johannes Schlumberger   Alexander Würstlein

March 3, 2009

MD5 is widely used as a cryptographic hash-function but also, for its easy implementation, as a benchmark for integer operations on various architectures. We provide an improved implementation on current Core 2 processors utilizing SIMD instructions as well as multiple cores, which can serve as a comparative benchmark and a building block for forensic software.

1 Introduction

Many publications use MD5 [6] as a benchmark to compare the speed of integer operations on various architectures. MD5 is by design suited to this purpose, because it consists of simple bit-operations on 32-bit registers with only few or no memory operation necessary. Intel CPUs serve as a basepoint for comparisons with massively parallel architectures such as GPUs and FPGAs. We will show that by using various parallelisation techniques leveraging SIMD and multiple cores a significant increase in the number of iterations performed per second is possible.

Possible applications beyond benchmarking include forensic and cryptographic techniques such as the creation and use of rainbow-tables, data integrity and detection of transmission errors.

The MD5 algorithm is a well-known hashing algorithm which is widely used in cryptographic signature schemes such as HMAC, public-key cryptography as well as to check for transmission errors in electronic communication. It is also used to securely[1] hash passwords for authentication purposes.

Most prominent examples are the extended Unix `crypt` function, the later versions of Windows LAN-Manager hashes and Kerberos.

The concrete implementation described by this paper relies on several assumptions regarding the nature of input data. We will first describe those assumptions and show their applicability to our usage scenario, also leading to several important

[1] As of yet, the known collission-based attacks [4] on MD5 rely on beeing able to append large amounts of data to a message. This makes those attacks impractical for purposes of finding collisions to passwords.

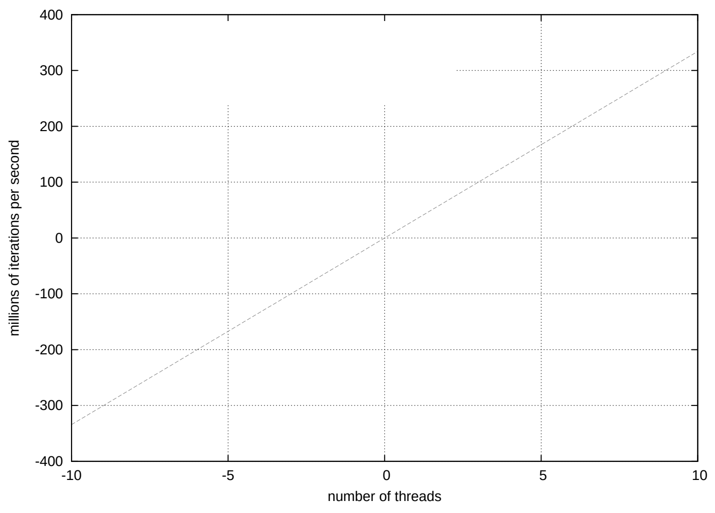

Figure 1: Performance in iterations per second with different numbers of threads on various CPUs for our 32 bit and 64 bit version. The dashed line is a linear extrapolations of the single threaded performance of the 16 core machine with the 64bit implementation to show the linear relation between the number of threads and the measured performance.

optimizations. Furthermore we will describe our implementation and its performance to several pre-existing implementations on different architectures.

2 Setting

In our following implementation we assumed a hypothetical function for hashing passwords which is similar but not identical to Unix `crypt` in its MD5 mode. For a given password p and random salt s, the hashed password h is given by[2] $h = MD5(s.p)$. The main difference to Unix `crypt` is that only one round of MD5 is used, whereas in Unix `crypt` the password and salt are hashed 1000 times to increase the cost of brute-force attacks by that factor. Benchmarks between our implementation and Unix `crypt` are easily comparable by multiplying or dividing the resulting rounds per second by 1000.

Our program searches, for a given hash and salt, for the password belonging to that combination of hash and salt by simply iterating over all possible combinations of all letters in the given alphabet. Passwords are tried from the shortest passwords to the longer ones up to a max-

[2] with "." as the append operator

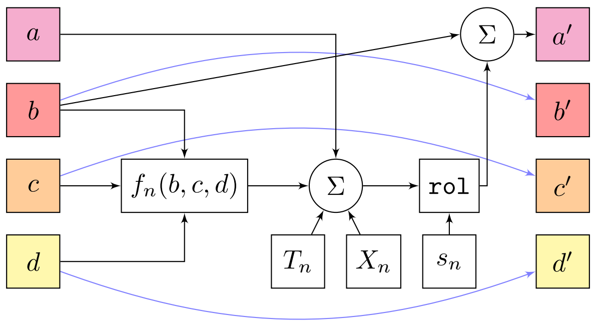

Figure 2: A single step of the MD5 hash function. Accumulated data from the previous step consists of 32bit values a through d in registers, after operations are performed, output values a' through d' are placed in those same registers. Values T_n are taken from a table of integer constants which is part of the MD5 specification. X_n is the appropriate part of the message to be hashed. s_n is the amount of bits by which the rotate-left operation is to be performed, this is also part of the specification. Only a is changed in each step, the other registers are left unchanged.

imum of 25 characters. The number of threads used can be specified.

2.1 MD5

Input data for MD5 is padded to a multiple of 512 bits by adding a single "1" bit after the message followed by an appropriate amount of zeros, such that, after adding the lower 64 bit of the length of the unpadded message, the whole input data is the desired multiple of 512 bits long.

Hashing is performed in as many iterations as there are 512 bit blocks. Each iteration consists of 4 rounds and each of those rounds consists of 16 steps like the one depicted in figure 2. The registers a through d are initialized by special constants.

In the n-th step, accumulated data is taken from registers a through d. First, the function $f(b, c, d)$ is applied. f can be any of four bit-operations on the inputs b, c and d like for example $b \wedge c \vee (\neg b) \wedge d$. f from each round to the next. To the result of f one of 64 constants T_n derived from the $|\sin(x)|$ function and one of the 16 input words from the current block of the padded message are added. A rotate left operation by s_n bits is performed and finally b is added and the end result placed in a. During those operations, b, c and d are left unchanged.

Between two steps the registers a, b, c and d are cyclically swapped. Thus, the second step will operate on registers b, c, d and a, and so on. After an iteration is completed, the registers a through d are incremented by their values before the iteration. After the last iteration, the hash is given by a through d.

We assume that our passwords are shorter than 25 bytes and that the salt is shorter than 6 bytes. This means that

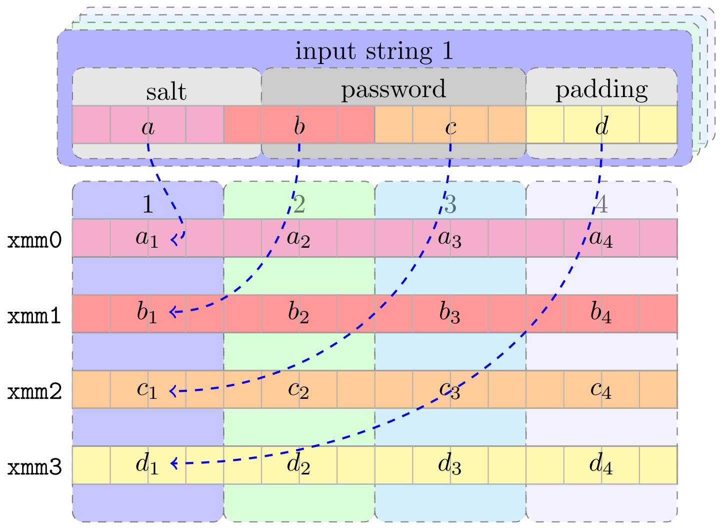

Figure 3: Usage of SSE registers by our implementation. A set of four SSE registers (e.g. `xmm0` through `xmm3`) holds 4 concurrent MD5 iterations 1 through 4, where each stores its accumulated values a through d in the appropriate 4 bytes of each SSE register as shown e.g. for input string 1. The inputs are interleaved in memory so that they can be directly loaded into the SSE registers.

we will only have to perform one iteration because the length of the unpadded input is shorter than $512 - 64$ bits.

3 Implementation

3.1 Threads

Parallelization is achieved in two distinct ways. First there are a (configurable) number of threads amongst which the search space is divided. After the threads are started, there is no further need for any communication, except in the case of termination through a thread finding the wanted cleartext password or exhaustion of its search space. Therefore our implementation scales linearly with the number of threads as shown in figure 1.

3.2 SSE

More importantly, we calculate several MD5-Sums in parallell on every single core. This is achieved through the SSE instruction set in current x86 processors. Usually MD5 is calculated using 4 registers of 32 bits each. In our implementation, 4 of those 32 bit registers are replaced by a single 128 bit wide SSE register as shown in figure 3. Because there is a non-removable data-dependency between every step of a MD5 iteration, it is not possible to speed up the runtime of a single iteration, instead the throughput is increased.

For four parallel MD5 iterations, four SSE registers are always needed. Additionally we need three temporary registers for prefetching the data to be hashed and to perform the calculation steps. Therefore, in the 32 bit mode where in total 8 SSE registers are available, we are able to perform 4 parallel MD5 iterations. In 64 bit mode, 16 SSE registers are available, leading to a theoretical maximum of 12 parallel MD5 iterations. As one can see in figure 1 this theoretical factor of 3 between the 64bit and 32bit version is not reached, in our benchmarks, the 64bit code is only about twice as fast.

We decided to use assembler as the implementation language for the inner loop function performing the MD5 algorithm. This function is called by a C program which generates all possible passwords[3], in a format fit to be loaded into SSE registers for our algorithm as shown in figure 3 and evaluates the return value to inform the user if a password has been found. Also threads are set up and the search space is divided among the threads. Various minor optimizations in the assembler code were used, for example branch prediction prefixes, prefetching of operands from memory into a temporary register and removal of constant operands[4].

3.3 Memory Access

The memory footprint of our algorithm is fairly minimal. Only the password generation and the constants needed for the MD5 steps are read from memory, which will, given the very small size (some hundred bytes) of the respective areas, easily fit into the L1 cache of a modern CPU. Also there is no writing into shared data structures, so that no thread should invalidate the cache of another core.

4 Related Work and Comparison

Several implementations of MD5 for brute-force attacks on password hashes are available ([3], [1], [2], [5]). In the case of "John the Ripper" [1] by which our implementation was inspired, sourcecode is available for review. On the Xeon machine used (see figure 1) John reaches about 7.3 million MD5 iterations per second in a single thread, whereas our 64 bit implementation reaches 33.4 million iterations per second. However, there are some differences which might affect performance, because John the Ripper builds its passwords by a more complicated grammar using dictionaries.

An earlier unpublished implementation by the authors of this paper using OpenSSL performed around 0.8 million iterations per second.

MDCrack [3] claims 42.3 million iterations on a 3.2GHz Xeon. 108 million iterations per second are claimed on [5] for a comparable MD5 search on a GeForce 8800 Ultra via CUDA. BarsWF [2] even claims over 300 million iterations for its GPU implementation and over 100 millioin iterations for its implementation on SSE.

[3]all words matching the regular expression $a\{1, n\}$ where a is an element of all character classes and n the maximum length specified on the command line

[4]This is based on the assumption made above, that passwords are short. Therefore most of the block to be hashed will always be padded with zeros, which saves us one addition and a fetch from memory

5 Conclusion

We have shown clear that definite improvements are possible when fully utilizing the available possiblities for parallelization. The performance of previous implementations was clearly exceeded by a significant factor. Yet there are some faster closed source implementations and implementations on different hardware like GPUs.

Possible fields of use for the techniques shown include the creation of rainbow tables, parallell integrity checking for high-throughput communications and networking devices or mass verification of certain signature methods such as HMAC-MD5. Also the implementation of different hashing algorithms such as SHA-1 would be easy to accomplish and should bring similar improvements in performance.

Further fields of study are numerous. While we used assembly language for our implementation, it is also possible that higher level languages could achieve a similar level of performance. Therefore a comparison of different implementations and toolchains with respect to the optimizations performed and the performance achieved should prove interesting. Architectures besides Intel x86 such as IBM's Power and Sun's Niagara also feature a number of cores and SIMD capabilities. A port of our software could therefore serve as a point of comparison between those different architectures, and if compared with implementations in a high-level language, also for the quality of the respective toolchains.

References

[1] John the ripper `http://www.openwall.com/john/`.

[2] Svarychevski Michail Aleksandrovich. Barswf `http://3.14.by/en/md5`.

[3] Gregory Duchemin. Mdcrack `http://c3rb3r.openwall.net/mdcrack/ln.html`. 2007.

[4] Xiaoyun Wang et al. How to break md5 and other hash functions `http://web.archive.org/web/20070604205756/http://www.infosec.sdu.edu.cn/paper/md5-attack.pdf`. 2005.

[5] Mario Juric. Notes: CUDA MD5 Hashing Experiments `http://majuric.org/software/cudamd5/`.

[6] R. Rivest. RFC 1321 - The MD5 Message-Digest Algorithm. 1992.