

# Software and Hardware Performance Tuning for Cryptographic Hash Functions

Michael Cloppert (Author), Tyler Newton (Contributor)

December 13, 2005

## 1 Abstract

In this document, the cryptographic hash algorithms SHA-1, MD5, and RIPEMD-160 are analyzed on a simulated superscalar processor. Performance enhancements including processor design, cache organization, and compiler optimizations are investigated for each algorithm, and for the cryptographic hash function application domain in general.

## 2 Experiments

The goal of this investigation was to address performance challenges for the entire secure hash algorithm application domain. To achieve practical results, three of the most common hashing algorithms were analyzed, in their most popular implementations:

- The MD5 algorithm, developed by Ron Rivest of RSA Security[1]
- The SHA-1 algorithm, developed by the NSA[2]
- The RIPEMD-160 algorithm, developed by Hans Dobbertin, Antoon Bosselaers, and Bart Preneel[3]. This will be referred to as RIPEMD throughout this document, not to be confused with the original RIPEMD based on MD4.

The SHA-1 and RIPEMD-160 algorithms are 160-bit cryptographic hashing algorithms, meaning a brute-force attack would theoretically require  $2^{80}$  attempts to find a collision (this is a measure of the algorithm's strength). MD5 uses a 128-bit digest algorithms with a theoretical complexity of  $2^{64}$ . While SHA-1 and RIPEMD-160 are theoretically more secure, the three algorithms are close in complexity and used interchangeably in practice, and will therefore be compared side-by-side in this study. The superscalar processor simulator "sim-outorder," by SimpleScalar LLC, was used

to test the algorithms on different hardware configurations, and basic gcc compiler optimizations were investigated as well. Where applicable, gcc and sim-outorder parameters used for each test are given. All experiments computed the hash of a plain-text version of Shakespeare’s *Hamlet*[4]. The order of optimization tests for all algorithms was

1. Functional unit and processor optimizations. All sim-outorder defaults were taken except the parameter being tested.
2. Cache-related optimizations. Only cache-related parameters were varied, and all other sim-outorder defaults were taken. All tests assumed separate data and instruction caches, configured identically. Cache optimizations focused on:
  - (a) Determining impact of block size by varying this parameter and keeping others static for a variety of cache sizes.
  - (b) Determining impact of associativity by varying this parameter and keeping others static.
  - (c) Determining impact of a level-2 cache by adding and removing this feature for a variety of cache sizes.
3. Compiler optimizations. The -O1, -O2, and -O3 optimization packages in gcc were compared for performance. Unless otherwise noted, all other tests used binaries compiled with the -O2 optimization.

Performance improvements identified were then combined to produce a fully-optimized set of hardware specifications and compiler optimizations for each algorithm. Finally, common high-impact enhancements common to all algorithms were identified and used to create a set of hardware specifications and compiler optimizations for the entire application domain. The specific implementations of the algorithms tested in these experiments were borrowed and modified. All algorithms were tested for accuracy using common benchmarks, and their original authors are credited in the code itself[3, 5].

### 3 Results

Varying the number of floating point and integer ALU’s available in the processor was ineffective as a performance enhancement for all algorithms. Surprisingly, the effect of branch prediction algorithms on overall performance was also muted, as shown in Table 1. The perfect branch prediction property illustrates only slight performance enhancement can be expected with the most effective predictions, making additional tests unnecessary. Note that the ”perfect” branch prediction algorithm (-bpred

perfect) cannot be implemented in a practical sense, and was not considered in identifying the final optimizations for each algorithm.

Table 1: Speedup and branch prediction accuracy. Data is in the format "speedup/accuracy"

<b>Alg</b>	<b>Taken</b>	<b>Not Taken</b>	<b>Combining Predictor</b>	<b>Perfect</b>
MD5	-0.064794/22	-0.065552/22	0.000932/95	0.002913/100
RIPEMD	-0.060276/11	-0.060581/11	0.000076/94	0.003581/100
SHA-1	-0.140480/18	-0.142513/18	0.011202/96	0.012613/100

Increasing the memory access bus width (-mem:width parameter) yielded only minor performance gains for each algorithm, with speedups of 0.4%, 0.2%, and 0.3% for MD5, RIPEMD, and SHA-1, respectively, when increased from 16 to 32 bits. Similarly, only slight performance increases were realized by adjusting the number of memory ports (-res:memport parameter) to 4 or 6 from the default setting of 2. Enhancing the register update unit, or RUU, was the single most effective overall optimization for SHA-1 (12.9% speedup), while its impact on RIPEMD (1.4%) and MD5 (0.1%) was measurable, but much smaller. These performance increases came when the RUU was raised to 32 from 16 entries by default (-ruu:size parameter).

Cache optimizations yielded the largest performance gains for the application domain as a whole. Initial tests focused on block size. For reasons that could not be determined, testing of 128-bit blocks was not possible in the simulator. Figure 1 shows performance characteristics for varying block sizes in each algorithm. For each cache size, the 64-byte blocks showed the lowest miss rate, and for each algorithm, the 256KB cache gave the lowest miss rate. Even though instruction and data caches were separate in this experiment, the data presented in Figure 1 is normalized across instruction and data misses, computed using the equation  $HitRate = \frac{hits_{instr} + hits_{data}}{accesses_{instr} + accesses_{data}}$ . Note that all cache tests, unless otherwise noted, included a default 256KB level 2 unified cache. For larger L1 cache sizes, this size of L2 cache was less appropriate and may have skewed those test results slightly.

As cache size increased, the impact of associativity on the algorithm’s performance diminished. Nearly optimal memory performance was achieved with either 256KB direct-mapped cache, or 64KB fully associative cache, for every algorithm. Table 2 shows the results of cache associativity testing for each algorithm. With the knowledge that the default configuration of sim-outorder is a L1 hit rate of 1 cycle for instruction and data, unified L2 cache with a miss penalty of 6 cycles, and memory latency of 30 cycles, average memory access time (AMAT) for the table was computed using Equation 1.[6]

$$AMAT = T_{hit} + MP_{L1} * \frac{Miss_{L1_{instr}} + Miss_{L1_{data}}}{accesses_{L1_{instr}} + accesses_{L1_{data}}} + (MP_{L1} + MP_{L2}) * \frac{Miss_{L2}}{accesses_{total}} \quad (1)$$

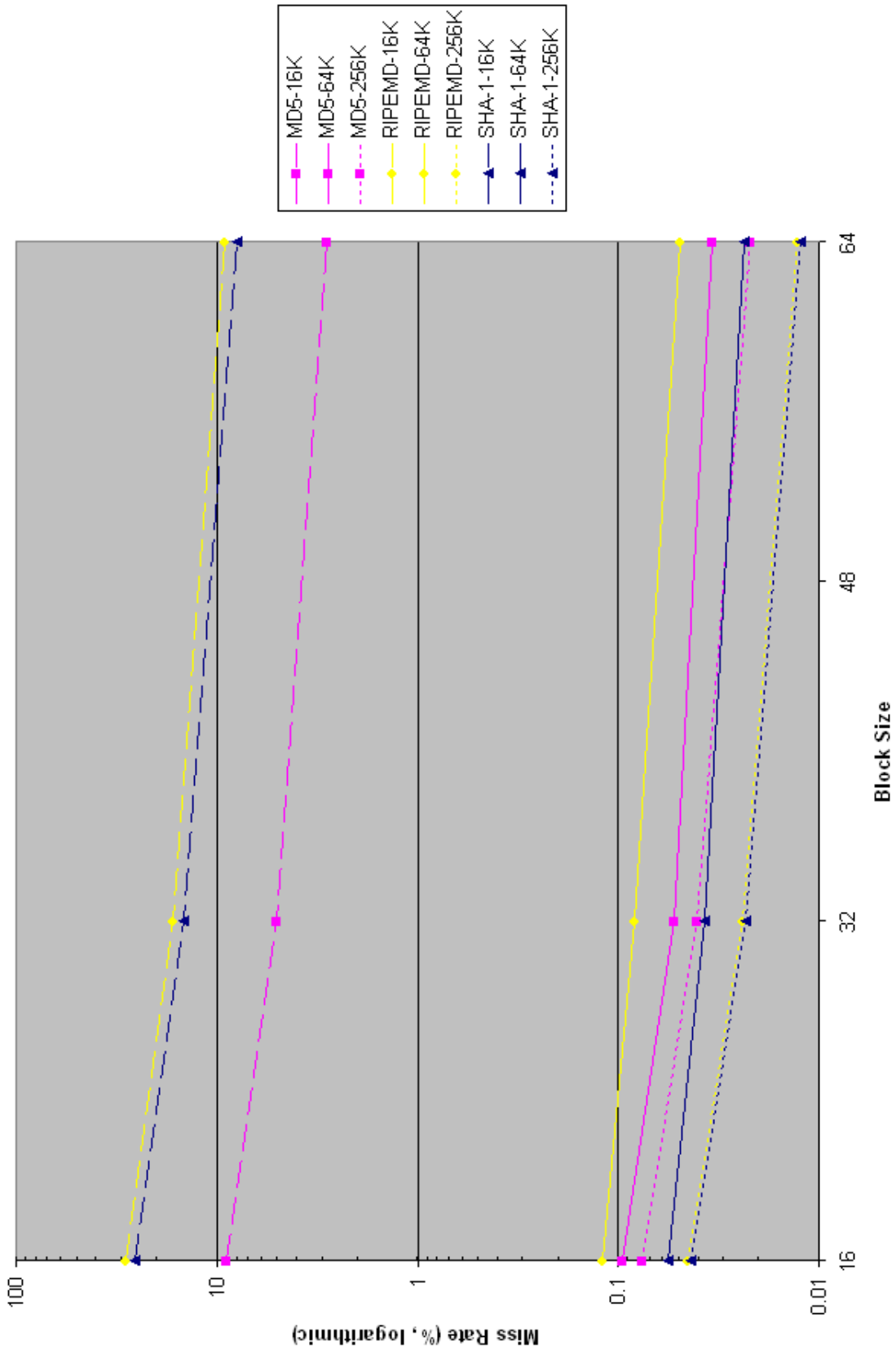


Figure 1: Impact of cache and block size on miss rate. Note that associativity was 2 using LRU replacement strategy.

Table 2: Average memory access time (in CPU cycles) of various levels of associativity, by algorithm and cache size.

Alg	Cache Size	Direct-mapped	2-way Assoc	4-way Assoc	Fully Assoc
md5	16	1.130964	1.175980	1.266215	1.539582
md5	64	1.016529	1.007724	1.007622	1.007028
md5	256	1.007012	1.007012	1.007012	1.007012
rmd	16	1.551492	1.550240	1.549903	1.549528
rmd	64	1.011563	1.006301	1.005142	1.004132
rmd	256	1.004115	1.004115	1.004115	1.004115
sha1	16	1.450335	1.479172	1.478745	1.478397
sha1	64	1.009245	1.004609	1.004433	1.003946
sha1	256	1.003937	1.003937	1.003937	1.003937

Level 2 cache test results were compared based upon AMAT once again using Equation 1. Figure 2 is an illustration of the results. In all cases, the level 2 cache was unified, and double the size of the level 1 cache following the level 2 cache sizing rule of thumb.[6] These results are skewed, however. The L2 cache testing went through a number of iterations in sim-outorder, but no combination of switches to completely disable this cache was found (it was expected that the switches "-cache:il2 none -cache:dl2 none" would achieve this, but level 2 hits were still found in simulation output with these parameters given). As a result, this data is provided only for completeness, and no conclusions are drawn from it. Given the minor performance impact shown in Figure 2, level 2 cache is ignored as a factor in comparing results for this paper.

Basic, extended, and advanced compiler optimizations were compared using gcc's standard -O1, -O2, and -O3 optimization packages, respectively. All algorithms experienced significant performance gains with the most basic optimizations, and none benefitted from loop unrolling and more advanced optimizations in -O2 and -O3 with -funroll-loops. Repair code from loop unrolling and code motion seems to have offset the benefits from these optimizations in all cases. Figure 3 illustrates these findings and relative speedup data. Speedup was calculated assuming a constant clock speed between the execution of baseline code and compiler-optimized code, as shown in Equation 2.

$$Speedup = \frac{CPI_{orig} * IC_{orig}}{CPI_{enhanced} * IC_{enhanced}} \quad (2)$$

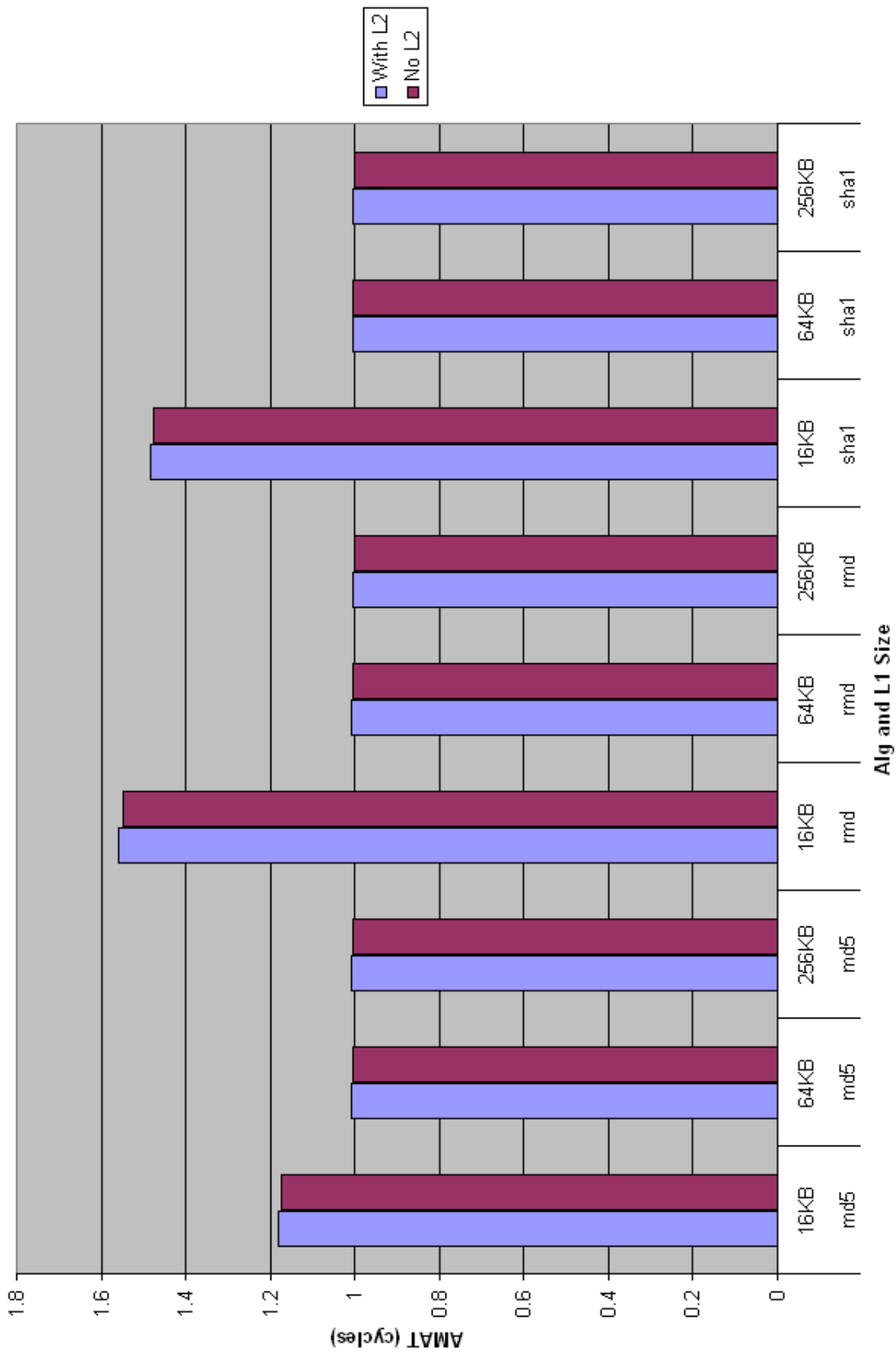


Figure 2: Results of testing sim-outorder with limited versus appropriately-sized level 2 cache configurations.

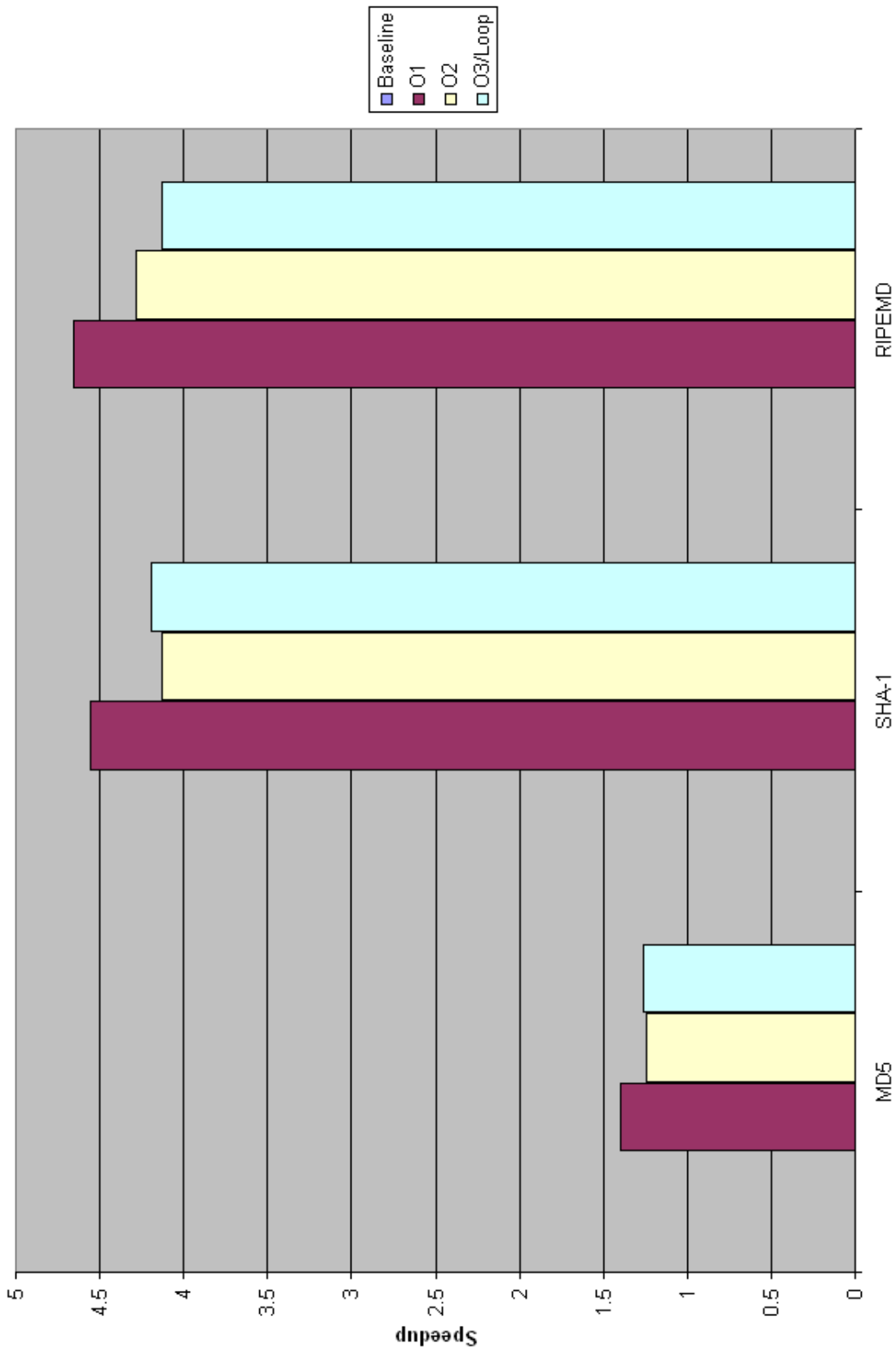


Figure 3: IPC comparison for compiler optimization packages and their impact on algorithms studied.

## 4 Conclusions

Processor configurations, cache properties, and compiler optimizations that yielded improved performance were aggregated for each algorithm. The configurations and their combined results are shown in Table 3.

Table 3: Results by algorithm of the algorithm’s optimal configuration, IPC, and speedup. ( \* = sim-outorder simulator default)

<b>Property</b>	<b>MD5</b>	<b>RIPEDM</b>	<b>SHA-1</b>
<b>L1 Cache Size</b>	256KB	256KB	256KB
<b>L1 Cache Assoc</b>	Direct-map	Direct-map	Direct-map
<b>L1 Cache Block Size</b>	64-byte	64-byte	64-byte
<b>Compiler Optimization</b>	Basic (O1)	Basic (O1)	Basic (O1)
<b>Memory Access Bus</b>	32-bit	32-bit	32-bit
<b>RUU Size</b>	16 *	16 *	32
<b>Branch Alg</b>	Combining	Combining	Combining
<b>Memory Ports</b>	4	2 *	4
<b>Load-store queue</b>	8 *	8 *	16
<b>Speedup</b>	<b>1.458</b>	<b>6.646</b>	<b>5.852</b>

The optimizations that yielded performance gains across all algorithms, or on one algorithm without adversely affecting the others, were aggregated across the application domain and applied to each algorithm. The gcc compiler optimization -O1 gave consistently the best results, and was chosen to compile the algorithms. The hardware consisted of a 256KB, 64-byte block, direct-mapped cache (-cache:il1 il1:2048:64:1:l -cache:dl1 dl1:2048:64:1:l), along with a 32-bit memory bus (-mem:width 32), combining branch predicting algorithm (-bpred comb), a 32-entry register update unit (-ruu:size 32), and 4 processor memory ports (-res:mempport 4). The performance increase for each algorithm optimized with these settings is shown in Table 4.

Table 4: Speedup of each algorithm when optimized for the Cryptographic Hash application domain.

<b>Alg</b>	<b>Speedup</b>
MD5	1.457
RIPEDM	6.752
SHA-1	5.752

As before, speedup was calculated using Equation 2. It’s interesting to note that some optimizations chosen for the application domain, when applied to RIPEDM, resulted in even better performance than optimizations customized specifically for that algorithm.

The cache optimization results were roughly equivalent for all algorithms between a 64KB, fully-

associative cache and a 256KB, direct-mapped cache, both with 64-byte blocks. While the larger cache will take up more chip space, the logic behind a fully-associative cache is much more complicated, and implementing the replacement algorithm (LRU was tested) in reality would most likely impact performance so as to make this a less wise choice. Cache optimization was such an important factor for this application domain due to the nature of cryptographic hashing algorithms, which typically iterate through blocks of memory, performing simple operations on the block, taking the result, and moving on to the next block. This process is often repeated, meaning that a larger direct-mapped cache will result in lower capacity and conflict cache misses, helping performance even while incurring the expense of higher compulsory misses initially. This also helps explain the impact of a larger RUU. These general observations are supported by data for smaller cache configurations in Table 1. RIPEMD and SHA-1 perform more operations on a block than MD5, and are more sensitive to this problem, which is manifested as a higher AMAT in this table.

Additional testing could investigate the impact of the digest (or hash) length on the performance characteristics analyzed in this paper. As weaknesses have been found in all of these algorithms recently, longer hashes will become more common in the future, making such data useful. The specific results of this study, however, can serve as guidance to engineers implementing hardware-based cryptographic hash solutions in embedded devices or FPGA's in the near-term.

## References

- [1] Ronald L. Rivest. *The MD5 Message Digest Algorithm*. <http://www.ietf.org/rfc/rfc1321.txt>.
- [2] NIST. *FIPS PUB 180-1: Secure Hash Standard*. <http://www.itl.nist.gov/fipspubs/fip180-1.htm>.
- [3] Antoon Bosselaers. *The hash function RIPEMD-160*. <http://homes.esat.kuleuven.be/bosselaer/ripemd160.htm>, 2004.
- [4] Project Gutenberg. *The Complete Works of William Shakespeare*. <http://www.gutenberg.org/etext/100>.
- [5] David Ireland. *NIST SHA-1 Compliant Algorithm*. <http://www.di-mgt.com.au/src/sha1.c.txt>.
- [6] John L. Hennessy. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, 3 edition, 2003.