Efficient execution of the MD5 algorithm in password recovery applications

Sam Mitchell and Nathanael Weidler
Deptartment of Electrical and Computer Engineering
Utah Stat University
Logan, Utah 84322
e-mail: samuel.alan.mitchell@gmail.com, NWeidler@gmail.com

***Abstract*—Summarize project and results (executive summary).**

I. Introduction

Describe the problem and what you're doing to address it (i.e. building architecture-informed md5 password cracker for software and hardware; how password cracking operates).

A. Related work

What is needed to understand your optimisations and/or implementations. Do you draw on existing techniques? If so, describe and cite them.

B. Structure of paper

Which sections discuss which aspect of the work.

II. Software implementation

Overview and objectives MD5 is a computationally simple operation with minimal execution time. An inefficient implementation of MD5 is sufficient for everyday hashing needs, however password recovery applications require an efficient method to test large numbers of passwords.

Parallel computing is an effective method to quickly analyze large sets of data. OpenMP is one parallel computing implementation which has been in use for C/C++ since 1998 [1]. The authors utilized this parallel computing method with 16 threads, separating the set of passwords into 16 equal portions.

Further optimization methods were selected or developed. These methods are: Single Instruction Multiple Data, loop unrolling, string precalculation, and first block precalculation.

A. SIMD

Single Instruction Multiple Data (SIMD) is a functionality of standard processors which allows a single operation to operate on multiple integers in parallel [2]. This is effective due to the current register size (128-bits) of modern cpus.

The MD5 hashing algorithm operations are performed on 32-bit integers. This allows for four (4) simultaneous operations on each core, which led us to believe that the algorithm would see a 300% speedup.

Our implementation of SIMD initially slowed down the hash — a 37% decrease in performance when compared to the nominal case. Reasons for this are not confirmed, however we theorize that the stack operations resulting from context switches caused this slow-down.

In the following optimizations, the SIMD implementation is shown to outperform the Single Instruction Single Data case.

B. Loop unrolling

Loop unrolling is an elementary optimization technique utilized to remove the overhead from loop operations [3]. The loop was unrolled to a specific length to assist in string precalculation. In the nominal (no SIMD) case, loop unrolling to 26 iterations gave slight improvements (2%). In the SIMD case, loop unrolling to 169 iterations resulted in a substantial speed increase. Nominal to SIMD unrolled resulted in a 103% speedup, while SIMD to SIMD unrolled resulted in a 220% speedup.

C. String precalculation

One significant delay in MD5 calculation is string creation. We strategically combined loop unrolling with precalculation methods. The loop was originally unrolled to have 26 iterations (SIMD had $\frac{26*26}{4} = 169$ iterations), which allowed precalculation of the first characters of the string.

MD5 is calculated on 16 32-bit numbers, and on a length seven (7) string there are 13 32-bit numbers that are always 0. The last 32-bit number is 7*8=56. This is consistent on every length seven (7) string, so these values were directly coded into the MD5 hash.

When inserted to the already unrolled system, string precalculation resulted in a 36% speedup in the SIMD case.

D. First block precalculation

The first block of the MD5 calculation can be precomputed due to the repetitive nature of the password cracker — the first block only accepts the first four characters of the password. The first block was computed in function G_MD5() and stored for the successive 168 hashes. While this saved operations, the anticipated speedup was a maximum of $\leq \frac{1}{64}$ or 1.5% due to the nature of MD5. Testing this optimization revealed a 3% speedup, which was unexpected and as of yet unexplained.

E. Results

As expected, the combination of SIMD and loop unrolling was the optimization with the greatest speedup. When combined, these methods result in fewer comparisons and branches.

String precalculation was an unexpectedly effective optimization. This was not an improvement in MD5 calculation efficiency, but a removal of excessive overhead operations.

Benchmarking was performed on all seven (7) character combinations of a lowercase, alphabetic character set. Comparisons were made between optimizations as well as best-case to the nominal OpenMP optimization. The comparisons of each optimization are noted in the respective section. The best-case system is compared to the nominal using the -O1, -O2, -O3, and -fomit-frame-pointer compiler flags. Results are found in Table I.

Compiler flags are very important in optimizing execution time of a process. The 690% speedup between md5pc_O0 and md5pc_O3 hash rates shows that performing operations in an out-of-order manner is often more effective than readable code.

F. Password recovery

A hash of an unknown password was provided (aebc994aa5b00a0308c9fd257bf63ebd). In order to ensure that no passwords would be missed, password testing was performed in discrete groups: (1) 1-6 character strings, (2) 7 character strings, (3) 8 character strings, and so on. These tests were performed using the same processor used in the benchmarking tests in Table I.

The 1-6 character strings were tested without optimizations because it was executed in an acceptably short time. No results were found. 5.91 second execution time.

The 7 character string was tested using the program used in benchmarking. No results were found. 51.15 execution time.

The 8 character string was tested using a modified version of the benchmarking program. The resulting string "tyygsuef" was found in 6:18.85 (minutes:seconds).

In terms of total execution time, the testing was completed in 7:15.91. This does not represent testing every string of lengths 1-8, because the program ceased executing once the password was found.

III. Hardware implementation

A. Overview and objectives

MD5 is a simple algorithm to implement, however it takes time to find calculate the hash. Using a Field Programmable Gate Array (FPGA) we hope to improve the time it takes to create a single hash. For this assignment a Xilinx XC3S1200E-4FG320 FPGA will be utilized on a Digilent Nexys 2 development board. The FPGA contains 1200K gates, 19,512 logic cells with a total slice count of 8,672. ISE 13.4 Project Navigator were used to synthesize the design and ModelSim was used to simulate and test it. The goal for this implementation is to fully pipeline the design so that for every clock cycle a valid hash will be calculated.

B. ARCHITECTURE

For the architecture a hierarchical implementation was chosen. The basic building blocks of the design include reg32_top and add32. Reg32_top synthesizes a D-flip flop on the FPGA and add32 synthesizes a 32 bit adder. These are organized as can be seen in Figure 1. The inputs for each round are : A, B, C, D, K, T, RESET and CLK. The outputs are : A_OUT, B_OUT, C_OUT and D_OUT.

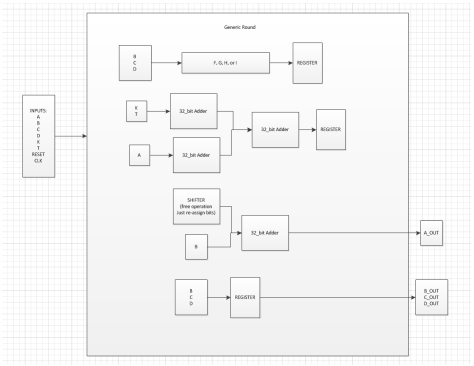

Figure 1: Generic Round Block Diagram.

For the top level hierarchy, this generic round is implemented 64 times - 16 times for each of the functions: F, G, H, and I. Then after these 64 cycles the outputs are added to the initial values to complete the MD5 hash algorithm. The top level block diagram can be seen in Figure2.

C. BENCHMARKING

The final benchmarking of this design was straightforward. It was determined using the performance equation : Throughput = (Block size * Clock Frequency)/(Cycles per block). The block size was 1, the clock frequency was up to 115MHz and the cycles per block was 1 so filling those numbers into the equation : (1*115Mhz/1) = 115 Mega Hashes per second. The maximum clock frequency was derived from the report given by Project Navigator that the best case achievable clock period was 8.680ns. 1/8.680ns = 115MHz.

D. Performance

The performance achieved was decent when thought of in hashes per dollar. The Nexsys 2 board can be purchased from Digilent for $200.00. 115 Megahashes per second divided by the cost of the board yeilds 0.575 Megahases per dollar or 5750 hashes per cent. This is much less costly than implementing it on a computer which can cost several hundreds of dollars more.

Higher performance can be achieved by procuring a larger FPGA. It is very simple to improve performance. All that is needed is to implement the same design multiple times on a larger FPGA and your performance is improved proportionate to the number of implementations instantiated. Another way is to increase clock speed on a different FPGA which can support it, or purchase the same FPGA with a greater speed grade.

The total gate count has been removed from the Xilinx design tools after the release of 10.1 so flip flops, 4 input LUTs and slices will be discussed instead. The FPGA used has a total number of 12,344 flip flops, of which, 12,044 were used or 69%, 7,588 of the available 17,344 4 input LUTs were used or 43% and 7,202 of the available 8,672 slices were occupied

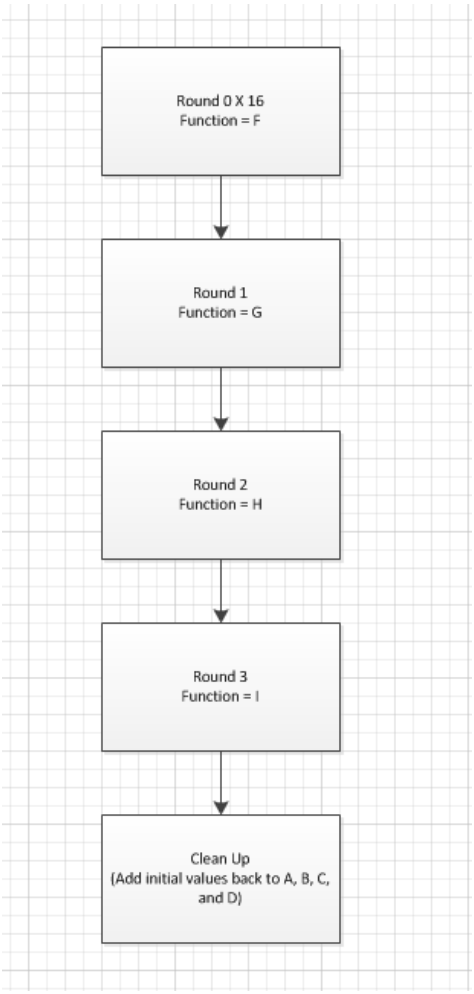

Figure 2: Top Level Block Diagram.

Table I: Benchmarking performed on a computer with two (2) Intel Core i7-4770K CPUs. Each CPU has 4 cores, 3.5 GHz, released June 2013.

Implementation	Hash rate (million hashes per second)
md5pc_base	40
md5pc_O0	20
md5pc_O1	155
md5pc_O2	157
md5pc_O3	158

or 83% [4]. Each slice contains two LUTs and two flip-flops. As can be seen, if the number of registers could have been reduced to under 50%, perhaps two top-level entities could have fit within the design. However, as I reduced the number of register, or pipeline stages, the maximum clock frequency dropped incredibly fast. In fact this design ended up being the one that yielded the best number of hashes per second.

IV. Conclusion

Summarize results and lessons learned.

References

[1] OpenMP Architecture Review Board, "The OpenMP API specification for parallel programming," http://openmp.org/wp/, 2015, [Online; accessed 02-Feb-2015].

[2] B. Hubert. (2004) Simd and other techniques for fast numerical programming with gcc. (Accessed: 4 February 2015). [Online]. Available: ds9a.nl/gcc-simd/

[3] J. J. Dongarra and A. Hinds, "Unrolling loops in fortran," *Software: Practice and Experience*, vol. 9, no. 3, pp. 219–226, 1979.

[4] Spartan-3e family data sheet. [Online]. Available: http://www.xilinx.com/support/documentation/data_sheets/ds312.pdf