MD5 Optimizations: Hardware and Software Implementations

Danny Froerer and Taylor Peterson
Utah State University Department of Electrical and Computer Engineering

***Abstract*—This paper discusses different optimizations of the MD5 hashing algorithm. Basic optimizations include a loop unrolled algorithm and the use of Intel AVX intrinsics. BLANK and BLANK were also used. A version of the MD5 was also implemented on a Diligent, Nexys 2 FPGA board demonstrating specific hardware optimizations.**

***Index Terms*—MD5, AVX Intrinsics, Hashing**

I. Introduction

As there are many implementations of the MD5 hashing algorithm, our goal was to combine different optimization strategies using the C coding language to increase the number of hashes per second. The baseline algorithm used was partially optimized when compared to the most basic version of MD5. The baseline included loop-unrolling and the use of Intel Advanced Vector Extension (AVX) Intrinsics. Other optimizations included BLANK and BLANK. These were then appended to the baseline code separately and together to verify that multiple, yet different optimizations could improve hashes per second rate. A version of MD5 was then written in Verilog and optimized to work specifically on an FPGA. This was done to show that by having a knowledge of the hardware, you can exploit and take advantage of it for a specific purpose.

II. Software Implementation

A very basic implementation of the MD5 algorithm was first written in the C programming language. After this basic version was verified to be working, both loop unrolling and AVX intrinsics were introduced into the code.

A. Loop Unrolling and AVX Intrinsics

Loop unrolling was a very basic and relatively simple optimization. This was done by explicitly stating each instruction inside of a loop thus eliminating extra instructions that control the loop. This increases the size of the program, but it also increases the execution speed of the program.
A more difficult, but worthwhile optimization was including AVX intrinsics into the code. AVX uses vectors to store information which allows the use of single instruction, multiple data (SIMD) parallel computing. An example of the power of parallel computing can be seen in Figure 1.

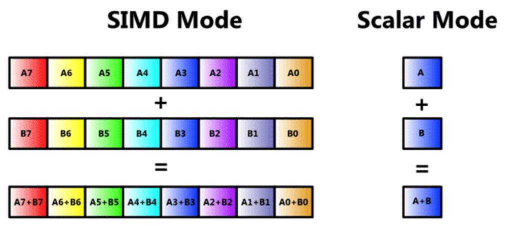

Fig. 1: SIMD vs. Scalar Arithmetic [1]

Combining both loop unrolling and the use of AVX intrinsics comprised the baseline version of the code. From that point, other optimizations were implemented used to optimize the code further.

B. Optimization I

After reviewing the baseline code, it was decided that when looking for a specific hash, depending on the number of initial candidates, it may not be necessary to compare the whole hash. This is because the hashed output is extremely random, and the probability of just the first half of the hash being equal to another hash is tremendously small. The first optimization implemented was only comparing the first half of the hash to see if there was a match and if not, the program would just continue. There was a very large increase in the hashing rate having implemented this shortened comparison.

III. Conclusion

The conclusion goes here.

References

[1] C. Lomont. (2011, June 21). *Introduction to Intel Advanced Vector Extensions*[Online]. Available: https://software.intel.com/en-us/articles/introduction-to-intel-advanced-vector-extensions