White Paper

Vinodh Gopal
Jim Guilford

IA Architects
Intel Corporation

A Novel Hashing Method Suitable for Lookup Functions

February 2012

326841-001

Executive Summary

The paper describes a novel hashing method suitable for lookup functions. Traditional methods are either of lower hash quality or inferior performance. In this paper, we show how to compute a high-quality hash digest at extremely high performance on Intel® processors using the crc32 instruction. Our method also generates multiple 32-bit hashes of an input data buffer, which have low correlation.

This paper describes a novel hashing method suitable for lookup functions, based on the crc32 instruction. We show how to compute a high-quality 64-bit hash digest at extremely high performance on Intel® processors. For large data buffers, the performance of our hash function tends to ~ 0.44 cycles/byte on a **single thread** of an Intel® Core™ i5 650 processor. Even for small buffers (~24 bytes), the performance is better than 1 cycle/byte.

Contents

Overview

This paper describes a novel hashing method suitable for lookup functions. Traditional methods are either of lower hash quality or inferior performance. In this paper, we show how to compute a high-quality hash digest at extremely high performance on Intel® processors using the crc32 instruction [3]. Our method also generates multiple 32-bit hashes of an input data buffer, which have low correlation and is useful in contexts such as network processing applications.

Background of Hashing for Lookups

Some of the best methods to perform hashing for lookups such as lookup, lookup2 and lookup3 have been developed by Bob Jenkins [1]. The requirements vary based on applications, but for the most critical usages in network processing, we need to generate a hash digest of an input data buffer of typically small size. In some instances, the data buffers are all fixed length, however in others they vary. Performance of the hash function and the hash quality are both critical. The hash function does not require cryptographic strength.

The digest size can be larger than 32 bits, such as 64 bits. In some usages, we need to generate a pair of 32-bit hash digests that have low correlation in addition to being individually strong. The general requirement is that we need a hash digest whose size is N*32 bits or a set of N 32-bit digests of low correlation. For the special case of 32-bit hashes, we can use the crc32 instruction, but the question is how to extend the power/speed of the crc32 instruction to the general case.

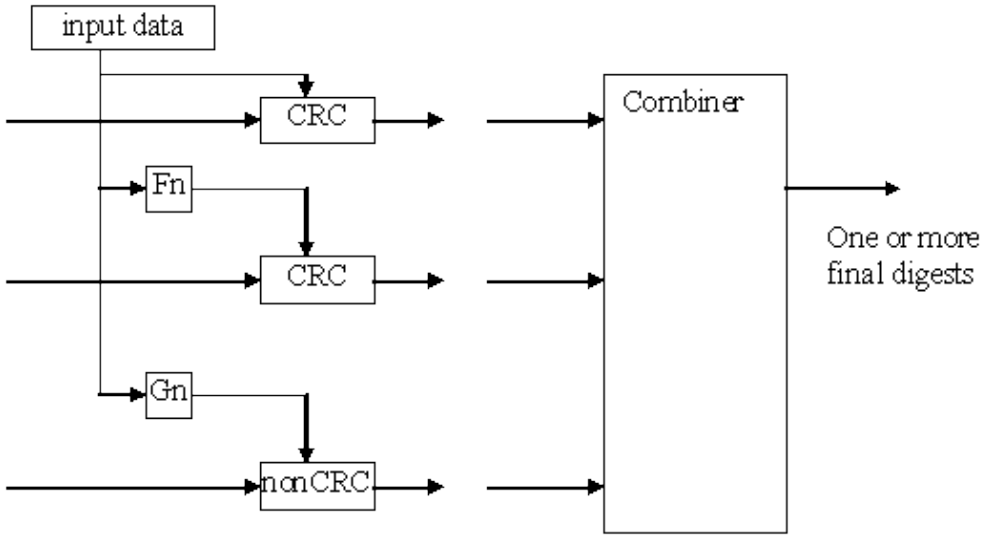
We describe a framework to achieve this and show the details of one such function that generates 64-bit digests (or pairs of 32-bit digests) called intel_hash64. Further details can be found in the patent application [2] on which the methods are based. We measured performance as the average cycles and the average rate in cycles / byte to hash a large number of unique data buffers.

In addition to checking the strength of our proposed hash function, we also checked the quality of a hash function derived from a cryptographic hash for reference (SHA1 based). One of the best known tests, the frog test [1], toggles bits in an attempt to generate digest collisions. We tested the hash functions for collisions with each individual 32-bit digest, and with a 64-bit digest created by concatenating the two 32-bit digests. The SHA1 algorithm generates a 160-bit digest, from which we had to form a 64-bit digest of the highest quality as measured by the frog test (for 32-bit digests). The SHA1 digest can be viewed as comprising five 32-bit digests, and we picked the best performing one (bits 127:96) as our first 32-bit digest. We constructed

our 2nd 32-bit digest from the entire SHA1 digest with the function: SHA[31:0] ⊕ SHA[64:32] ⊕ SHA[127:96] – (SHA[95:64] ⊕ SHA[159:128]). We found this function yielded the best quality, on par with the 1st digest.

Description of the crc32-Based Hash method

Figure 1: Overall crc32-Based Method for Hash Functions



In Figure 1, we show the generalized flow for computing multiple 32-bit digests:

- An intermediate digest computed as the CRC of the data
- Some number of intermediate digests computed as the CRC of a modified form of the input data (using functions F1, F2, …)
- Some number of intermediate digests computed by non-CRC operations (e.g., xor, add) on a modified form of the input data (using functions such as G1, G2, etc.)
- A combination step that takes these intermediate digests and combines them with each other and possibly other data such as the input length to produce multiple final digests or a single larger digest

This generalized method is mainly intended for usages where we need at least two 32-bit digests. If we needed just one digest, the crc32 of the input data is a good digest.

Figure 2: Example of a crc32-Based Hash Function Returning two Digests

```
C = length of data <<< 19
while (more than 8 bytes of data left){
   data_chunk = next 8 bytes of data
      A = CRC (A, data_chunk)
      B = CRC (B, data_chunk <<< 31)
      C = C ⊕ data_chunk
}
Return A, CRC(B,C)
```

The proposed intel_hash64 function is based on the pseudo-code shown in Figure 2. In the terminology of Figure 1, we can see that the F1 function is rotate-left-by-31 (**<<< 31**) and the G1 function is the trivial identity function (i.e., output is the same as the input). The non-crc accumulating function is a bit-wise XOR operation (⊕). The combiner block returns a computed crc (A) directly as the 1st digest, but performs another final crc operation using the two intermediate digests B, C, and returns CRC(B,C) as the 2nd digest.

Performance

The performance results provided in this section were measured on an Intel® Core™ i5 650 processor at a frequency of 3.20 GHz, supporting Intel® crc32 instruction. The tests were run with Intel® Turbo Boost Technology off, on a **single thread**.

Methodology

We measured the performance of the hash function on data buffers of different sizes. We called the function to hash the same buffer a large number of times, collecting many timing measurements. For each data buffer, we then sorted the timings, discarded the top and bottom quartiles and averaged the rest.

The timing was measured using the **rdtsc**() function which returns the processor time stamp counter (TSC). The TSC is the number of clock cycles since the last reset. The 'TSC_initial' is the TSC recorded before the function is called. After the function is complete, the **rdtsc**() was called again to record the new cycle count 'TSC_final'. The effective cycle count for the called routine is computed using

# of cycles = (TSC_final-TSC_initial).

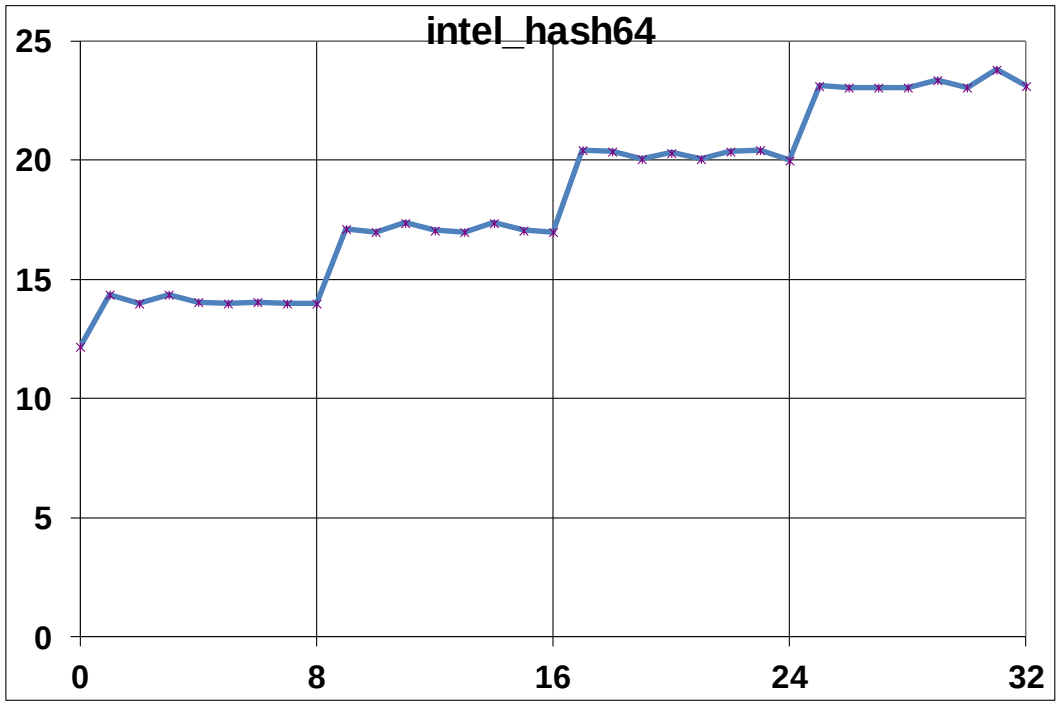
A large number of such measurements were made for each data buffer and then averaged as described above.

Note: *Performance tests and ratings are measured using specific computer systems and/or components and reflect the approximate performance of Intel products as measured by those tests. Any difference in system hardware or software design or configuration may affect actual performance. Buyers should consult other sources of information to evaluate the performance of systems or components they are considering purchasing. For more information on performance tests and on the performance of Intel products, Go to:*
http://www.intel.com/performance/resources/benchmark_limitations.htm

Results

We show performance with a single thread in cycles for varying sizes of input data buffers with a single thread, for small buffer sizes.

Figure 3: Cycles vs. Data Buffer Size in Bytes with Single Thread



For large buffers, the performance tends to ~ **0.44 cycles/byte**. As can been seen from the graph, even for small buffers (~24 bytes), the performance is better than 1 cycle/byte. As the crc32 instruction has a latency of 3 cycles, and a throughput of 1 cycle, we anticipate that Intel® HT Technology will bring a substantial additional performance gain on applications using this hash function.

Testing for Quality

Using the frog test, we tested our hash function and the reference one derived from SHA1 for quality. The test sets bits in the input data in an attempt to generate digest collisions through the careful, similar construction of input. All the inputs to the hash function are the same fixed length, and consist of zero bytes with very few bits set. The original program tested 64-bit digests. We modified this to also test 32-bit digests. Results for this test are reported as the logarithm of the highest number of data pairs tested before the first collision is found. The higher the score, the better is the hash quality. The tests are exponential in nature, running for weeks if a collision cannot be found within 2^{67} pairs.

Figure 4: Results of the hash quality tests

	frog_64	frog_32	
		Digest1	Digest2
sha1	**>=68**	33	33
intel_hash64	>=67	**39**	**39**

Due to run-time of the tests, the frog_64 test results are incomplete, and the ">=" implies that no collisions were found at that point, but the test was still running. Note that the frog_32 tests are much faster and **intel_hash64 is much better than SHA1 for 32-bit digests**.

For the 64-bit collision tests, both functions show extremely strong quality since a completely random mapping to 64-bit values is expected to result in a collision by 2^{63} pairs. It is also noted in Bob's article that both lookup.c and lookup2.c start seeing collisions after 2^{53} key-pairs, whereas lookup3.c gets its first collision shortly after 2^{63} pairs.

Conclusion

In this paper, we show how to compute a high-quality hash digest at extremely high performance on an Intel® Core™ i5 650 processor using the crc32 instruction. We describe a general method using CRC calculations and show how that can be used to generate larger digests than the degree of the CRC Polynomial. The functions can generate larger digests or multiple uncorrelated digests.

It is possible to develop higher-performance functions for specific usages that target very large data buffers or work only on fixed sized buffers, but such

optimizations have not been attempted in the current versions of our functions.

Contributors

We thank Schuyler Eldridge, Gil Wolrich, Erdinc Ozturk and Wajdi Feghali of Intel Corporation for their substantial contributions to this work.

References

[1] Bob Jenkins' Hash functions http://burtleburtle.net/bob/

[2] US Patent Application: "A Novel Hashing Method suitable for Lookups" *Vinodh Gopal, Jim Guilford, Schuyler Eldridge, Gil Wolrich, Erdinc Ozturk, Wajdi Feghali*

[3] Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 2A: Instruction Set Reference, A-M
http://www.intel.com/products/processor/manuals/

Authors

Vinodh Gopal and **Jim Guilford** are IA Architects with the IAG Group at Intel Corporation.

Acronyms

IA	Intel® Architecture